Ken Davies
Fred Ris

Real-Time Signal Processor Software Support

The Real-Time Signal Processor (RSP) is a microprocessor optimized to provide fast, cost-efficient processing for signal processing applications. In order for the RSP to become fully useful, a complete set of software support tools needed to be developed. The hardware design and software development, which took place between 1978 and 1980, resulted in many architectural features which minimized hardware complexity at the expense of programmability. This paper describes the tools that were developed and the decisions that were involved, and includes hindsight comments on what was done. Particular emphasis is placed on the most interesting aspects of the software development, i.e., how the special architectural features of the RSP were handled to make the overall hardware/software system more programmable.

Introduction

The architecture for the RSP was developed from a few key concepts concerning common functions involved in signal processing [1]. Concurrent with the development of these ideas was Winograd's work on reducing the arithmetic complexity of the Discrete Fourier Transform (DFT) [2], and the work of Agarwal and Cooley on digital convolutions [3]. The architecture and organization of this processor and its subsequent evolution are described in the paper by Mintzer and Peled in this issue [4].

It is important to note, at this point, that the hardware design for the chip version of the RSP was begun in 1978, with a view to fabricating production-level chips two to three years later. This resulted in a design which would require approximately 10 000 equivalent gates of logic (30 000 transistors). The aim was to satisfy the computing requirements of a wide spectrum of signal processing applications. In order to achieve this, a 16-bit fixed-point processor was designed with the capability of performing extended-precision computations without undue stress. In view of the 10 000-gate target, it was decided that a fast multiplier, a key element of most signal processors, would occupy too much chip area. The RSP was thus designed with a slow multiplier but with features which would take advantage of the work in reduced-complexity algorithms to achieve good performance. The advantages of the use of rectangular transforms for reduced computational complexity are discussed in the paper by Cooley in this issue [5].

There are a number of features, of both the architecture and its implementation, which, though they contribute to the cost-efficiency of the processor, cause programming difficulties. These include the following features:

- The addressing mechanisms of the RSP, although allowing great flexibility in accessing data from a signal processing viewpoint, do not allow multiple data areas/objects to be accessed concurrently. This causes a problem with parameter passing. Also, in order to take advantage of the circular addressing mechanism, data objects must be located on power-of-two boundaries.
- In order to make multiplication by constants as fast as possible, sequences of "shift-and-add/subtract" instructions must be constructed to perform the desired multiplication.
- A four-phase pipeline, with no interlocks, is operational in the RSP to achieve a much faster cycle time for the processor. This requires instruction sequences to be corrected for the effect of the pipeline, so that results are not used before they are computed.

- The implementation of the architecture was to use a 4K-word program (instruction) store and a 4K-word data store. This does not allow room for an operating system; thus, all programs are stand-alone and must have all operating system services performed before they are loaded into the processor and executed.

The underlying principle was to design a fast microprocessor whose hardware component contained the basic functions and whose software made the composite system easy to program *for signal processing*. We emphasize the latter because, once a processor exists, there is a temptation to use it for functions for which it was not intended. We should also stress, at this point, that *signal processing* consists of very standardized components, configured and parameterized in different ways for different applications. Thus, the RSP is a composite of hardware (single chip plus storage) and software which can be coupled to perform efficiently over a wide range of signal processing applications, as described in [4]. The software was, therefore, a key component of the RSP design.

Programming considerations

At the time the software support effort was initiated, the hardware prototypes were still under construction and it was clear that a considerable amount of software assistance would be required for testing and debugging the processors. As an additional requirement, when the processors became operational we would wish to code applications to verify and demonstrate the success or deficiencies of the processor design. These two requirements pointed to the need for developing an early programming capability with a low-level language for exercising the hardware and an application programming facility to develop real programs reasonably quickly.

At this point, it was clear that we should first develop an assembler and loader, and therefore a basic cross-assembler and loader were developed. What was not clear was whether we should then build a high-level language (HLL) compiler. There were a number of factors which influenced the decision on this:

- An important customer for the processor was the Defense Department, which was at that time trying to define and create as a standard the Ada programming language. Unfortunately, Ada was neither fully defined nor accepted. Also, Ada is a very large language with no subsets, by definition.
- It is not clear that any HLL exists that satisfactorily captures the primitives of signal processing in a manner that allows a compiler to generate good code for a machine suited to signal processing [6].

There were a number of other factors, such as lack of manpower and the fact that the applications programmers were already very familiar with assembly languages for signal processing. As a result of these factors, it was decided not to build a compiler, but instead to extend the assembly language so as to provide the functions of a HLL, and also to provide in a high-level form the functions that were only provided by the hardware in a very low-level manner.

Impact of RSP architecture on its software

The assembly language is, for the most part, a regular assembly language, having a one-to-one mapping from statement to machine instruction. It was modeled on the System/370 Basic Assembly Language. This was done partly for convenience, since the automatic assembler-generator, used to construct the assembler, was designed for System/370-style syntax, but mainly for familiarity to the programmers. The interesting aspects of the language are, however, those features that attempt to restore the facilities of a HLL and those features that circumvent the programming problems, previously mentioned, that are associated with the RSP.

- *Addressing mechanisms*

Although there are seven addressing modes in the RSP, there is no general "base plus index" mechanism which allows easy access to several data areas concurrently. There is only one base register, and for most instructions this can be used only in combination with one of the two index registers, and this base register is inconvenient to load. This causes a problem with passing parameters to subroutines, since indexing into a parameter array requires the base register to point to that array, whereas indexing into a work array requires that the base register point to the base of that also (or to some known offset from that base address). Thus, for array parameters, passing by address is not convenient and therefore one of two mechanisms is used instead. The parameter can either be passed by value, or its name can be known globally. The latter mechanism, although not esthetically pleasing, is far more efficient. Even scalar parameters are not well passed by address, since they then require one of the three addressing registers to be set to point to them before being accessed.

The HLL concepts of procedures, arguments, and parameters were carried over into the assembly language. A procedure is defined with a "PROC" statement, its parameters are identified by "INPUT," "INOUT," and "OUTPUT" statements, and the invoking procedure identifies it by the "XPROC" statement, which also names the arguments. The following example shows how these statements are used together to pass parameters by value from one procedure to another:

```
SUB     PROC
        INPUT     BUF
        INPUT     VAL
        OUTPUT    ANS
BUF     DATA      4
VAL     DATA      1
ANS     DATA      1
```

Another assembly language procedure can invoke this procedure as follows:

```
SUB     XPROC     B,V,A
*Move data into B and V
        BS        SUB     Branch and stack to SUB
*Answer is now in A
```

In this example, B and BUF are thus synonyms for the same data object. This mechanism is, of course, provided in a CALL macro:

```
CALL  SUB,((MYB(INPUT),(MYV(INPUT),
            (NEWV(OUTPUT)))
```

This moves the data from MYB into BUF, MYV into V, branches to SUB, and on return moves ANS into NEWV.

In fact, this parameter-definition mechanism also interfaces with a PL/I calling program. The PL/I-to-RSP interface moves PL/I arguments into the RSP parameters and, after execution, moves the OUTPUT parameters back to the corresponding PL/I arguments.

In order to share data objects among many procedures, the "XDATA" statement is an extension of the basic "DATA" statement. It reserves data storage, allows initialization, and makes the name known to all other procedures.

As another consequence of the addressing modes, not only are parameters inconvenient to handle via addresses but so also are any other data objects whose location is unknown at program-load time. In particular, dynamic storage for temporary use cannot be conveniently addressed if an operating system function is used to acquire the storage at execution time. Thus, for just this purpose another data statement was defined: "ADATA," which reserves storage in a shared pool of temporary storage, to be allocated at load time by the linkage loader. The implementation of this mechanism is described in greater detail in a subsequent section on execution-time support.

As a result of the circular addressing mechanism available on the RSP, it is necessary to ensure that data objects to be accessed in this way are located on an appropriate power-of-two boundary in data memory. Thus, if a 32-word buffer is used in this manner, it must be located on a 32-word boundary. A final extension of the basic "DATA" statement permits just this alignment. The "BDATA" statement reserves data storage, allows initialization, and also allows specification of the boundary on which the object is to lie.

- *Coefficient multiplication*

The RSP does not, in the present implementation, contain a fast multiplier. Instead, a 16-by-16-bit multiply yielding a 31-bit result is performed two bits at a time for eight cycles, using a modified Booth's algorithm [4]. Much of signal processing, however, involves multiplications by constants, the exceptions being adaptive processing and direct correlations. The RSP utilizes this fact by providing "shift-and-add"-type instructions to perform these multiplications in much less than the eight cycles required for the full multiply. The reduction can be seen in the following example. If we take a binary fractional number, *e.g.*, $A = 0.0001110$ (eight bits only, for convenience), we can represent this in canonical signed-digit format as

$$A = 0.001001\bar{0} = 0.0010000 - 0.0000010.$$

Now to multiply any other number B by A we need only compute

$$A \times B = B \times 2^{-3} - B \times 2^{-6},$$

i.e., B shifted right 3 places minus B shifted right 6 places. This can be coded as ["H" is a mnemonic letter for sHift ("S" for Subtract)]

```
LZ      B       Z = B
HY      Z,3     Y = B × 2^-3
HSY     Z,6     Y = Y − B × 2^-6,
```

i.e., $A \times B$ can be computed in only two cycles instead of four for an 8-bit number, or eight for a 16-bit number. As can be seen from this example, the technique can achieve significant savings in computation time. In fact, typical uses of multiplication by constants occur in "window" operations, in fixed filters, and in transforms. In all these cases, the sum of the constants, for one iteration of the algorithm, is typically one. Thus, many of the constants are close to zero and require very few "shift-and-. . ." operations to complete. As a result, the multiplication by a constant typically averages only two to four cycles. In order to allow easy use of this architectural feature of the RSP, the assembly language provides an "instruction" MCY (multiply by coefficient), which generates the appropriate sequence of machine instructions to perform the multiplication.

The RSP architecture also has an extension of the above technique; since only four bits of the instruction are actually used to indicate which "shift-and-add/subtract" is intended and another four to indicate the shift amount, it is only necessary to store eight bits of the instruction. For this

purpose, the RSP defines a section of instruction store to have only eight bits, the remaining bits being provided to satisfy the decoding process. This storage is used to hold the "shift-and-. . ." sequences and they are executed by a coroutine linkage. There is a coroutine branch to initiate a sequence; thereafter, the last "shift-and-. . ." instruction in the sequence for one constant also performs a coroutine branch back to the mainline sequence, which loads the next data value to be multiplied and also performs a coroutine branch back to the next shift sequence. Due to the operation of the pipeline, this coroutine branching does not cause any extra cycles to be taken over the basic operations being performed. Since the "coefficient storage" is only eight bits wide, compared to the regular instruction width of 24 bits, this is a valuable coding technique when used.

The assembly language provides statements which take lists of constants as arguments and generate the appropriate "shift-and-. . ." sequences, and designate these instructions to be loaded into the coefficient storage by the linkage loader. In this way, with a single statement, an entire vector of multiplication instructions is stored away ready for use. There are three types of coefficient statement:

COEF A(. . .)	All results are added to Y register;
COEF S(. . .)	First result loaded into Y register and subsequent results are added; and
COEF L(. . .)	All results are loaded into Y register.

• *Instruction format*

Apart from specifying the operation to be performed, the op-code for the machine instruction also contains the specification of the source and target registers for the instruction and the addressing mode of the "real" operand. The instruction length is 24 bits, 8 bits for the op-code and 16 bits for the operand. This format yields 244 different instructions, with an overload of information in the operation name. This is replaced in the assembly language by a two-operand format with the target register specified in the operation mnemonic, the source register and the "real" operand specified as the two operands, and the addressing mode specified as a "tag" on the "real" operand.

The addressing modes are

I = Immediate,
N = Nonindexed (absolute),
X1 = Indexed using Index Register 1,
X2 = Indexed using Index Register 2,
M1 = Masked (circular) using Index Register 1,
M2 = Masked (circular) using Index Register 2,
B = Offset + Base Register with no indexing.

The tag is optional, with an appropriate default being taken. Some examples are

AY	Z,29(I)	Y = Z+29
AZ	Z,INCR(N)	Add the value of the data item INCR, addressed with absolute addressing, to Register Z and put the result in Z.
MY	Z,INPUT(M1)	Multiply Register Z by the circularly addressed vector INPUT, indexed by X1, and put the result in Y.

• *Miscellaneous features*

For application development ease, a large number of arithmetic functions were provided in the assembly language and evaluated by the assembler. This enriched the language considerably, and was particularly useful in the area of coefficient definition, *e.g.*,

C1 COEF S(COS(PI×1/32)−SIN(PI×1/32), . . .)

which is part of the coefficient sequence used in the FFT.

The assembler generator that was used incorporates the OS/VS Assembler H macro processor [7]. Thus, this fairly powerful macro facility is available as an extension to the basic language.

• *Optimization for pipeline execution*

As previously mentioned, the RSP achieves increased performance through the use of a noninterlocking, pipelined execution unit [4]. This creates sequencing problems for the programmer which can in most cases be overcome by reordering the instructions to perform useful noninteracting instructions in the "dead space" while other instructions in the pipeline are completing. If the simple technique of inserting no-operation (NOP) instructions is used, about 20–25% execution overhead can be expected. This falls to less than 5% in almost all cases when reordering is performed, and in many cases to less than 1%.

In order to free the programmer from this error-prone activity, the assembler incorporates a reordering algorithm. The algorithm operates from a table listing the data-flow characteristics and dependencies for each instruction. It performs label-to-label data-flow analysis and optimizes the code by reordering the sequence to maintain the same data flow with the minimum number of pipeline dependencies still exposed. Even though it operates only label-to-label with worst-case assumptions at each end-point, the algorithm still reduces the NOP overhead to less than 5%.

Code scheduling in a compiler is a common practice, but in assembly languages it is rare and has an interesting side issue. There is a problem with presenting the assembled code to the programmer: In which sequence should it be listed, as reordered or as written? The compromise chosen was to leave the written sequence on the listing with any deviation from this highlighted in the "OBJECT TEXT" section. In Table 1 (taken from a real program listing), the "LOCATION" field lists the offset from the beginning of the program in decimal and hexadecimal, and is not important. The "OBJECT TEXT" lists the location of the "real" operand (D indicates data storage), the hexadecimal op-code, the operand in decimal and hexadecimal, and finally any difference between the instruction at that location and the one coded in the "SOURCE STATEMENT" field for that line. The "STMT" field is a statement number count only. The "SOURCE STATEMENT" records the line from the source program. As can be seen from the example, the instruction at statement 464 has been moved down to 466 and statements 465 and 466 have moved up. The second reordering is similar.

Programming methodology

The approach chosen for programming the RSP was to develop as much as possible on the System/370, and only as the last step to actually run programs on the RSP hardware. This general approach is illustrated in Fig. 1. PL/I was chosen as the high-level development language because it is possible to define data types exactly matching those supported by the RSP architecture.

The hardware testing configuration is illustrated in Fig. 2. Most applications developed were one-dimensional, using speech, music, or telephone as the source of the signal. For those cases where the signal was originally digital, the IBM Series/1 minicomputer was used to provide data and to collect results for later analysis.

Tools

In order to support development using this methodology and configuration, a relatively complete set of software support tools were written. These were mainly for application development but also for testing and debugging the hardware. They initially consisted of the

Cross-assembler,
PL/I-to-RSP interface,
Linkage loader,
Simulators,
System/370-to-RSP interface, and
Debug package.

The cross-assembler executed on the System/370 and generated a System/370 object module which could be processed by the linkage loader and then simulated. It could also be

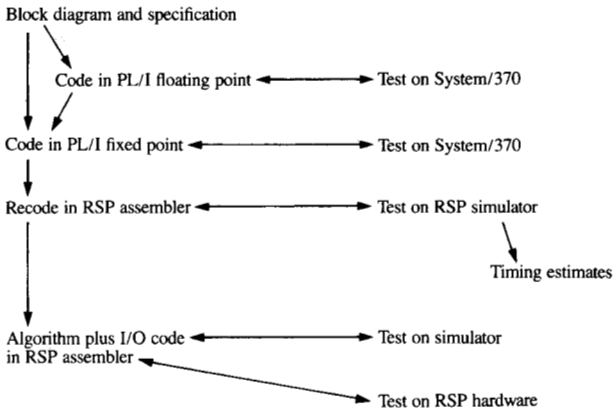

Figure 1 General approach used for programming the RSP.

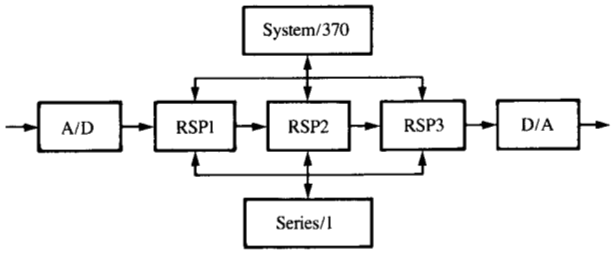

Figure 2 Configuration for hardware testing.

Table 1 Sample assembler listing illustrating code reordering.

Location	*Object text*	*STMT*	*Source statement*	
56/0038	D 75 2056/0808	463	LV	W1HALF
57/0039	D 5D 2056/0808 LZ	464	LR	W1(R),V
58/003A	B0 1/0001 HZ	465	LZ	W1HALF
59/003B	C0 16/0010 LR	466	HZ	Z,1
60/003C	D 41 2056/0808	467	STZ	W1HALF
61/003D	D 75 2056/0808	468	LV	W1HALF
62/003E	D 7D 2056/0808 LX2	468	LR	B(R),V
63/003F	D 25 2057/0809 AZ	470	LX2	W1HALF
64/0040	C0 8/0008 LR	471	AZ	Z,DATADDR
65/0041	D 41 2054/0806	472	STZ	UR
66/0042	D 75 2054/0806	473	LV	UR
67/0043	D 79 2057/0809 LX1	474	LR	W2(R),V

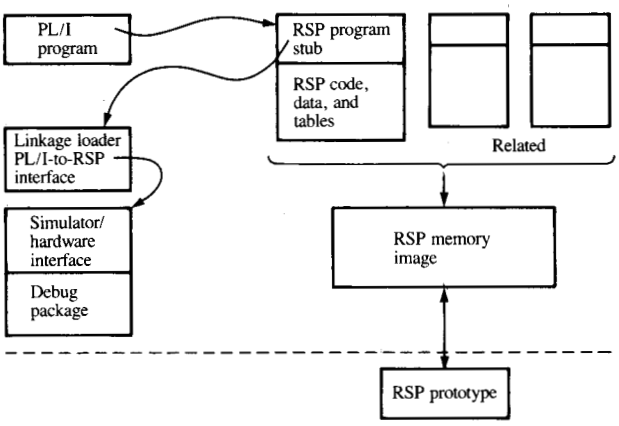

Figure 3 Operation flow for execution-time tools.

down-loaded into the RSP prototypes and executed there through the System/370-to-RSP interface. In either case, the debug package could be used to interrogate the state of the RSP (either simulated or real). The PL/I-to-RSP interface allowed the substitution of RSP assembler programs for PL/I subroutines, without the necessity of changing any of the PL/I programs. The design and implementation of these tools have several novel aspects; these are described in the following sections.

Execution-time support

Traditionally, simulators for machines under development have been stand-alone processors, interfacing with a user at a terminal. Commands are usually provided to load a program image file into the simulator's storage, to start simulation, and for a host of display and change facilities to examine and control the state of the simulated machine during and after the program execution. This same set of interfaces is often used to execute programs on the hardware model of the machine being developed. The approach taken for the RSP was entirely different. It was, rather, to make the RSP (simulator or hardware model) a black-box computational resource that was activated when necessary, with no detailed specification or commands from the user. Full traditional monitoring and debugging facilities were also provided. This approach was chosen to make application development as easy as possible, though it was less convenient for debugging the hardware. The most novel aspects of the execution-time tools are those which support this black-box concept and those facilities of the linkage loader which support the HLL features of the RSP assembly language.

The operation of the execution-time tools is illustrated in Fig. 3. The output of the assembler is an object module, suitable for loading into the System/370 storage by the linkage editor. The first portion of this is a System/370 program "stub," which receives control when the RSP program is invoked from a HLL. This small program invokes the RSP linkage loader, passing as parameters the several tables and data areas which comprise the rest of the assembled RSP program. One of these tables contains the information about parameters from the INPUT, INOUT, and OUTPUT statements. When the RSP program is loaded into the RSP storage image, the PL/I-to-RSP interface uses this table to move parameters between the invoking PL/I program and the RSP program, and after execution it moves the values in the INOUT and OUTPUT data objects back to the PL/I arguments.

Apart from the basic operations of any linkage loader, the RSP linkage loader also performs the processing to support the pseudo-dynamic storage mechanism in the assembly language. It does this by constructing a "call-tree" as it resolves the external references. (Note, this implies that recursive programs are rejected, though this is not a problem for signal processing applications.) For each RSP program, there is a table containing a list of the ADATA objects for that program. These tables are then used, together with the "call-tree," to determine at each invocation level the maximum ADATA requirements, *i.e.*, the program at each level which requires the most ADATA space. The maximum at each level is then reserved by the linkage loader and all programs at that level have their ADATA objects mapped into this area. Also, as the linkage loader is creating the "call-tree," the parameter table description is checked for agreement with the external reference table of the invoking RSP program. This catches gross interface errors very quickly.

After the linkage loader has processed all the necessary RSP programs and created load images for the instruction and data stores of the RSP, it invokes the simulator/hardware interface to execute the RSP program. This interface uses an external option to determine whether to download the memory images into the hardware and start execution there, or to invoke the simulator. When a "STOP" statement is executed, the simulator returns control to the interface. When the hardware option is used, the interface awaits a signal from the RSP processor indicating that a "STOP" has been executed and then up-loads the stores into the storage image. In either case, the PL/I-interface portion of the linkage loader then moves the values of any OUTPUT parameters to the PL/I argument variables and PL/I execution continues. Owing to this movement of parameter values back and forth from PL/I to the RSP program, the RSP program can be invoked in exactly the same manner as a functionally equivalent PL/I subroutine would be. Also, whether the RSP program is executed on the simulator or the hardware prototype is not reflected at all in the invoking PL/I program.

• *Simulator design*

Owing to the stage of hardware development at the time the simulator was designed, it was expected, correctly, that the simulator would be the main application development processor during the hardware implementation. Thus, it was decided to make the simulator operate as fast as possible, so as to simulate entire applications within a reasonable amount of time. In order to achieve this, all monitoring, debugging, and I/O functions were implemented outside the simulator kernel, which was written in a systems programming language. As a result, the simulator operates at approximately 50 KIPS on a System/370 Model 168. While this is 100 times slower than the real processor, it still allows most one-dimensional signal processing algorithms to be executed within a reasonable time. This is because most such algorithms are designed to run in 10 to 50 ms before repeating on fresh data.

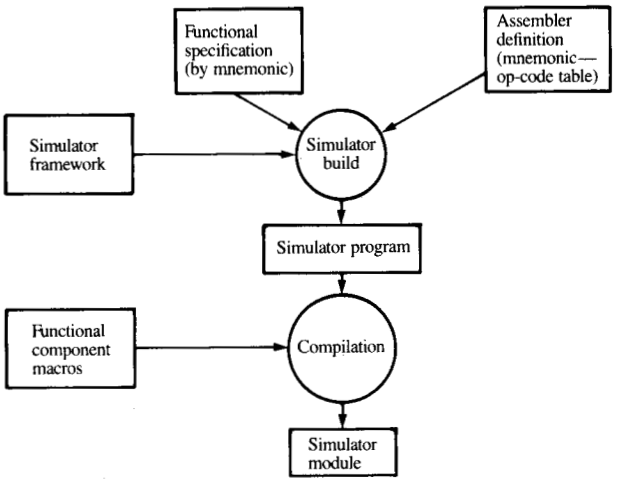

Figure 4 Illustrating method of construction of RSP stimulator.

An interesting aspect of the design of the simulator is the way in which it is constructed. The RSP is a four-phase pipelined machine with each phase performing a certain function. The simulator is built from two components: a framework and a multi-level macro definition of the instructions. The framework contains the support routines which perform the more complex functions such as ALU operation and address computation, and the housekeeping routines of the simulator. For the instruction definition, a macro-level interface was designed to allow easy specification of each phase, and each instruction was then defined as a sequence of four macros. So the simulator was *constructed by a program* which matched the instruction definitions already created with the op-code, using the mnemonic as the link, and which then inserted these definitions into the simulator framework before compilation. This allowed the simulator to be always in synchronism with the assembler, and permitted both to track changes in the RSP definition/implementation very quickly. This process is illustrated in Fig. 4.

Program generators

So far we have been concerned with the more novel aspects of the otherwise relatively standard software support tools for any new machine. We now look at a different approach, one that has been partially successful in the business world—that of program generators. As stressed in the introduction, signal processing is very standardized, with filters, transforms, and correlators forming the bulk of the computational load. Thus, with a processor specifically designed to solve these particular problems, and with the only programming language implemented being an assembly language, there is a good opportunity for program generators. For almost any analytical signal processing, the Fourier Transform constitutes most of the processing. The RSP has an architecture in which the Winograd Fourier Transform (WFT) is strongly favored. Unfortunately, general WFTs are difficult to program and so, initially, the first program generator that was constructed was a finite-impulse-response (FIR) band-pass-filter generator.

• *FIR filter generator*

In order to automate the construction of FIR filter programs, an interactive questionnaire was developed. (It should be noted that the RSP is very well suited to symmetric FIR filter programs, and can execute them with no cycles lost to pipeline hazards.) The questionnaire seeks a specification of the filter characteristics, such as sampling frequency, pass bands, stop bands, attenuation, etc. These data are collected, as illustrated in Table 2, and an annotated file is produced.

Having collected the specification, the Parks-McClellan-Rabiner [8] program is used to derive a symmetric FIR using floating-point arithmetic. This process is iterated upon, usually increasing the filter length, until a filter is found that, when implemented in the limited precision specified, still meets requirements over the entire range. For more details on the results obtained, see [9]. The impulse-response coefficients produced by the Parks-McClellan-Rabiner program are used to build a very standardized, but efficient, symmetric FIR filter program for the RSP. The resulting program is an RSP subroutine that accepts signal input data and outputs filtered data as RSP parameters (see previous description of this mechanism).

As an option with the filter generator, the program can be assembled and simulated, if input data are made available, or even executed on the hardware configuration. A digital mixer program is executed on one RSP to combine analog input with a wide range of other signals. The mixer program can generate white noise and most of the standard signal-

Table 2 Interactive questionnaire used in development of symmetric finite-impulse-response (FIR) filter generator program.

Computer question	*User response*
Type of filter?	B (for band-pass)
Sampling frequency?	10,000 (10-kHz sampling)
Precision?	12 (12 bits)
Specify filter shape:	B 0 1000 0 (band from 0 to 1 kHz is to be a stop band) B 2000 3000 1 (2–3 kHz is the pass band) B 4000 5000 0 (4–5 kHz is also a stop band)
Sampling ratio?	1 : 1 (no decimation/interpolation)
Filter length or 0?	0 (Let the computer work out an appropriate length)
Pass band ripple in dB?	0.2
Stop band attenuation in dB?	40
End of specification.	

generator waveforms, with any combination, at any frequency. These signals can be mixed with any analog input and the resulting waveform fed digitally to the filter program under test. The output from the filter is sent to a D/A converter and can be displayed on an oscilloscope, frequency analyzer, or distortion meter, to verify that it is performing as required. In all, this technique was very successful, and in a few minutes at a terminal, a digital filter satisfying most common requirements could be designed, built, and tested.

• *DFT generator*

In those cases where the power-of-two Fourier Transform is not a suitable transform size, the Winograd Fourier Transform is particularly attractive on the RSP. This is so because the dynamic multiply takes eight machine cycles as opposed to one cycle for almost all other operations, and the addressing modes of the RSP allow very irregular patterns with no penalty. The algebraic manipulations required to construct a general WFT are, however, sufficiently difficult to deter most programmers. The computer can be a tireless manipulator of algebra [10] and so all that is required for a WFT generator is a set of decomposition rules to bring large transforms down to reasonable size (*e.g.*, a 420-point becomes a 20×21-point), a base set of smaller transforms already coded, and an escape mechanism into an algebra manipulator for previously unencountered sizes. Agarwal has already suggested a set of decompositions for many transforms up to 2000 points [11]. A library of some of the primitive WFT components has been generated, and the algebra manipulation is well understood. Unfortunately, no package has been produced to automatically build a WFT of a specified size for the RSP.

Summary and conclusions

The overall approach to the software support of a processor so specialized, and therefore limited, as a signal processor was to use a host System/370 computer wherever possible. While obviously the RSP could never support a useful assembler/compiler, this approach is more evident in the lack of any bootstrap code to control loading and debugging. The cross-assembler itself illustrates the approach: HLL features wherever possible and extensive expression handling to produce highly optimized multiplication sequences from natural specifications. The execution-time support also illustrates this, with the RSP processor itself treated as a black box and integrated into PL/I. Finally, even the programming of some functions is automated on the host computer by use of the program generators.

This, then, was the strategy as it evolved; how well did it work?

• *Programming experience*

The first use of the programming tools was in the architectural evaluation of the processor before and during its construction. For this function, producing timing estimates of common kernels or entire applications, the tools proved to be very effective. Particularly useful also was the fast tracking of architectural alternatives. This was due to the multi-level macro construction and table-driven implementation.

The second use of the support tools was to load programs into the RSP models to test and debug them. This worked fairly well, though the engineers still preferred to load small instruction sequences "by hand," either for assurance of what was really in the machine or for faster turnaround. If a truly interactive debugging facility had been available, even via the host computer, it would have assisted this process.

After these two slightly unusual uses, the bulk of the programming was in building applications or subsystems, which was the envisioned role of the tools. In general the experience was good; most signal processing programmers had a strong hardware background and were comfortable with assembly-language programming. In fact, as extra HLL features were added, it was difficult to encourage their use at first. All programmers tried to hand-optimize one program for the pipeline; some even tried two programs, but all quickly found that the assembler did a very good job and produced correct code. Some programmers did not use the PL/I interface, except as required; others used it heavily, gradually converting a PL/I application to RSP code. All

programmers, however, developed their code on the simulator before trying it on the hardware models. Finally, despite its potential time-saving appeal, the filter generator received almost no real usage. The reason for this may have been that in the applications developed, the filters were not simple band-pass filters; usually extra shaping was required in the pass-band or transition region.

- *Missing functions and future support*

There are three major areas of missing support: hardware operation, HLL compiler, and WFT generator. Undoubtedly, the testing, debugging, and later development work on the hardware models would have been assisted considerably had there been support equivalent to that available in the microprogramming development systems currently in use with the popular microprocessors. This requires software to be developed on a mini- or microcomputer, at least the debugging facilities and hardware interface. Also, it would be more convenient if programs could be developed on the same computer. Building a HLL compiler, however, is certainly easier on a mainframe and is somewhat at odds with developing a microprogramming development system on a minicomputer. Already, however, some applications are developed entirely in PL/I and then translated into RSP code. Thus, we know that PL/I, at least, is not intolerable for coding signal processing applications. In fact, it has proved much more convenient than expected, and the choice not to build a compiler, may, in retrospect, have been wrong, though the choice of language is still as difficult as ever. For nondefense applications, it now seems clear that a FORTRAN, PL/I, or Pascal compiler would be very beneficial, though some programmers would not wish to leave the comfort of an assembly language.

The WFT generator is always useful, regardless of the programming language or development technique. This is particularly so for a machine with a multiply-to-add-time ratio like the RSP and with flexible addressing capability.

Beyond all these, however, is a yet higher-level application development facility that is possible for signal processing applications. As mentioned earlier, these applications almost always exist in the early stages of development as a block diagram of such functions as filters, transforms, band shifts, integrators, and correlators. Each of these functions takes a fixed amount of input data, performs a standardized function, and produces a fixed amount of output data, which is then passed to one or more of these functions for further processing, etc. This whole process can today be defined to the computer through interactive graphics. If to this we append information about input and output data and specify which standard function is to be performed in each box, entire applications can be *automatically* programmed. Thus, while the type of support tools described in this paper are still necessary primitives, the future of signal processing software support research lies in developing such sophisticated automatic programming systems.

Acknowledgments

The authors gratefully acknowledge the many people in IBM who have contributed much to the work reported here. Particular thanks are due to R. Riekert and M. Sachs for their good ideas early on, N. Brenner and A. Wadia for their hard work on the implementation, L. Nackmann for inventing and building extensive Series/1 support, S. Miller for the multi-level macro implementation, and A. Ruiz for finding most of the bugs. We also wish to thank the referees for their excellent suggestions.

References

1. A. Peled, "On the Hardware Implementation of Digital Signal Processors," *IEEE Trans. Acoust., Speech, Signal Processing* **ASSP-24,** 76–86 (1976).
2. S. Winograd, "On Computing the Discrete Fourier Transform," *Proc. Nat Acad. Sci. U.S.* **73,** 1005–1006 (1976).
3. R. C. Agarwal and J. W. Cooley, "New Algorithms for Digital Convolutions," *IEEE Trans. Acoust., Speech, Signal Processing* **ASSP-25,** 392–410 (1977).
4. Fred Mintzer and Abraham Peled, "A Microprocessor for Signal Processing, the RSP," *IBM J. Res. Develop.* **26,** 413–423 (1982, this issue).
5. James W. Cooley, "Rectangular Transforms for Digital Convolution on the Research Signal Processor," *IBM J. Res. Develop.* **26,** 424–430 (1982, this issue).
6. K. Davies, "Why Not a High Level Assembly Language," *Proceedings of the 1980 International Conference on Acoustics, Speech, and Signal Processing,* Denver, CO, April 1980, pp. 927–930.
7. *OS/VS-DOS/VS-VM/370 Assembler Language,* Order No. GC33-4010; also, *OS Assembler H Language,* Order No. GC26-3771; available through IBM branch offices.
8. J. H. McClellan, T. W. Parks, and L. R. Rabiner, "FIR Linear Phase Filter Design Program," in *Programs for Digital Signal Processing,* IEEE Press, New York, NY, pp. 5.1-1–5.1-13.
9. K. Davies, "Filter Design Using Finite Precision," *Research Report RC-7987,* IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1979.
10. J. H. Griesmer, R. D. Jenks, and D. Y. Y. Yun, "SCRATCHPAD User's Manual," *Research Report RA-70,* IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1975. See also J. H. Griesmer and R. D. Jenks, "Experience with an On-Line Symbolic Mathematics System," *Proceedings of the ONLINE 72 Conference,* Vol. 1, Brunel University, Uxbridge, Middlesex, England, September 4–7, 1972, pp. 457–476. Also available as "The SCRATCHPAD System," *Research Report RC-3925,* IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1972.
11. R. Agarwal, unpublished results, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

Received September 25, 1981; revised February 17, 1982

The authors are located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.