Fred Mintzer
Abraham Peled

A Microprocessor for Signal Processing, the RSP

Signal processing is a data processing domain that contains a diversity of applications, including speech processing, image processing, radar, sonar, medical imaging, data communications, seismic processing, and many others. Despite the diversity of the applications, this processing domain has a very structured set of characteristics. These include real-time operation, dominance of arithmetic operations, and well-structured data flows. The Real-Time Signal Processor (RSP) is a microprocessor architecture that was created to exploit these characteristics in order to provide an expeditious and economical way to implement signal processing applications. In this paper, the organization and architecture of the RSP are described. Features of the RSP, such as the instruction pipeline and the fractional fixed-point arithmetic, which exploit the characteristics of signal processing to provide additional computational power, are emphasized. Other features, such as the powerful indexing, the saturation arithmetic, the guard bits, and the double-word-width accumulator, which add much to the processor's versatility and programmability, are also highlighted. The performance of the RSP is illustrated through examples.

Introduction

Digital signal processing is an application domain of computing that includes a great diversity of applications. In the realm of man/machine interfacing, applications include speech recognition, audio response, and image compression/expansion. In the realm of communications, applications include data transmission and voice coding. In the medical area, applications include x-ray imaging, ultrasonic imaging, and the processing of EEG (electroencephalograph) and EKG (electrocardiograph) signals. Defense applications include radar, sonar, and other forms of electronic surveillance. There are many other areas where digital signal processing is also applied, including oil exploration, speech enhancement, television signal improvement, audio enhancement, and speaker verification. In [1], a survey of the major applications is presented, with a chapter devoted to each.

Despite the diversity of applications, however, there is a common set of processing characteristics associated with this application domain. Let us consider an example, Adaptive Transform Coding (ATC) of speech [2], in order to focus on these characteristics. ATC is a technique that is used to reduce the number of bits required to express the information content of speech. For normal telephone transmission, speech is coded at rates of 56 or 64 kilobits per second (kbps). However, ATC, with modifications, has been successfully used to encode speech at rates from 12 kbps to 16 kbps with good quality [3, 4]. Thus, in storing speech for later playback, ATC reduces the storage requirement by more than a factor of three. Similarly, a digital channel dedicated to speech could exploit ATC to transmit three times the number of conversations.

A block diagram of the processing that takes place in ATC is shown in Fig. 1. Let us first examine the coder. It is seen that the analog speech waveform is first filtered by an analog low-pass filter (LPF) and then sampled by an analog-to digital converter (ADC) to form the input data stream. For error-free reconstruction, the sampling rate must be at least twice the highest frequency present in the incoming signal. Typically, the analog filter is chosen to filter out frequencies above 3.2 kHz, and the speech is sampled at an 8-kHz rate. Following digitization by the ADC, the speech is passed through a "window," which selects a finite block from the continuous signal and shapes it. Usually, the selected blocks of data overlap and the window tapers the data at the ends of the blocks in such a way that the windowed blocks sum to the input stream. The output points of the window are then input to a Discrete Fourier Transform (DFT) or other fast trans-

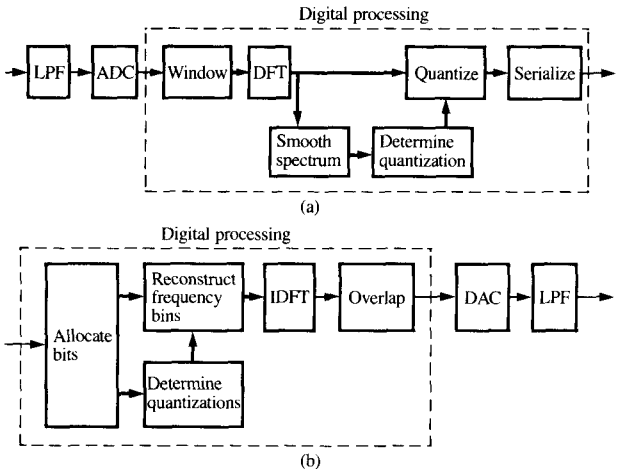

Figure 1 An adaptive transform coder (a) and decoder (b).

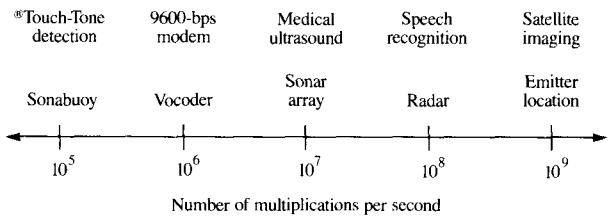

Figure 2 Computational requirements of a selected set of applications. (Note: ®Touch-Tone is a registered trademark of AT&T.)

form, which computes the spectrum of the windowed block. The DFT output points are then quantized. Those regions of the DFT that have more energy, as determined by spectral averaging or smoothing, are allocated more bits. Those regions of the DFT that have less energy are allocated fewer bits or no bits at all. Finally, the quantized DFT output points and the bit-allocation information are combined and serialized, forming an output bit stream.

The ATC decoder performs essentially the inverse operation. First, the bit-allocation information is extracted and the DFT points are reconstructed. Then an inverse DFT (IDFT) operation is performed. The output blocks of the IDFT are then overlapped and added to reconstruct the input data stream. If the transmitter window is properly chosen, no audible discontinuities at the block boundaries will be present. Analog speech is reconstructed by passing the output data through a digital-to-analog converter (DAC) and an analog low-pass filter.

Let us now examine some of the characteristics of this processing.

1. The processing has a real-time speed requirement. For the ATC example, if the processor cannot process the input data as fast as the ADC supplies it, the coder will not operate correctly.
2. The processing is essentially arithmetic. There are few data-dependent branches. A large percentage of the calculations is devoted to filtering and DFT calculations. In the ATC coder, for example, the windowing, the DFT, and the spectral smoothing dominate the processing load. Yet there is not one data-dependent branch in all of this processing.
3. High computation rates are required. A rough sizing of the processing requirements, using a Fast Fourier Transform (FFT) algorithm to implement the DFT [5], a 21-tap spectral-smoothing filter, and a block size of 256 points with 20% overlap, reveals that the ATC coder requires approximately 100 000 multiplications and 150 000 additions per second. Even for fairly simple signal processing, millions of multiplications per second are often required.
4. Most operations are linear, plus or minus truncation error. For these operations, each output point is a linear combination of the input points. This is true of the windowing, the DFT, the IDFT, and the spectral smoothing of the ATC coder and decoder. Because of this, the dynamic range of the data does not greatly change.
5. The data flows are periodic. Most processing blocks consist of a single input stream and a single output stream. This is certainly true of the processing blocks of the ATC coder and decoder.
6. The processing decomposes nicely into sequential and parallel subprocesses. The block diagram of the processing demonstrates this at a certain level of detail. The processing blocks could also be similarly decomposed, if necessary [6].

There are certain aspects in which signal processing applications differ, however. Perhaps the most important of these is in the range of the computational requirements. Although it is difficult to specify exactly what the computational requirements are for any application, because this differs according to the algorithms used, it is possible to size the application well enough to find what order of magnitude of computation is required. In Fig. 2, sizings are shown for a small set of signal processing applications. Commercial applications form the top row, and defense applications form the bottom row. For both categories, the computational requirements vary by a factor of 10 000 to 1!

The precision required of the computations also varies with the application. While for some applications 8 bits of

output precision are sufficient, for others 20 bits are insufficient. The system complexity of the implementation also varies. Some of the least demanding applications, computationally, will undoubtedly require only a single-processor solution, the simplest system possible. Others, with multiplication requirements in the billions-per-second range, will undoubtedly require multiprocessor implementations, independent of the processor chosen. These systems will have inherently greater complexity.

The interested reader will find that several good texts on digital signal processing are currently available; [7] provides a good description of the field, with particular emphasis on practical problems and hardware; [8] is intended as an undergraduate text; [9] is intended as a senior/graduate level text; [10] is also suitable as a graduate text, but is perhaps a more complete reference.

There were several pioneering efforts in the late sixties and early seventies aimed at building signal processing hardware for digital filters and FFTs, for example [11–14]. These developments were inspired by the appearance of commercially available SSI and MSI digital hardware, but they were also limited by its capabilities. The devices constructed were essentially single-purpose boxes of considerable size, and they required considerable effort to produce each replicate. The appearance of LSI technologies in the seventies offered special opportunities for signal processing implementations that are economical with respect to cost, space, and power consumption. The Real-Time Signal Processor was developed to capitalize on LSI technology to provide expeditious and economical implementations for signal processing applications.

Also during the seventies, new algorithms for Fourier Transforms and convolutions (filtering) were discovered by Winograd, Cooley, and Agarwal at the IBM Thomas J. Watson Research Center [15, 16]. The new algorithms reduce the number of multiplications required for these operations dramatically, although they do require a more complicated program flow. In Table 1, a comparison is given of the computational requirements for the previously known Fast Fourier Transform (FFT) algorithm and the Fourier Transform algorithm discovered by Winograd (WFT). It was hoped that the RSP would be able to exploit the new algorithms, in addition to the characteristics of signal processing, to achieve increased performance. In another paper in this issue [17], Cooley demonstrates the advantages of the rectangular transforms for the RSP.

Capitalizing on LSI technology has two implications. The most obvious is limiting the hardware complexity to the capabilities of the technology. Many features were excluded from the RSP, not because they were undesirable, but because their inclusion would have required exclusion of other more important features. The other implication of LSI technology is that, to capitalize on LSI, a large enough volume of chips must be produced so that the per-chip contribution to the development cost is small. This requires that the RSP have the flexibility to satisfy a broad range of applications exhibiting the diversity of signal processing. To do this it had to be fully programmable, economically connectable into systems, and able to provide the precision required.

Table 1 Computational requirements of Fourier Transform algorithms.

Fast Fourier Transform (FFT)			*Winograd Fourier Transform (WFT)*		
N	*Mult.*	*Add.*	*N*	*Mult.*	*Add.*
128	1152	2368	120	288	2076
256	2304	5248	240	648	5016
1024	12288	26624	1008	4212	25224

The cost of the hardware in a signal processing system may be only a small fraction of the cost, especially if few versions are constructed; the cost of generating the application software can dominate both the system cost and the system development time. To provide both a quick and economical solution, the RSP had to be made easy to program. Many features were included in the RSP hardware to make it easy to program, even though they added little to the raw performance of the processor.

In the following sections, the organization, architecture, implementation, and performance characteristics of the RSP are described. The emphasis is on those features of the RSP that make it powerful for the signal processing application domain, versatile enough to satisfy a broad application domain, and easy to program. Due to the limits of technology, not all of the desired features could be included on the RSP chip. The rationale for the inclusion of those RSP features that were chosen is also given.

Basic architecture

The organization of the RSP, as shown in Fig. 3, interfaces the external world through a parallel I/O interface. This interface provides data, address, and control lines. Normally, two data ports and one control port are attached to the RSP at this interface. The data ports manage the normal flow of data into and out of the RSP, while the control port is used for loading programs into the RSP, receiving commands from the system controller, and debugging. If fewer data or

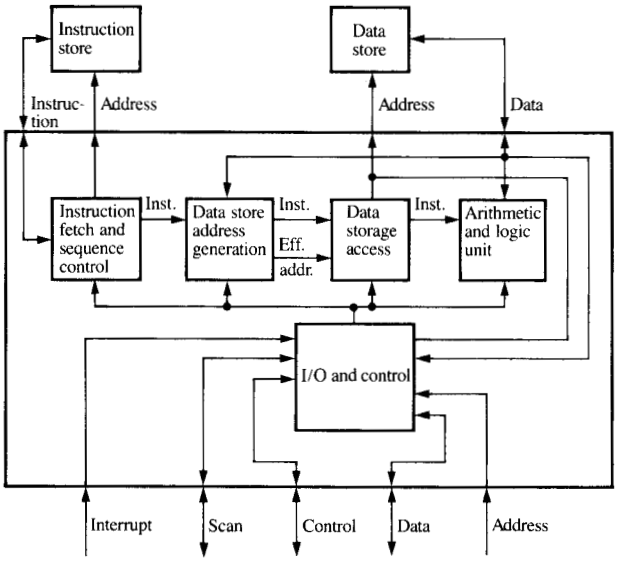

Figure 3 Organization of the RSP.

control ports are needed, the RSP is equipped with only those required. Scan lines are available for use in debugging, and a vectored maskable interrupt capability is also provided.

The RSP is a fully programmable signal processing machine, with external data and instruction storage. The separate external data and instruction stores each have sixteen bits of addressing. This provides the RSP with much flexibility. Since the RSP can access up to 64K words (K = 1024, word length = 16) of data and 64K instructions, it is not severely storage limited and can achieve its full performance without bending to storage constraints. The RSP also has the flexibility of using either RAM or ROM storage, or combinations of the two.

The RSP instructions are 24 bits wide, with 8 bits of op-code and 16 bits of operand. A total of 244 op-codes form the RSP instruction set. The single-op-code, single-operand format was chosen to simplify the RSP programming. It was felt that the increased performance offered by horizontal microcode was not worth the increased programming complexity that it entails. The single operand format also makes the RSP code a suitable target for a compiler.

As shown in Fig. 3, the RSP is structured into five functional subunits. They are (1) the instruction fetch and sequence control unit, (2) the data-store address-generation unit, (3) the data-store access unit, (4) the arithmetic and logic unit, and (5) the I/O and control unit.

Instruction pipelining

An "instruction pipeline" is used to increase the performance of the RSP. Most RSP instructions use each of the first four functional units once, but only one of them is used during any processor cycle. In the first cycle of execution of an instruction, that instruction is fetched. In the second cycle, the data-store address is computed. In the third, the data store is accessed, and in the fourth, arithmetic or logical operations take place. Thus, this is a load-preferred architecture. The four cycles that it takes each instruction to execute are called the four phases of that instruction, phase one through phase four, respectively, and the hardware units that execute them are called stage one through stage four. During each cycle, a new RSP instruction is initiated. However, while the newest instruction is using the stage-one hardware, previous instructions are using the stage-two, -three and -four hardware, as is shown in Fig. 4. To execute n instructions, as is seen from the figure, $n + 3$ cycles are required. Although each individual RSP instruction takes four cycles to complete, the effective throughput of the processor is one instruction per cycle.

The presence of the instruction pipeline creates certain "pipeline hazards" for the RSP. One of these can be observed in the following example. Suppose we wish to calculate

$$a = \max(a, b). \tag{1}$$

Coding this for the RSP, while assuming that each instruction completes before the next begins, yields

```
        LZ      b       Load contents of b into Z
                        register
        SY      Z,a     Y equals the contents of Z
                        minus contents of a
        BN      NEXT    If the result is negative,
                        branch to NEXT
        STZ     a       Store the contents of Z in
                        memory location a
NEXT                    Proceed with the rest of the
                        program
```

The operation of the instruction pipeline in executing the first three instructions of this code is shown in Fig. 5, where *e.a.* refers to the data-store *effective address* calculated. It is seen that the stage one unit must decide whether to take the conditional jump during time interval 3. But the subtraction upon which we intend that jump to be conditioned is not computed until time interval 5! If that code were executed as written, the jump would actually be conditioned on the state of the accumulator at the end of time interval 3, and results other than those intended would occur.

Fortunately for the programmer, the RSP software support, which is the subject of the paper by Davies and Ris in this issue [18], remedies this problem. The programmer can write the code, as previously discussed, assuming that each instruction completes before the subsequent instruction begins. The assembler then transforms this code to accommodate the RSP instruction pipeline by either adding NOP

(no op) instructions and/or reordering the code. For the subject example, the code produced is

LZ	b	Load contents of b into Z register
SY	Z,a	Y equals the contents of Z minus contents of a
NOP		No operation instruction
NOP		No operation instruction
BN	NEXT	If the result is negative, branch to NEXT
STZ	a	Store the contents of Z in memory location a
NEXT		Proceed with the rest of the program

The operation of the beginning of this code sequence is illustrated in Fig. 6. The jump decision is now made at the end of time interval 5, by which time the difference has been computed. Since the jump is conditioned on the proper result, the RSP code computes as intended.

However, the pipeline hazard has caused some inefficiency (some do not). As we noted in the introduction, however, signal processing is predominantly arithmetic; for signal processing, few pipeline hazards occur. Typically, inserted NOPs account for less than 5% of the executed cycles. Thus the instruction pipeline capitalizes on the nature of signal processing to increase performance. The increase in performance is by a factor equal to the ratio of the sum of the stage execution times to the maximum of the stage execution times, a factor of about 3 to 1! In general, other types of computing, especially those with abundant data dependencies, are not able to capitalize on this instruction pipelining due to the number of NOPs that must be inserted.

We note that separate instruction and data stores are necessary for the instruction pipeline to function as described. Because of this connection, they are important architectural features of the RSP.

RSP arithmetic/logic unit

Since the purpose of the RSP is to perform calculations, we begin our look at the five major subunits of the RSP by examining the arithmetic/logical unit, stage four. The RSP represents all data as pure fractions in two's-complement fixed-point arithmetic with 16 bits of precision. Thus, the datum d is expressed as

$$d = -b_0 + \sum_{i=1}^{15} b_i 2^{-i}. \qquad (2)$$

Fixed-point arithmetic has its limitations for computing in general and it does require more complicated code generation than floating-point arithmetic. However, it is adequate

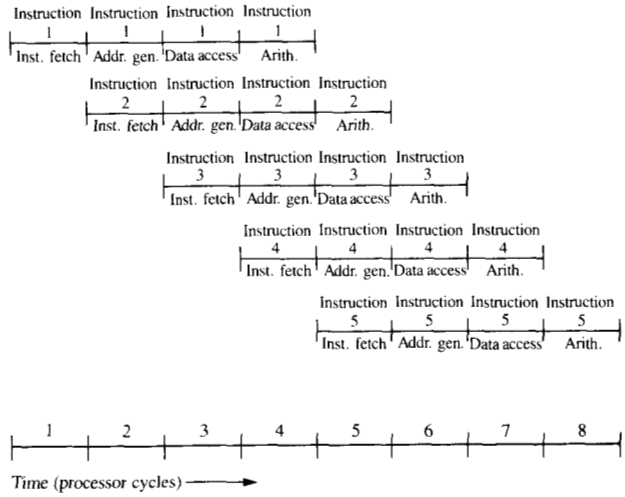

Figure 4 Operation of the RSP instruction pipeline.

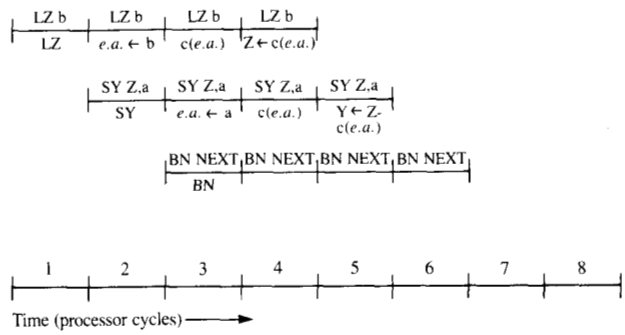

Figure 5 Operation of the RSP instruction pipeline on code kernel, executed as coded.

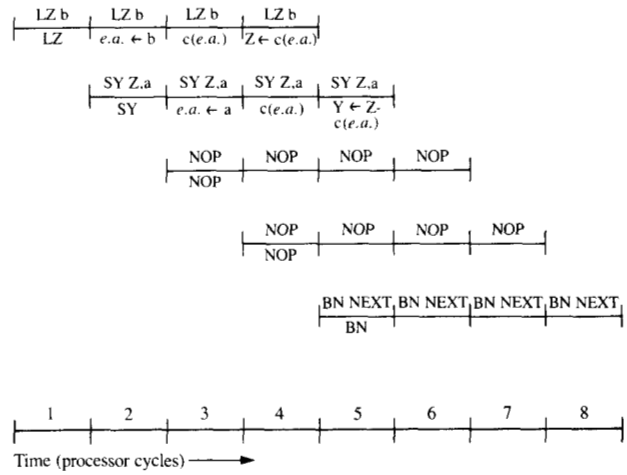

Figure 6 Operation of the RSP instruction pipeline on code kernel of assembled code.

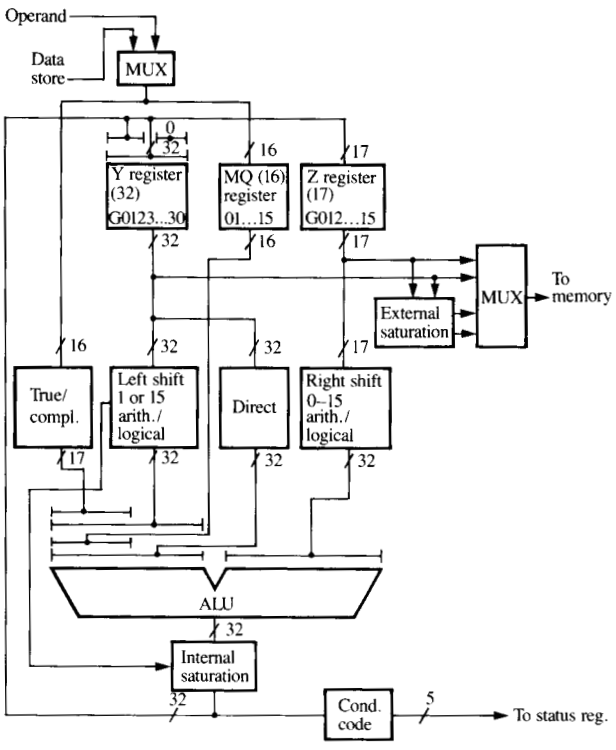

Figure 7 The RSP arithmetic/logic unit.

for most signal processing applications because of the limited dynamic range requirements noted in the introduction. The ability to use fixed-point arithmetic, as opposed to floating-point arithmetic, permits much simpler hardware and much greater processing speed. Thus the RSP's fixed-point arithmetic exploits the characteristics of signal processing to achieve increased performance.

The arithmetic/logic unit of the RSP is illustrated in Fig. 7. This unit has two main arithmetic registers, the Z register and the Y register. Multiplications in the RSP are mechanized by right-shifting the contents of the Z register and accumulating the partial sums in the Y register.

Thus the RSP executes multiplications as a series of shifts and adds rather than in a fast parallel multiplier. This is a highly controversial feature of the RSP, since many consider a fast parallel multiplier a necessity for a signal processor. This does entail some reduction of raw performance for the RSP and it does encourage the use of the RCC algorithms, which are unfamiliar to many and may be more difficult to program.

However, there were several reasons for this choice. Perhaps the most important is that a full-word parallel multiplier (16×16 bits with 31-bit result) would occupy considerable chip acreage. Presence of such a multiplier would have undoubtedly required the RSP to exclude many of the features that make it versatile and easy to program. It was also felt that the Reduced Computational Complexity (RCC) algorithms discovered by Winograd, Cooley, and Agarwal relieved the architecture of the necessity of providing a fast parallel multiplier. With these algorithms, the addition-to-multiplication ratio is usually in the range of 8:1 to 10:1. Their performance benefits only marginally from the presence of a fast multiplier. For example, a 1008-point complex Fourier Transform that has been coded for the RSP would require only about 10% fewer cycles if the RSP had a one-cycle multiplier. In practice, the RSP utilizes the RCC algorithms to capture some of the performance that might otherwise be provided by a fast multiplier. Lastly, a fast parallel multiplier could easily be added to the architecture, in a more advanced technology, to improve performance without requiring the application code to be rewritten. (This would require modifying the assembler to generate multiply instructions instead of shift/add sequences and re-assembling the application code.) The addition of other features to the architecture would not be nearly so forgiving.

Variable multiplications in the RSP are mechanized by the hardware two bits at a time by using a modified Booth's algorithm (pp. 182–183 of [7]). They require one cycle of throughput for each two bits of precision. The precision of the multiplication is set by a PRECISION instruction, and is thus under program control. Obviously, reduced precision requirements can thus be used to increase throughput. Coefficient multiplications, in the RSP, exploit the canonical signed-digit representation of filter coefficients to improve the RSP's performance. This is illustrated in the following example. The binary representation of the coefficient 0.4375_{10} is 0.011100000000000_2. This representation requires three shift/add instructions to execute the multiplication, one for each 1 in the representation. The canonical signed-digit representation is $0.100\bar{1}00000000000_2$, where $\bar{1}$ is -1. This representation requires only two shift/add or shift/subtract instructions. On the average, the canonical signed-digit representation reduces the number of shift/add and shift/subtract operations by about one-third. Our experience has been that for most finite-impulse-response (FIR) filters of interest, an average of only two to four shift/add or shift/subtract operations per coefficient is needed. More details of this technique are given in [19].

The generation of shift/add sequences would be a horrible burden if placed on the programmer. However, the RSP assembler automatically generates the optimal sequences from COEFFICIENT statements in the application code and relieves the programmer of this burden. This feature of the RSP software support is more adequately described in [18].

In general, there are two main problems with fractional fixed-point arithmetic. One is the overflow problem. Normally, with fractional fixed-point arithmetic, additions are performed modulo 2. The result of the addition of a and b then is $(a + b)$ modulo 2. This operation is error-free if the result falls within the range from 1.0 exclusive to -1.0 inclusive. However, an error occurs when one attempts to add two numbers whose sum equals or exceeds 1 or is less than -1, and the magnitude of that error is 2.0—a catastrophic error. The other problem is the underflow problem. This occurs when a datum has such small magnitude that insufficient bits of precision remain. The RSP has means for handling both of these problems in ways such that the programmer is not unduly burdened.

The RSP arithmetic unit has several features intended to deal with the overflow problem. One of these is the presence of guard bits in the Z and Y registers. The Z register, with 16 bits plus one guard bit, can represent numbers from -2 inclusive to $+2$ exclusive. Thus any two 16-bit numbers from the data store can be added in the Z register without fear of overflow. The Y register also has a single guard bit and can represent data in the same range, although with greater precision.

The RSP also features saturation-mode arithmetic, of both internal and external varieties. With internal saturation on, any sum in the Z or Y registers which would equal or exceed $+2$ or be less than -2 is saturated to $+1.9997$ or -2, respectively. With external saturation on, if the contents of the Y or Z registers equals or exceeds $+1$ or is less than -1, a data transfer to memory transfers the quantities 0.9997 or -1, respectively.

Let us now consider a simple example, to illustrate the benefit. Suppose we wish to calculate the output of the filter

$$y_n = h_1 x_{n-1} + h_0 x_n + h_{-1} x_{n+1}, \quad (3)$$

where $h_1 = h_{-1} = 0.50$, and $h_0 = 1.00$. Assuming that the value of n has been already loaded into index register X1, the code for computing this sum is

LZ	x−1(X1)	Load the contents of x_{n-1} into the Z reg
AZ	Z,x+1(X1)	Z equals Z plus the contents of x_{n+1}
HPZY	1	Y equals the contents of Z shifted by 1 bit
LZ	x(X1)	Load the contents of x_n into Z
HAZY	0	Y equals Y plus the contents of Z shifted 0 bits
STY	y(X1)	Store the result in y_n

Let us now assume that $x_{n-1} = 0.5$, $x_n = 0.625$, and $x_{n+1} = 0.5$. The intended result is then 1.125. If the RSP had neither saturation arithmetic nor guard bits, the result of the addition x_{n-1} plus x_{n+1}, which is computed in the Z register, would be 1.0 mod 2, $= -1.0$. Then the sum computed in the Y register would be 0.125, which is assigned to y_n. However, with the guard bit in the Z register, the sum computed in the Z register is 1.0, and the result computed in Y is 1.125. With external saturation on, the result assigned to y_n is 0.99997. As the example illustrates, these features do not entirely prevent errors, but they do make them less severe.

The arithmetic left shifter of the RSP helps the application programmer deal with the underflow problem by permitting the programmer to upscale a number, while in the saturation mode. Thus, upscaling can be performed without fear of the catastrophic errors that upscaling might otherwise create, and there is no need to test each number individually to determine when an overflow might occur. Such an individual test would be both tedious to the programmer and wasteful of processing time. This instruction requires $b + 1$ cycles of throughput for b bits of upscaling.

The RSP arithmetic unit also has many other features which simplify programming. One such feature is the double-word width (32 bits) of the Y register. Since the Y register is a functional accumulator, it is desirable to have more than 16 bits, so that arithmetic truncation errors are reduced. The double-word length is more than sufficient for that. Just as importantly, the double-word length also permits a convenient mechanism for double-precision arithmetic. Instructions that operate selectively on the upper and lower bits of the Y register facilitate the programming of double-precision arithmetic.

The RSP also has a DIVIDE instruction which requires 17 cycles of throughput. This also operates in a saturating fashion. The MULTIPLY, DIVIDE, and arithmetic-left-shift instructions are the only RSP instructions which require more than one cycle of throughput.

The RSP ALU also provides a variety of logical instructions. Although seldom used, they provide a great convenience to the programmer, when their function is required.

Instruction fetch and sequencing

A variety of instruction sequencing possibilities are provided by the RSP architecture in order to simplify programming. The stage-one unit of the RSP performs the RSP instruction sequencing, as well as the instruction fetch.

Normally, the processor accesses the instruction store sequentially. However, other sequencing may be executed in response to either the program or an external interrupt. Unconditional and conditional jump statements are provided. The conditional jumps may be conditioned on either

index conditions supplied by the stage-two hardware or ALU conditions supplied by the stage-four hardware.

The stage-one hardware also mechanizes subroutine branching. When a branch-to-subroutine (BS) instruction is executed, the current address plus 1 is pushed onto the subroutine stack. When the associated return-from-subroutine is executed, this address is popped from the stack and becomes the address of the next instruction fetched. The subroutine stack is 64 words deep, a depth that prevents stack overflow, and is maintained in the data store. The mechanization of subroutine stacking is a valuable aid to the programmer.

PROCEED sequencing is used so that the RSP can conveniently access coefficient subroutines in a sequential manner. The first coefficient subroutine is accessed through a branch-to-subroutine instruction. When the subroutine terminates with a return statement, the address following the return is loaded into the SAVE register. Let us consider how this aids our prior example, the calculation of (3). The code for our convolution can also be written

```
    LZ     x-1(X1)      Load x_{n-1} into the Z reg
    AZ     Z,x+1(X1)    Z equals Z plus x_{n+1}
    BS     H0           Jump to coefficient sub-
                        routine at location H0
    LZP    x(X1)        Load x_n into Z reg, pro-
                        ceed to next coefficient
    STY    y(X1)        Store the result in y_n

H0  HPZR   1            Y equals Z shifted 1 bit,
                        return
    HAZR   0            Y equals Y plus Z shifted
                        0 bits, return
```

This code gives the same result as the code given earlier for the same example, and one might wonder what its advantages are. In a complete signal processing program, perhaps one-half of the instructions would be shift/add type instructions associated with coefficients. Shift and add instructions in the RSP require only 8 bits of storage, since the other 16 bits of the instruction are always 0s. Thus, writing the code, as just discussed, allows one to populate a part of the RSP's instruction store with only 8 bits of memory. This section of the memory, which we call the coefficient store, is of arbitrary length and placement in the instruction store. The savings in memory due to using this technique can be considerable, as discussed in [19].

The RSP may also alter its instruction sequencing in response to an interrupt. An RSP interrupt is executed much like a jump-to-subroutine. In this case, the RSP jumps to the address provided by the interrupt logic, and a return address is pushed onto the stack. After the interrupt processing has completed, the state of the RSP internal registers is restored (as part of the interrupt subroutine), and a return statement is executed to return the RSP to its prior processing.

Data-store address-generation unit

Data-store address-generation is performed by the stage-two hardware. Four different modes of addressing are permitted by the RSP instruction set. They are (1) direct addressing, (2) offset addressing, (3) indexed addressing, and (4) masked addressing. Let us denote the operand of an RSP instruction that involves a memory access by a, and the address of memory to be accessed as the *effective address*. In the various addressing modes, the *effective address* is computed as

a	Direct, indicated in the program by (N),
$a + B$	Offset, indicated in the program by (O),
a + Xi	Indexed, indicated in the instruction by (Xi), and
$B + (a$ + Xi)&M + a& M	Masked, indicated in the instruction by (Mi),

where B is the contents of the BASE register, X1 and X2 are the two index registers, M is the contents of the MASK register, M is the complement of the contents of the MASK register, and & is the logical and. The masked mode is extremely useful in implementing the circular buffering associated with FIR filtering. Normally, when using this addressing mode, the BASE register is set to zero, the upper bits of the MASK register are set to 0, and the lower bits are set to 1. Then, as can be seen by examining the equation, the lower bits of the effective address are the lower bits of a + Xi, and the upper bits are the upper bits of a.

Let us apply this technique to the calculation (3), our prior example. With x on a four-word boundary, and the MASK register set to 3, the code for our FIR filtering becomes

```
    LZ     INPUT(N)     Load the contents of IN-
                        PUT into Z
    STZ    x+2(M1)      Store Z into x_{n+1}
    AZ     Z,x+0(M1)    Z equals Z plus x_{n-1}
    BS     H0           Jump to coefficient subrou-
                        tine at location H0
    LPZ    x+1(M1)      Load x_n into Z and proceed
                        to next coefficient
    STY    OUTPUT(N)    Store the contents of Y in
                        OUTPUT
    AX1    X1,1         Increment index reg X1 by
                        1
H0  HPZR   1            Y equals Z shifted 1 bit, re-
                        turn
    HAZR   0            Y equals Y plus Z shifted 0
                        bits, return
```

It is easily seen that the upper bits of all of the operands, x+2, x+1, and x+0, equal x, since x is on a four-word boundary; and the respective effective addresses then are x+((j+X1))4, where $((n))m$ is n modulo m. A careful examination of the addresses generated reveals that circular buffering indeed takes place. To accept the next input point and compute the next output point, we need only increment X1 by 1, as is done in the AX1 instruction, and re-execute the same code. We note that a data buffer of only four words, for the four prior values of x_n, is now sufficient for these calculations, and no buffer overflow checking is required. The code given in the preceding section would have required buffer overflow checking, a nuisance and a hazard to the programmer.

In addition to two index registers, the RSP address generation unit also has two work registers. Either work register may be used to increment either index register, and either work register may be compared against either index register for index control.

The presence of two index registers and two work registers in the RSP is extremely useful. The two index registers provide convenient means to mechanize the double-loop indexing that is commonly required for FFT, WFT, and correlation calculations. With the two work registers, "DO LOOPS" of the form DO I = J TO K BY L are conveniently implemented.

The RSP address-generation unit provides the RSP with a powerful addressing capability, but it also spends a sizable part of the RSP hardware budget. It was felt that powerful addressing capability was necessary, however, so that the RSP would truly be easy to program. It also provides an architecture that has the ability to improve performance with increased technology without substantially changing the instruction set. A technology upgrade that enhanced indexing capabilities would require major changes to the instruction set.

Data-store access

Data-store interfacing to the RSP is handled by the stage-three unit. The effective address is latched into the Data-Store Address Register (DSAR) by the address-generation unit. One data-store access per instruction is permitted; however, instructions may instead have immediate data, that is, data that form the operand of the instruction.

Data to or from the I/O unit interface the data store through the stage-three unit. This is done on a cycle steal basis. The stage-one, -two and -four units of the RSP are delayed for one cycle, while the stage-three unit makes the appropriate transfer. With this operation, only one cycle of throughput is lost for each word transferred, and the operation of the entire pipeline appears merely retarded by one cycle.

I/O and control unit

As was discussed in the introduction, the computational requirements of signal processing applications vary greatly. Although many applications can be satisfied by single-processor implementations, many cannot. In order to provide the versatility to cover a broad application range, it was felt that the RSP must have the ability to be easily connected into systems.

An examination of many applications was undertaken, and it was found that for most applications, one of the three system architectures, listed below, was adequate.

1. Single-processor systems.
2. Systems configured according to the application data flow. These systems are configured in the serial/parallel manner in which the applications decompose.
3. Systems configured around a central bulk storage. These systems, due to mode changes, need flexibility beyond that provided by a simple data flow clustering. Configuring around a central storage for the cluster provides this flexibility.

This collection of system architectures lacks the generality of distributed computing architectures, in general, but is adequate for most signal processing applications. An I/O architecture that would support all of these systems was defined. It was decided to implement it off-chip to conserve RSP chip resources. This architecture is called the SPIO. (Other I/O architectures could instead be connected to the RSP, when advantageous.)

Perhaps the greatest I/O requirements derive from some of the more sophisticated distributed systems. The ability to IPL and to re-IPL with mode changes is often required. A means of system synchronization is needed. Means for performing preventive maintenance and fault location are also required. It is also often necessary that block-data transfers be executed with minimum impact on the performance of the processor. For the RSP to be sufficiently versatile to implement distributed systems, it must contain features that allow these requirements to be satisfied.

In an RSP/SPIO combination, the RSP performs the processing and the SPIO manages the flow of data and control into and out of the RSP. Each SPIO provides one control port and one data port. A fully configured RSP can support two SPIOs (with one of the control ports disabled). The control port is used for IPL, system synchronization, preventive maintenance, and fault location. It also provides a port through which to debug operation of the RSP without

interrupting the normal progress of data transactions. The data port is used to mechanize the data transfers occurring during normal operation of the system.

All communication with the RSP occurs across the RSP's parallel I/O interface. This interface has parallel address, data, control, scan, and interrupt lines. The RSP can be notified of a pending data transfer or of a system reconfiguration through its interrupts. The parallel interface then permits a control port or data port to read data from or write data into the RSP on a cycle-steal basis. It also enables an external controller to read or write instruction store, through the control port, when the processor is in the stop mode. Through the control port, a system controller can also read the status registers of the RSP or change the state of the processor to run, reset, or start. While in the stopped state the scan lines can be used to scan out the internal state of the RSP.

Other debugging functions are also supported. The RSP has a real-time clock, which can be used to generate an interrupt at some specified time. This permits the RSP to "time out" if a given processing task takes unexpectedly long, a convenient preventive maintenance feature.

Although the RSP I/O unit does not provide the RSP with the versatile stand-alone function desired, it does permit the RSP to attach to an external I/O which can provide the function desired. The parallel organization of the I/O interface also permits sufficient throughput so that the RSP can support the data-transfer requirements of most applications without being I/O bound.

RSP performance

A 15 000-gate version of the RSP has been developed by the IBM Federal Systems Division in Manassas, Virginia [20]. This 7.6-mm (300-mil)-square chip uses 2-μm NMOS polysilicon-gate technology and provides 171 off-chip pads. It has a 200-ns cycle time and requires approximately 2.5 watts of power.

It is difficult to quantify the processing power of a signal processor with any one statistic, such as cycle time. Perhaps the best measure is its ability to perform typical signal processing algorithms and applications. A few are given herein to demonstrate the power of the RSP.

A 1008-point complex Fourier Transform, based on a Winograd algorithm, has been programmed on the RSP. This calculation takes 200 915 cycles, or 40 ms of computing time, to complete.

A 9600-bps modem, with 5 bits per baud (1920 baud) and with quadrature amplitude modulation (QAM), has been sized on the RSP. With a 32-tap complex adaptive equalizer, adapted every third baud, the modem is estimated to require 94% of one RSP's real-time computational power.

Voice-Excited Predictive Coding (VEPC) [21] is another technique used for speech compression. An 8500-bps coder and decoder, using this technique, were sized for the RSP. To implement both the coder and decoder, about 90% of the processing power of one RSP is required.

Other signal processing microprocessors

There are a number of other signal processing microprocessors that have been announced and described in the literature. A good survey of these is provided in [22] which is current to its June 1981 publication date. The features of four of these chips, well known at that time, are tabulated. Two other signal processing microprocessors were more recently announced [23, 24].

These six chips differ in a variety of ways. However, they all rely on on-chip memory for storage. Most of them have ROM instruction stores, and their programs must be added to the chip masks. The largest instruction store provided by any of these chips is 1536 words. One has a 194-word EPROM instruction store. All six have RAM data stores, with the maximum provided being 192 words.

They also have very different I/O capabilities. One of them has on-chip A/D and D/A converters, while the others rely on serial and parallel digital ports for input and output data. The widths of the parallel ports range from four bits to sixteen bits.

These signal processors also differ greatly in their multiplication capabilities. Some mechanize multiplications as shifts/adds, some accumulate partial products, and others have parallel multipliers. Their precision of multiplication ranges from 12×12 bits with 16 bits of precision to 16×24 bits with 40 bits of precision; and their multiplication times range from 400 ns per shift/add instruction to 200 ns for a complete multiplication.

They also differ in the programming conveniences provided. One of the chips has neither indexed addressing nor subroutine capabilities, while the others support both. However, only two have subroutine stacks more than one level deep.

Although the RSP does not have the most powerful multiplication rate, it currently has the most powerful I/O interface, and it is not as limited by storage. These qualities make it especially suitable in distributed system applications. Also, we believe that the presence of the powerful indexing, saturation arithmetic, subroutine capabilities, arithmetic

left-shift, double-word-width accumulator, and DIVIDE instruction make it the easiest to program.

References

1. *Applications of Digital Signal Processing,* A. V. Oppenheim, Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.
2. R. Zelinski and P. Noll, "Adaptive Transform Code of Speech Signals," *IEEE Trans. Acoust., Speech, Signal Processing* **ASSP-25,** 299–309 (1977).
3. J. M. Tribolet and R. E. Crochiere, "A Modified Adaptive Transform Coding Scheme with Post-Processing Enhancement," *Proceedings, 1980 International Conference on Acoustics, Speech, and Signal Processing,* Denver, CO, April 1980, pp. 336–339.
4. R. V. Cox and R. E. Crochiere, "Real-Time Simulation of Adaptive Transform Coding," *IEEE Trans. Acoust., Speech, Signal Processing* **ASSP-29,** 147–154 (1981).
5. G. D. Bergman, "A Fast Fourier Transform Algorithm for Real-Valued Series," *Commun. ACM* **11,** 703–710 (1978).
6. F. Mintzer, "Parallel and Cascade Microprocessor Implementations for Digital Signal Processing," *IEEE Trans. Acoust., Speech, Signal Processing* **ASSP-29,** 1018–1027 (1981).
7. A. Peled and B. Liu, *Digital Signal Processing: Theory, Design and Implementation,* John Wiley & Sons, Inc., New York, 1976.
8. K. Steiglitz, *An Introduction to Discrete Systems,* John Wiley & Sons, Inc., New York, 1974.
9. A. V. Oppenheim and R. W. Schaefer, *Digital Signal Processing,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.
10. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.
11. L. B. Jackson, J. F. Kaiser, and H. S. McDonald, "An Approach to the Implementation of Digital Filters," *IEEE Trans. Audio, Electroacoust.* **AU-16,** 413–421 (1968).
12. A. Peled and B. Liu, "New Hardware Realization of Digital Filters," *IEEE Trans. Acoust., Speech, Signal Processing* **ASSP-22,** 456–462 (1974).
13. G. D. Bergland, "Fast Fourier Transform Hardware Implementations—an Overview," *IEEE Trans. Audio, Electroacoust.* **AU-17,** 104–108 (1969).
14. B. Liu and A. Peled, "A New Hardware Realization of High-Speed Fast Fourier Transformers," *IEEE Trans. Acoust., Speech, Signal Processing* **ASSP-23,** 543–547 (1975).
15. S. Winograd, "On Computing the Discrete Fourier Transform," *Proc. Nat. Acad. Sci. U.S.* **73,** 1005–1006 (1976).
16. R. C. Agarwal and J. W. Cooley, "New Algorithms for Digital Convolutions," *IEEE Trans. Acoust., Speech, Signal Processing* **ASSP-25,** 392–410 (1977).
17. James W. Cooley, "Rectangular Transforms for Digital Convolution on the Research Signal Processor," *IBM J. Res. Develop.* **26,** 424–430 (1982, this issue).
18. Ken Davies and Fred Ris, "Real-Time Signal Processor Software Support," *IBM J. Res. Develop.* **26,** 431–439 (1982, this issue).
19. A. Peled, "On The Hardware Implementation of Digital Signal Processors," *IEEE Trans. Acoust., Speech, Signal Processing* **ASSP-24,** 76–86 (1976).
20. IBM Federal Systems Division press release, Bethesda, MD, February 1981.
21. D. Estaban, C. Galand, D. Mauduit, and J. Menez, "9.6/7.2 kbps Voice Excited Predictive Coder (VEPC)," *Proceedings, 1978 International Conference on Acoustics, Speech, and Signal Processing,* Tulsa, OK, April 1978, pp. 307–311.
22. R. H. Cushman, "Signal-Processing Design Awaits Digital Takeover," *EDN* **26,** 119–128 (June 24, 1981).
23. T. Nakamura, M. Yoshida, T. Uno, Y. Ichikawa, M. Mizutani, and K. Sawada, "A Digital Signal Processing LSI," *IEEE International Solid-State Circuits Conference, Digest of Technical Papers,* San Francisco, CA, February 1982, pp. 30–31.
24. S. S. Magar, E. R. Caudel, and A. W. Leigh, "A Microcomputer with Digital Signal Processing Capability," *IEEE International Solid-State Circuits Conference, Digest of Technical Papers,* San Francisco, CA, February 1982, pp. 32–33.

Received August 19, 1981; revised January 26, 1982

Fred Mintzer is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. Abraham Peled is located at the IBM Research Division laboratory, 5600 Cottle Road, San Jose, California 95193.