DESIGN OF A 300-BAUD FSK MODEM USING CUSTOMIZED DIGITAL SIGNAL PROCESSORS

by

W. L. Abbott

Memorandum No. UCB/ERL M84/93

3 August 1984

DESIGN OF A 300-BAUD FSK MODEM USING CUSTOMIZED DIGITAL SIGNAL PROCESSORS

by

W. L. Abbott

Memorandum No. UCB/ERL M84/93

3 August 1984

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

DESIGN OF A 300-BAUD FSK MODEM USING CUSTOMIZED DIGITAL SIGNAL PROCESSORS

William L. Abbott

University of California, Berkeley
Department of Electrical Engineering
and Computer Sciences

I. INTRODUCTION

There are several approaches that may be taken to the design and implementation of a single chip digital signal processing (DSP) system. One approach is to use a processor that can be programmed in a way similar to that of a microprocessor and that is specifically designed for DSP. While this type of processor may be adequate for implementing simple DSP functions, it becomes limited and inefficient when applied to the task of realizing complex single chip DSP systems. Other alternatives are to do a custom design or to use some kind of semicustom approach. An all-custom design takes relatively too long to complete and is not financially worthwhile unless the finished chip is one that will be produced in large quantities. The semicustom method allows a designer to more specifically tailor the processors to the desired application while avoiding an entire chip design.

The 300-baud frequency shift keyed (FSK) modem described in this paper utilizes parallel digital signal processors which are configured according to a set of macrocell descriptions. The computer-aided design (CAD) system used to specify the modem will be referred to as the DSP integrated circuit (IC) Layout Generator and will be described more completely in the following chapter.

One of the purposes of this research project was to demonstrate the feasibility of using the DSP IC Layout Generator to design and implement monolithic DSP systems. Since the modem implementation requires a variety of operations such as addition, multiplication, division, absolute value, delay, multiplexing, and conditional, it is a good example for testing the versatility of the DSP IC Layout Generator. Although a 300-baud FSK modem could be implemented using one of the generalized digital signal processors mentioned previously, a more complex and higher speed modem could not be realized in this manner. It is in this more complex case that the DSP IC Layout Generator has a great advantage over the generalized digital signal processors.

The modem discussed in this paper includes the functions of modulation, demodulation, and filtering in two parallel processors on a single chip. The chip layout for the processors and their associated control lines and input/output lines was generated automatically using the DSP IC Layout Generator.

Research sponsored by the Defense Advanced Research Projects Agency under contract no. N00039-84-C-0507.

II. THE DSP IC LAYOUT GENERATOR

The DSP IC Layout Generator provides for the automated macrocell design of customized DSP systems [1]. To develop a DSP chip using the Layout Generator, the designer must specify a design file (the design file for the 300-baud modem is shown in Appendix A). The design file consists of a special purpose language that describes which macrocells are needed and how they are to be configured. An emulator allows the designer to test and debug his design file. The Layout Generator produces a chip layout given the design file and information from the emulator.

The basic architecture of a DSP system designed with the DSP IC Layout Generator includes parallel processors, serial buffered data connections for interprocessor communication, a parallel signal I/O bus, and a buffered host I/O section (see Figure 1). An understanding of the processor structure is enough to allow a basic understanding of the modem implementation. Therefore, only the processor structure will be described in more detail here.

II-1. Processor Structure

Each processor is made up of a number of macrocells, with the particular configuration being dependent on the design file description.

The data memory macrocell consists of up to 128 RAM locations and is used for storage of variables and constants. It may be addressed in an absolute mode or in an indexed mode.

The arithmetic unit (see Figure 2) is a pipelined structure and has a wordlength specified by the designer. Data is represented in two's complement notation. The main functional blocks in Figure 2 are structured as follows:

1. Mor Register: The mor register is loaded with the contents of a RAM word in a (r)ead operation and with ∿ mir (∿= bit inversion) in a (w)rite operation.

2. Barrel Shifter: The barrel shifter can perform a right shift of 0 to 6 bits or a left shift of 0 to 1 bit. Either the mor or sor register may be used as input. Using the sor as input makes it possible to perform a shift of more than 6 bits.

3. Complementer: The complementer outputs the true value, the inverse value, or the absolute value of the sor register.

4. Adder: Since the adder saturates, a positive or negative overflow is represented by the maximum or minimum value. Adder output is accumulator input.

5. Mir Register and mbus: The mir register is loaded from the mbus and is a transparent latch. The mbus may be driven by the accumulator (default), the mor register, or the I/O unit.

A connection to the parallel I/O bus (for only one of the processors), the serial interconnections with the other processors, and the host I/O unit make up the processor I/O unit. The serial I/O units include serial-to-parallel (s/p) or parallel-to-serial (p/s) converters and buffering so that the designer does not have to worry about the timing of the variable transfers.

The control sequencer macrocell consists mainly of two cycle counters and the microcode instruction ROM. This macrocell is kept simple by allowing no branching and only a restricted form of looping.

To handle the necessity of having conditional operations, a processor may also include a finite state machine unit. The finite state machine provides a program set condition code bit which controls a conditional write instruction.

The address arithmetic unit supports either direct addressing or indexed addressing of the data memory unit. Indexed addressing is accom-

plished using index registers ix and iy and makes it possible to execute a number of iterations of a subprogram or the main program to implement specialized addressing schemes such as table look-up.

II-2. Design File

As previously mentioned, a designer can completely specify a DSP IC by writing out a design file. The syntax of this file and the instructions that may be used are thoroughly discussed in part 2 of Reference [1]. The designer first specifies global variables to be used for signal and host I/O and interprocessor communication and then specifies the processors. A majority of the effort in writing the design file goes into these processor specifications. Variables (locals) and constants are defined to make up the data memory. A finite state machine is defined if the designer needs conditional operations. Finally, a main program and optional subprogram are written. The programs contain the instructions to be executed each sample cycle and perform the DSP mathematical, data storage, and data transfer operations. The desired mathematical and storage operations are converted into the special microcode instructions, most of them being related to the processor arithmetic unit.

Two's complement multiplications of two variables are performed using a parallel-serial method in which a sequence of partial products is generated in a bit-parallel, word-serial format and accumulated by the single accumulator. An example of this method is as follows:

To multiply the two's complement numbers x and y ($-1 \leq y < 1$),

Represent y as $y = -y_0 + \sum_{i=1}^{n-1} \frac{y_i}{2^i}$ where y_i is the ith bit of the two's complement representation.

Then $x*y = -x*y_0 + \sum_{i=1}^{n-1} \frac{x}{2^i} * y_i$

Thus, start with zero when y is positive and -x when y is negative and add x, shifted by i positions, if the ith bit of y is a one. The bits of y are needed serially on successive cycles, MSB first, in order to control the addition of the shifted values of x.

Example:

x = 011000 (3/4)

y = 110100 (-3/8)

x*y = 101000 (= -x)
001100 (x/2)
000000
000011 (x/8)
111111 = (-9/32)

Multiplication of a variable with a constant is much easier and is required for digital filtering applications. The constant coefficient is represented in the canonical signed digit form:

$$c = \sum_{i=0}^{j} x_i * 2^{n_i}$$

where (c = constant)
(x_i = -1 or +1)
(n_i and j chosen to minimize the total number of digits)

Thus, a multiplication such as c*y, where c = 0110111 (55/64) can be achieved by implementing $(2^0-2^{-3}-2^{-6})*y$.

Division of two variables is necessary for such operations as signal normalization. Divisions are accomplished as follows:

To perform N/D (|D|>|N|)

1. Determine the sign of the result.
2. Load |N| in the accumulator.
3. Subtract |D|/2, |D|/4, etc., from the accumulator on successive cycles.
4. Accumulate the results of these subtractions only when they are positive, and place a 1 in the appropriate bit position of the quotient.
5. The quotient is in sign magnitude form and must be converted to two's complement form.

Example: N = 001 (1/4) D = 0101 (5/8)

```
First cycle:  001000      quot = 0
              110110 =    -|D|/2
              ------      (do not accumulate since <0)
              111110      quot = 00

Second Cycle: 001000
              111011 =    -|D|/4
              ------      (positive, so place 1 in quot)
              000011      quot = 001

Third Cycle:  0000110
              1111011 =   -|D|/8
              -------     quot = 0011
              0000001
etc.
```

III. OVERALL MODEM IMPLEMENTATION

III-1. Modem Functions Included

A full-duplex 300-baud FSK modem is made up of five main functional blocks. These blocks are the modulator, demodulator, transmit and receive filters, timing and control logic, and line driver and hybrid. All of the above functional blocks are implemented in the modem discussed in this paper except for some timing logic and the line driver and hybrid circuits (see Figure 3).

The full-duplex feature of the modem means that it can simultaneously transmit and receive data. This feature is accomplished by using different frequency bands for transmitting and receiving and by using the hybrid to put both sets of data on the same telephone wire pair. The different frequency bands of the modem are shown in Table 1 along with the corresponding mode of operation (originate or answer).

Data	Originate Mode		Answer Mode	
	Transmit	Receive	Transmit	Receive
0 = space	1070 Hz	2025 Hz	2025 Hz	1070 Hz
1 = mark	1270 Hz	2225 Hz	2225 Hz	1270 Hz

Table 1. Full-Duplex 300-Baud Modem Tone Allocation

The inputs and outputs of the modem lowband and highband filters are switched according to the desired mode of operation (controlled by $O/\overline{A}$ input). The filters are necessary because the hybrid combines the transmit

and receive signals onto the same wire pair resulting in some interference between the two frequency bands. The receive filter removes the signal energy in the adjacent transmission band so that the demodulator may more clearly detect data in the receive band. The transmit filter is necessary to remove sidebands caused by the FSK modulation of the data. This filter does not have to have as high an attenuation in the adjacent band as does the receive filter; however, since both a lowband and a highband receive filter are necessary, it is no additional design effort to use these filters on the transmitted signal.

III-2. Modem Design File

The modem design file listings shown in Appendices A and B implement the functions shown within the dotted line in Figure 3. Two processors are used in the design: one for the modulator and demodulator, and one for the lowband and highband filters. Processor "filters" is the processor that is attached to the parallel signal I/O bus. The requirement on the width of this bus is 12 bits and was set by demodulator considerations discussed later. Two input words and two output words use this bus each sample period (see Table 2).

Processor "filters" has a 20-bit word length in order to achieve the computational accuracy required to meet the filter specifications. "Filters" performs the multiplexing functions required to allow the modem to operate in either originate or answer mode and either full-duplex or self-test mode. The lowband filter is used in the originate self-test mode, and the highband filter is used in the answer self-test mode. Of course, both filters are used when the modem is in full-duplex mode, and their inputs and outputs are determined according to Table 1.

Input words: wordin:

O/$\overline{A}$	TXD	SQT	ALB

O/$\overline{A}$ = Mode pin (1 = originate, 0 = answer)
TXD = Digital data to be transmitted (1 = mark, 0 = space)
SQT = Squelch (modulator output = 0 if SQT is set (=1))
ALB = Self-test mode (if ALB = 1, then output of modulator after transmit filter is fed directly to demodulator) (ALB = 0 is full-duplex mode)

rxin: 12-bit digital word from A/D of pin RXA. FSK signal to be filtered and demodulated.

Output Words: wordout:

RXD	$\overline{CD}$	- - -

RXD = demodulated digital data (1 = mark, 0 = space)
CD = carrier detect signal (0 = carrier present, 1 = carrier absent)

txout: 12-bit digital word for input to D/A and transmission on pin TXA. FSK modulated and filtered data.

Table 2. Modem Data Words

Processor "modem" receives the "wordin" and filtered "rxin" variables from "filters" and passes the "wordout" and unfiltered "txout" variables to "filters" for output. "Modem" is 14 bits wide in order to achieve the computational accuracy necessary to meet the demodulator bandpass and lowpass filter specifications. The modulator segment of "modem" produces one of four frequencies depending on O/$\overline{A}$ and TXD (see Table 1) or zero if SQT is set. The demodulator segment of "modem" takes a filtered word along with signals O/$\overline{A}$ and ALB as input and produces $\overline{CD}$ and RXD as output. If ALB is set, the demodulator operates in the frequency band opposite to that of its normal full-duplex operation. The necessity for this is apparent from the fact that in self-test mode only one frequency band can be used at a time.

IV. MODULATOR

The modulator segment in processor "modem" must produce one of four frequencies depending on $0/\overline{A}$ and TXD as shown in Table 1. These frequencies must be produced to within ±1 Hz, and transitions between mark and space frequencies must be phase coherent. Harmonics should be not greater than -32dB. Two different modulator designs were coded and tested.

IV-1. Sawtooth Wave Modulator

A digital sine wave oscillator may be implemented by first generating a simple sawtooth wave and then modifying it by piece-wise linear transformations [2]. The negatively sloped sawtooth wave of the appropriate frequency is generated by successively subtracting a value from a variable "WAVE." When WAVE is less than zero, one is added to WAVE to start the next period. The frequency of oscillation is determined by the equation $F_o = kF_s$ where F_o = oscillator frequency, k = step size, and F_s = sample frequency. For the modem design presented here, the sampling frequency was chosen to be 9600 Hz. Therefore, the step sizes shown in Table 3 are necessary to generate the four required frequencies.

Frequency	Step Size k
1070 Hz	0.111458
1270 Hz	0.132292
2025 Hz	0.210938
2225 Hz	0.231771

Table 3. Sawtooth Wave Step Sizes

Once the sawtooth wave has been generated, it must be shaped into a sine wave approximation to remove some harmonic energy. Notice that this shaping cannot be easily done by digital lowpass filtering since the sawtooth wave samples are not band limited, and aliasing of higher harmonics could interfere with the filtered waveform. Thus, the following piece-wise linear transformation is used to shape the sawtooth wave:

```
WAVE := WAVE - 0.5 (center WAVE vertically around 0)
WAVE := |WAVE*2| (triangle wave from 0 to 1)
WAVE := WAVE - 0.5 (triangle wave from 0.5 to -0.5)
WAVE := WAVE*3 (clipped triangle wave approximation to sine wave)
```

The last operation depends on the saturating overflow characteristic of the adder. The accumulator is purposely overflowed in order to clip the top of the triangle wave. The above transformation removes all even harmonics and attenuates the nth odd harmonic by $\sin(n\pi/3)|(n^2\pi/3)$ (see Reference [2]). Thus, the third harmonic is absent and the fifth harmonic is down by 30 dB.

The microcode listing of this modulator is given in Appendix B. By using 14-bit wordlengths, all frequencies are realized to within ±0.5 Hz (e.g., 9600/(8191/1084) = 1270.47 Hz, etc.). The transitions between frequencies are phase coherent since only the step size is adjusted when the frequency changes, and the continuity of variable WAVE is not upset. The performance of this modulator is demonstrated in Figures 4 and 5 which are 512 point FFTs of 512 data samples generated from "modem3.df" (Appendix B) at a sample rate of 9600 Hz. Figure 4 shows the originate mode space frequency of

1070 Hz. As expected, the aliased fifth harmonic at 4250 Hz is down by approximately 30 dB, and the aliased seventh harmonic (2110 Hz) is down by approximately 35 dB. These harmonics are removed by the transmit filter. Figure 5 shows the answer mode mark frequency of 2225 Hz. It is more difficult to distinguish the harmonics in this case, and some components appear to be attenuated by only 18 dB. This relatively high component level is probably caused by two aliased harmonics constructively interfering. When used with the transmit filter, this modulator scheme meets the specifications.

IV-2. Sine Table Look-up Modulator

The sine table look-up modulator produces a sine wave of the desired frequency by varying the angle increment (step size) of a table indexing variable. The step size through a table containing n sine values between 0 and 90 degrees for a desired frequency of f_o is given by: $4nf_o/9600$ = step size, where 9600 Hz is the system sample rate. A table of 24 values is used in the modem design file (Appendix A). Hence, the step sizes shown in Table 4 are necessary to produce the four required frequencies.

Frequency	Step Size
1070 Hz	10.70
1270 Hz	12.70
2025 Hz	20.25
2225 Hz	22.25

Table 4. Sine Table Look-up Step Sizes

Implementing the fractional part of the step is done as follows:

1070 Hz: Step by 11 each program cycle; however, step by 8 every 10th cycle. (Total of 107 steps in 10 cycles = 10*(10.70).)

1270 Hz: Step by 13 each cycle; however, step by 10 every 10th cycle

2025 Hz: Step by 20 each cycle, by 21 every 4th cycle.

2225 Hz: Step by 22 each cycle, by 23 every 4th cycle.

If the length of the table had been less than 24 values, accounting for the fractional step would have been quite difficult, and the modulator would produce the wrong frequency.

The table look-up modulator algorithm as coded in "modem.df" can be summarized as shown in Table 5. This modulator scheme produces frequencies whose accuracy depends only on the accuracy of the 9600 Hz sampling frequency. The transitions between frequencies are phase coherent since only the step size is adjusted when the frequency changes, and the index variable is not reset. The modulator performance is demonstrated in Figures 6 to 11 which are 512 point FFTs of 512 data samples generated by the modulator. As expected, the highband frequencies (2025 Hz and 2225 Hz) show very little harmonic content, since the step is never off by more than one (approximately four degrees). However, the lowband frequencies (1070 Hz and 1270 Hz) show significant harmonic content at many different frequencies. This lowband effect results from the fact that the step must be adjusted by three (twelve degrees) every tenth count. These harmonics are never greater than 31 dB. Figures 10 and 11 show the effect of the transmit bandpass filter on the

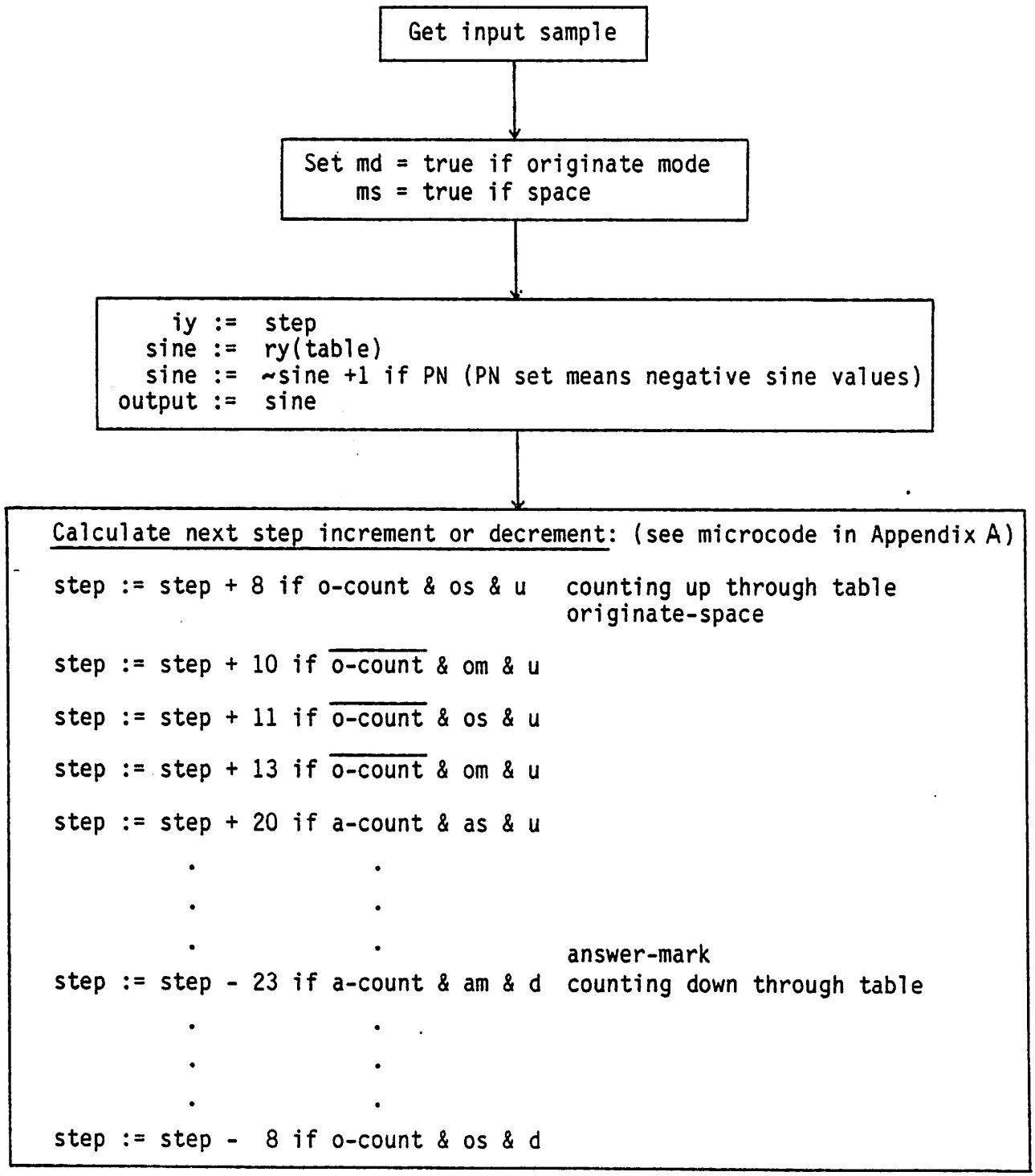


Table 5. Table Look-Up Modulator Algorithm

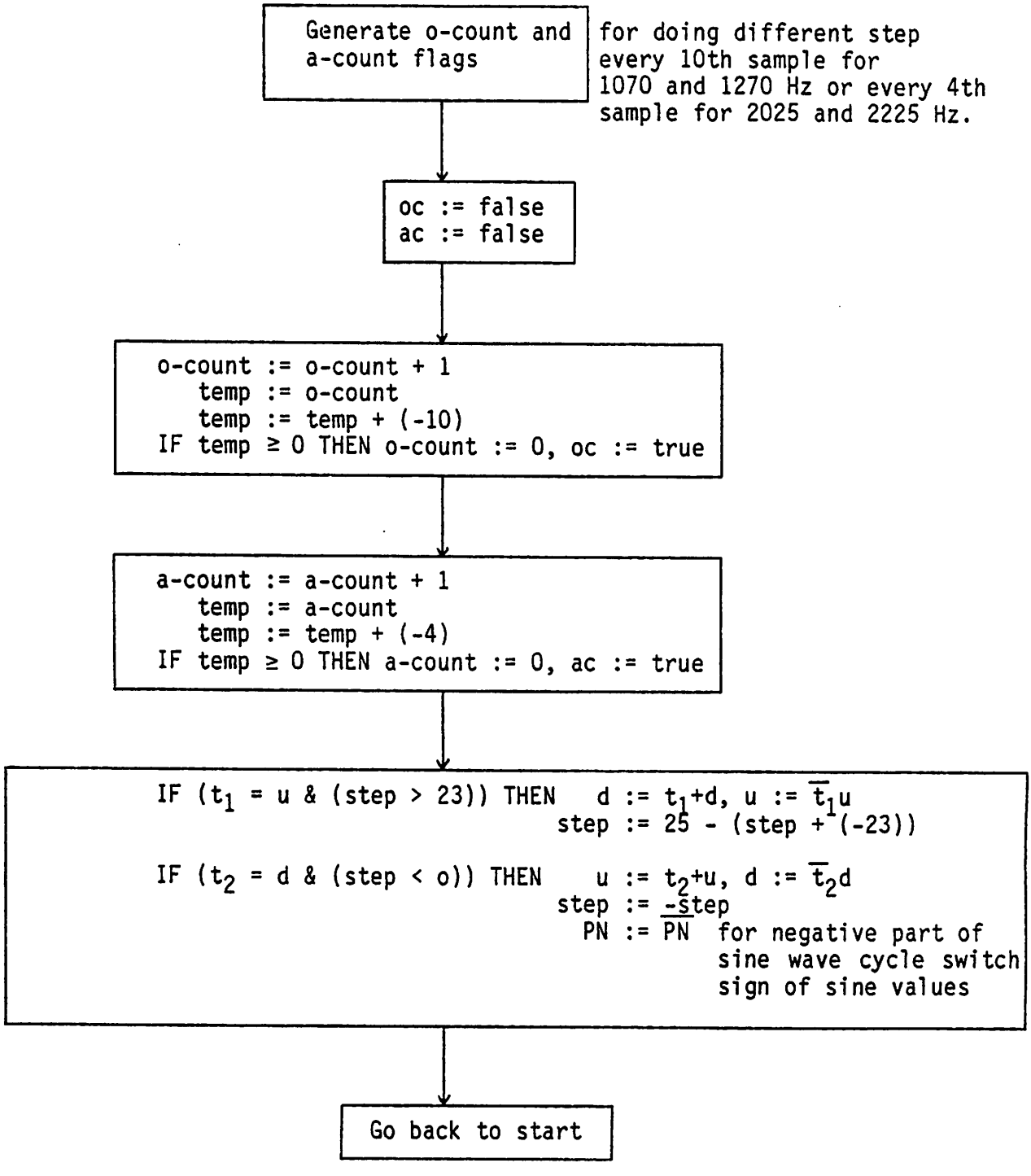


Table 5. (Continued)

modulator output. In each case, the harmonics have been removed and the signal is attenuated by at least 48 dB in the adjacent receiving band. This performance is more than adequate. Figures 12 and 13 show the effect of the bandpass filters on FSK modulated data that is toggling between mark and space (300 bps). All sidebands of the modulation have been significantly attenuated.

IV-3. Comparisons of Digital and Analog Modulators

Most analog modulators are based upon the principle of dividing down a high frequency input clock. The division factor is changed according to what frequency is desired, and this frequency is used to clock sine values from a ROM or PLA [3]. However, this concept cannot be applied to DSP modulators. DSP systems have a fixed sample rate (9600 Hz in this case), and values coming out of a modulator block must be at that system sample rate in order to be used as input to the next DSP functional block (digital filter in this case). The analog method implies faster sampling to generate higher frequency sine waves, since there are still the same number of sine values per cycle. Thus, in the DSP approach, the number of values per cycle of the desired sine wave frequency is varied. (Larger step size through a sine table for higher frequencies.)

Another major difference in the two approaches is that the digital input data is sampled for the DSP modulator while this data is not sampled in the analog modulator. The data causes the analog modulator to switch clocking rate as soon as the data changes (delayed only by circuit propagation delays). However, in the DSP modulator, a change in input data is not detected until the next sample is taken. This effect introduces modulator bit jitter of $1/F_S$, where F_S is the sampling frequency.

V. FILTERS

The lowband and highband bandpass filters in this modem must meet the requirements shown in Table 6.

	Lowband Filter	Highband Filter
Bandwidth	300 Hz	300 Hz
Center Frequency	1170 Hz	2125 Hz
Group delay variation between mark and space frequencies	≤100 μsec	≤100 μsec
Amplitude Response:		
50-600 Hz	<-20 dB	<-60 dB
1020-1320 Hz	1 dB ripple	<-65 dB
1975-2275 Hz	<-65 dB	1 dB ripple
3000-4800 Hz	<-55 dB	<-24 dB

Table 6. Filter Requirements

The filters were designed interactively using the computer program Filsyn [4]. The design was carried out in the digital frequency domain using a bilinear z-transform, and the sample frequency is the system sample frequency of 9600 Hz. For the lowband filter, zeros were placed at 2025 Hz and 2225 Hz in order to get sharp attenuation of the adjacent band signal frequencies. Similarly, for the highband filter, zeros were placed at 1070 Hz and 1270 Hz. Each filter has an equal ripple type passband. Sixth order

filters met the amplitude response specifications, but not the group delay requirement. Therefore, two second order sections were added to each filter to equalize the delay in the passband. This resulted in filters that are each tenth order.

Each filter is implemented as a cascade of five direct form II [5] second order sections (see Figure 14). Filsyn was used to order and scale the sections according to optimizations performed with respect to noise gain and overflow. The coefficients were rounded to seven places by Filsyn. Table 7 lists each filter's coefficients and scale factors in both decimal and canonical signed digit forms. Table 8 shows the performance of each filter as calculated by Filsyn. Amplitude response performance easily exceeds requirements and group delay performance is also within the requirements.

To implement the filters in microcode as listed in processor "filters" in Appendices A and B, the following equations must be calculated (in order) each sample period:

$$
\begin{aligned}
& & q_{i+2} &:= q_{i+1} \\
& & q_{i+1} &:= q_i \\
\text{For } i = 1, 4, 7, 10, 13 \quad & & q_i &:= \text{scale} * in_i + \alpha_{1i} * q_{i+1} + \alpha_{2i} * q_{i+2} \\
& & out_i &:= q_i + \beta_{1i} * q_{i+1} + \beta_{2i} * q_{i+2}
\end{aligned}
$$

Processor "filters" requires 20-bit wordlengths to accurately process data samples and to provide for the 65 dB of rejection in the adjacent band. Figures 15 to 18 show the impulse responses of the lowband and highband filters generated by processor "filters" using both 20-bit and 32-bit wordlengths. These graphs were generated using a 1024 point FFT on 512 output data samples from processor "filters." Notice that the impulse responses

generated using 32-bit wordlengths show negligible effects of finite register length and closely match the performance calculated by Filsyn.

	Lowband Filter		Highband Filter	
	Decimal	Canonical Signed Digit	Decimal	Canonical Signed Digit
Scale Factor	0.0625	2^{-4}	0.125	2^{-3}
Section 1				
$\alpha 1$	1.3125	$2^{0}+2^{-2}+2^{-4}$	0.3671875	$2^{-2}+2^{-3}-2^{-7}$
$\alpha 2$	-0.8515625	$-2^{0}+2^{-3}+2^{-5}-2^{-7}$	-0.8515625	$-2^{0}+2^{-3}+2^{-5}-2^{-7}$
$\beta 1$	-1.640625	$-2^{0}-2^{-1}-2^{-3}-2^{-6}$	-0.5703125	$-2^{-1}-2^{-4}-2^{-7}$
$\beta 2$	1.171875	$2^{0}+2^{-3}+2^{-5}+2^{-6}$	1.3984375	$2^{0}+2^{-2}+2^{-3}+2^{-6}+2^{-7}$
Scale Factor	0.25	2^{-2}	0.125	2^{-3}
Section 2				
$\alpha 1$	1.5078125	$2^{0}+2^{-1}+2^{-7}$	0.40625	$2^{-2}+2^{-3}+2^{-5}$
$\alpha 2$	-0.90625	$-2^{0}+2^{-3}-2^{-5}$	-0.7109375	$-2^{-1}-2^{-2}+2^{-5}+2^{-7}$
$\beta 1$	-0.484375	$-2^{-1}+2^{-6}$	-1.53125	$-2^{0}-2^{-1}-2^{-5}$
$\beta 2$	1.0	2^{0}	1.0	2^{0}
Scale Factor	0.5	2^{-1}	0.5	2^{-1}
Section 3				
$\alpha 1$	1.2421875	$2^{0}+2^{-2}-2^{-7}$	0.1328125	$2^{-3}+2^{-7}$
$\alpha 2$	-0.9375	$-2^{0}+2^{-4}$	-0.8984375	$-2^{0}+2^{-3}-2^{-5}+2^{-7}$
$\beta 1$	-1.4921875	$-2^{0}-2^{-1}+2^{-7}$	-0.2890625	$-2^{-2}-2^{-5}-2^{-7}$
$\beta 2$	1.203125	$2^{0}+2^{-3}+2^{-4}+2^{-6}$	1.2578125	$2^{0}+2^{-2}+2^{-7}$

Table 7. Filter Coefficients (see Figure 14)

	Lowband Filter		Highband Filter	
	Decimal	Canonical Signed Digit	Decimal	Canonical Signed Digit
Scale Factor	0.25	2^{-2}	0.25	2^{-2}
Section 4				
$\alpha 1$	1.2421875	$2^{0}+2^{-2}-2^{-7}$	0.5546875	$2^{-1}+2^{-4}-2^{-7}$
$\alpha 2$	-0.8359375	$-2^{0}+2^{-3}+2^{-5}+2^{-7}$	-0.9375	$-2^{0}+2^{-4}$
$\beta 1$	-0.2265625	$-2^{-2}+2^{-5}-2^{-7}$	-1.3515625	$-2^{0}-2^{-2}-2^{-3}+2^{-5}-2^{-7}$
$\beta 2$	1.0	2^{0}	1.0	2^{0}
Scale Factor	0.25	2^{-2}	0.5	2^{-1}
Section 5				
$\alpha 1$	1.3984375	$2^{0}+2^{-2}+2^{-3}+2^{-6}+2^{-7}$	0.2265625	$2^{-2}-2^{-5}+2^{-7}$
$\alpha 2$	-0.8515625	$-2^{0}+2^{-3}+2^{-5}-2^{-7}$	-0.796875	$-2^{0}+2^{-2}-2^{-4}+2^{-6}$
$\beta 1$	0	0	0	0
$\beta 2$	-1.0	-2^{0}	-1.0	-2^{0}
Scale Factor	2.125	$2^{1}+2^{-3}$	1.75	$2^{0}+2^{-1}+2^{-2}$
Overall Gain in Passband	0.93	-	0.97	-

Table 7. Filter Coefficients (See Figure 14) (Continued)

Frequency (Hz)	Lowband Filter		Highband Filter	
	Loss (dB)	Delay (sec*10^{-4})	Loss (dB)	Delay (sec*10^{-4})
100	48.52	1.930	89.41	0.945
600	26.58	4.396	77.16	1.16
1020	0.0042	51.69	92.72	1.81
1070	-0.1367	53.59	119.28	1.95
1170	-0.1283	53.19	89.32	2.29
1270	0.1103	53.32	108.09	2.77
1320	0.2599	61.49	81.11	3.08
1975	--	--	0.96	54.80
2025	115.05	1.935	0.83	43.92
2125	86.14	1.583	0.51	44.66
2225	119.23	1.327	0.44	44.68
2275	--	--	0.59	45.77
3000	67.97	0.552	32.72	2.46
3500	68.59	0.407	41.07	1.18
4000	71.93	0.339	47.92	0.794
Group delay between mark and space frequencies	27 μsec		76 μsec	

Table 8. Filter Impulse Response Performance

VI. DEMODULATOR

VI-1. Automatic Gain Control (AGC)

To ensure accurate demodulaton of the received signal, an automatic gain control (AGC) function is implemented after the receive filter. The AGC output is at a constant amplitude regardless of the amplitude of its input. Although most analog implementations of an AGC use a hard limiter to achieve the constant signal amplitude, this method cannot be used reliably in a DSP approach. A hard limiter is a nonlinear function, and it produces harmonics of the signal frequency. In an analog system, these harmonics can be removed by a lowpass filter; however, in a DSP system these harmonics alias down and interfere with the signal. It is therefore desirable to implement a linear AGC function.

The AGC function used in the modem is shown in Figure 19 [6]. The full wave rectifier (FWR) and lowpass filter act as an envelope detector. When the FWR output is divided by the envelope, the result is a signal uniform in maximum level. This level depends on the gain of the lowpass filter and in this modem is calculated to be slightly less than the maximum. Restoring the original sign of the signal is just the inverse function of the FWR.

The FWR is realized using an absolute value instruction. The divide function is implemented using microcode instructions that essentially execute the divide algorithm discussed in Chapter II. The lowpass filter is third order with a cutoff frequency of 300 Hz. It is the same as the one used in the demodulator, except it has a gain of 1.625 in this case. This gain factor is calculated so that sine wave dc levels of 0.7 are scaled up to be larger than one.

The AGC function is tested using an input signal (2225 Hz) that is attenuated by 24 dB. The wordlength of the inputs and outputs to the modem design file is twelve; therefore, the maximum level possible is 2047 and a signal down by 24 dB has a maximum level of 127. The observed signal output from the AGC is normalized to the maximum level of 2047. An FFT analysis of the two signals shows that the spectral properties of the original signal are unchanged by the AGC.

VI-2. Carrier Detect

The carrier detect signal ($\overline{CD}$) indicates the presence or absence of a carrier in the received signal to be demodulated. The level of the carrier signal is available at the output of the lowpass filter in the AGC function. This level is compared with a threshold to produce the $\overline{CD}$ signal. The requirements for the $\overline{CD}$ signal are as follows:

$\overline{CD}$ should turn off (=1) at -48 dB (10 msec delay)

$\overline{CD}$ should turn on (=0) at -43 dB (20 msec delay)

Hysteresis is provided in the turn-off and turn-on levels of the $\overline{CD}$ signal to prevent flickering of $\overline{CD}$ as the carrier is lost or regained. The delays associated with the turn on and turn off of $\overline{CD}$ are desirable to prevent $\overline{CD}$ from responding to noise spikes, quick fades, or surges of the signal. The microcode algorithm used for generating the $\overline{CD}$ signal is shown in Table 9. This algorithm was adopted from reference [6].

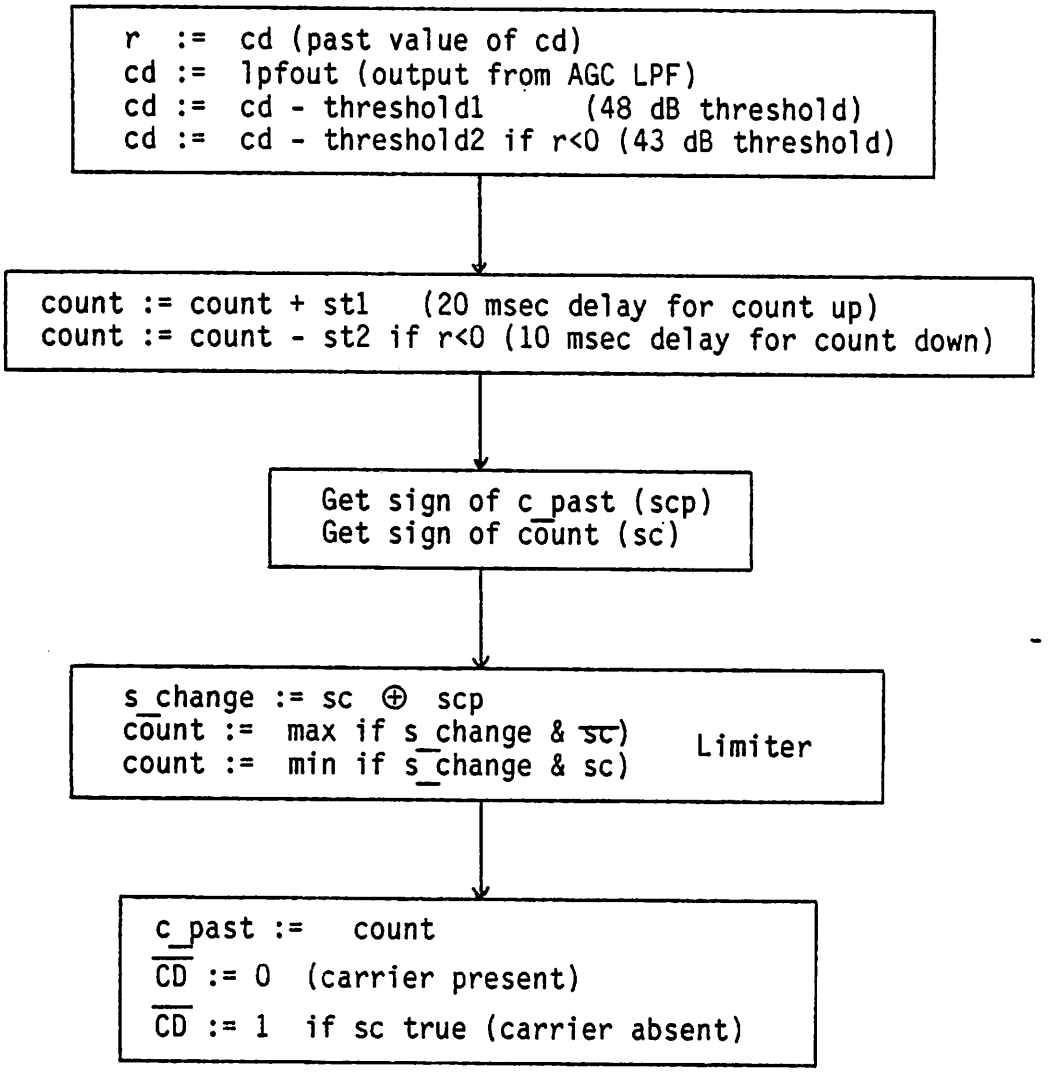


Table 9. Carrier Detect Algorithm

Constants st1 and st2 are calculated to give a 20 msec turn on delay and a 10 msec turn off delay from the equation:

$$delay = (8191)/(st*9600)$$

The threshold constants are calculated from the equations:

$$th1 = (8191)*10^{-48/20}$$

$$th2 = (8191)*10^{-43/20}$$

The carrier detect function is tested using step sizes st1 = 164 and st2 = -328 (st2 = -328 corresponds to using a -164 step since the program always adds st1). These step sizes correspond to on and off delays of 50 samples. Table 10 shows the characteristics of the input test signal and $\overline{CD}$ output. Note that both the delay interval of 50 samples and the hysteresis feature are tested by the "recin" test data. Referring to the table, "status" means that the signal takes on this state at the specified sample number and retains it until the next sample number. The $\overline{CD}$ signal does not turn on until sample 64 because of delays associated with the initial transient response of the receive filter and the lowpass filter in the AGC function. Filter delays also account for $\overline{CD}$ not turning off until sample 229.

"recin" Signal		$\overline{CD}$ Signal	
Sample Number	Status	Sample Number	Status
0	0 dB	0	off (=1)
64	-60 dB	64	on (=0)
256	0 dB	229	off
272	-60 dB	318	on
304	-24 dB	496	on
432	-45 dB		
496	-45 dB		

Table 10. Carrier Detect Test Data

VI-3. Delay Line Discriminator

In this scheme, the FSK signal is demodulated using a delay line of one or two sample delays, a multiplier, a lowpass filter (LPF), and a threshold comparator. Figure 20 shows a block diagram of this demodulator.

The frequency of the received FSK signal determines the level at the output of the LPF. Therefore, the output of the LPF can be compared with a threshold to determine the correct output data. Specifically, let:

FSK input = $A\cos(\omega t)$

delay = T = one sample period

$\phi = \omega T$ or $\omega 2T$

Then: $A\cos(\omega t)*A\cos(\omega t + \phi)$ describes the delayed signal multiplied by the undelayed signal:

$$A\cos(\omega t)*A\cos(\omega t + \phi) = \frac{A^2}{2}\cos(\phi) + \frac{A^2}{2}\cos(2\omega t + \phi)$$

The LPF removes the $2\omega t$ term and leaves a voltage proportional only to ω since $T = 1/9600$ and A are both fixed. "A" is held constant at approximately one by the AGC function.

To make the threshold values as close to zero as possible (for discriminator linearity), one delay is used for demodulation in the highband (originate mode), and two delays are used for demodulation in the lowband (answer mode). Each threshold is chosen to correspond to halfway between the mark and space frequencies and is calculated as follows:

Highband Threshold:

$\omega = 2\pi*(2125)$
$T = 1/9600$
$A = 1.0$
$\frac{A^2}{2}\cos(\omega T) = \frac{A^2}{2}(0.179)$
$= 0.0895$
$= 733$ (for 14-bit processor where maximum value is 8191)

Lowband Threshold:

$\omega = 2\pi*(1170)$
$T = 1/9600$
$A = 1.0$
$\frac{A^2}{2}\cos(\omega 2T) = \frac{A^2}{2}*(0.393) = 0.0196$
$= 161$ (for 8191 maximum value)

The design file listing for this demodulator is shown in Appendix C under file name "dm3.df". The originate threshold was adjusted empirically (to 680 instead of 733) to obtain the best demodulation characteristic. The

delays are implemented in the same manner as the delays in the bandpass filters discussed previously. The multiply function is implemented in microcode using instructions which allow the designer to implement the algorithm discussed in Chapter II. The LPF used in this demodulator is the same as the one used in the demodulator of the following section and will be discussed in that section. It should be noted that the LPF must be able to attenuate frequencies which are generated by the multiply which are at twice the signal frequencies. The minimum of these frequencies is 2140 Hz, so the LPF stopband should begin at about 2000 Hz.

This demodulator was tested in both modes on FSK data that was toggling between the mark and space frequencies. The data was toggled at 150 Hz, corresponding to the maximum rate of 300 bits per second (bps). At this rate, 1 bit is represented by 32 samples (since the sampling rate is 9600 Hz). The requirement for the demodulator is that bit jitter be 100 usec. In this modem, the minimum bit jitter that can be observed (besides zero) is 1 sample, or 1/9600 = 104 usec. In both the originate and answer modes, the demodulated data showed no bit jitter.

VI-4. Bandpass Filters Demodulator

This demodulator compares the signal energy at the mark and space frequencies to determine which one is being received. A block diagram is shown in Figure 21.

The bandpass filters are centered slightly above the mark frequency and slightly below the space frequency so that there is a reasonably linear discriminator response curve. The linearity is important to minimize jitter in the receive data. The filters should show an equal attenuation (of 5-10 dB) at the center frequency between mark and space. Four different bandpass filters must be realized corresponding to the four possible FSK data frequencies.

The bandpass filters and the lowpass filter were designed interactively using the Filsyn program (see Chapter V). Again, the designs were carried out in the digital frequency domain using a bilinear z-transform. The sampling frequency is 9600 Hz. Second order bandpass filters proved to be sufficient, and each is implemented as one second order section (see Figure 14). The filters all have a maximally flat passband and monotonic stopband. The coefficients were rounded to seven places using Filsyn and are shown in Table 11 in both decimal and canonical signed digit forms. Table 12 shows the performance of each filter as calculated by Filsyn. For the originate mode frequency betwen mark and space of 2125 Hz, the mark filter shows an attenuation of 8.74 dB while the space filter shows an attenuation of 8.44 dB. As expected, these two values are close together. For the answer mode frequency between mark and space of 1170 Hz, the mark filter shows an attenuation of 8.95 dB while the space filter shows an attenuation of 8.12 dB. Again, the attenuations are close enough to provide the linearity in discriminator response that is desirable.

Originate Mode:	Mark Filter Decimal	Mark Filter Canonical Signed Digit	Space Filter Decimal	Space Filter Canonical Signed Digit
Scale Factor	0.015869141	$2^{-6} + 2^{-12}$	0.015869141	$2^{-6} + 2^{-12}$
$\alpha 1$	0.1875	$2^{-3} + 2^{-4}$	0.5	2^{-1}
$\alpha 2$	-0.9375	$-2^{0} + 2^{-4}$	-0.9375	$-2^{0} + 2^{-4}$
$\beta 1$	0	0	0	0
$\beta 2$	-1.0	-2^{0}	-1.0	-2^{0}
Scale Factor	2.0	2^{1}	2.0	2^{1}
Answer Mode:				
Scale Factor	0.015869141	$2^{-6} + 2^{-12}$	0.015869141	$2^{-6} + 2^{-12}$
$\alpha 1$	1.28125	$2^{0} + 2^{-2} + 2^{-5}$	1.5	$2^{0} + 2^{-1}$
$\alpha 2$	-0.9375	$-2^{0} + 2^{-4}$	-0.9375	$-2^{0} + 2^{-4}$
$\beta 1$	0	0	0	0
$\beta 2$	-1.0	-2^{0}	-1.0	-2^{0}
Scale Factor	2.0	2^{1}	2.0	2^{1}

Table 11. Demodulator Bandpass Filters Coefficients

Mark Frequency Filter		Space Frequency Filter	
Frequency (Hz)	Loss (dB)	Frequency (Hz)	Loss (dB)
Originate Mode:			
2000	14.34	1750	14.44
2025	13.45	1875	8.75
2050	12.47	1950	3.07
2125	8.74	2000	-0.13
2200	3.16	2025	0.77
2225	1.00	2050	2.82
2250	-0.13	2125	8.44
2300	2.77	2200	12.14
2375	8.45	2225	13.11
2500	14.08	2250	13.99
Center Frequency = 2250 Hz		2000 Hz	
Answer Mode:			
1045	14.94	795	15.30
1070	13.97	920	9.14
1095	12.91	995	3.23
1170	8.95	1045	-0.13
1245	3.10	1070	0.70
1270	0.95	1095	2.67
1295	-0.13	1170	8.12
1345	2.79	1245	11.66
1420	8.28	1270	12.58
1545	13.63	1295	13.41
Center Frequency = 1295 Hz		1045 Hz	

Table 12. Demodulator Bandpass Filters Performance (Impulse Response)

To implement the filters in microcode as listed in processor "modem" in Appendices A and B, the same types of equations as described in Chapter V must be calculated. Since all four bandpass filters have the same coefficients except for α_1, a subprogram is written that calculates the equations for the desired bandpass filter by calling the proper α_1 coefficient from memory depending on the mode and whether the mark or space frequency filter is desired. The subprogram executes twice each sample period. The first iteration corresponds to the space frequency filter, and the second iteration corresponds to the mark frequency filter. Similarly, the equations for the lowpass filter in the space frequency path are calculated in the first subprogram iteration, while the equations for the lowpass filter in the mark frequency path are calculated in the second iteraton. This "functional multiplexing" reduces by a factor of two the number of microcode lines that have to be written.

Processor "modem" requires a 14-bit wordlength to produce accurate bandpass filter impulse responses. Figure 22 shows the answer mode mark and space frequency bandpass filter impulse responses generated by 512 point FFTs run on 256 output data samples from processor "modem." Figure 23 shows the originate mode bandpass filter impulse responses. In both cases, the discriminator response is fairly linear.

Each LPF in the two paths of the demodulator shown in Figure 21 is exactly the same. This LPF must attenuate frequency components at twice the signal frequencies which are generated by the FWR operation. It must have a bandwidth wide enough to allow the fastest rate of data to easily pass through. In a 300-baud modem, the fastest data rate is 150 Hz. The LPF cutoff frequency is chosen to be 300 Hz to make sure that 150 Hz data rates

are not hindered by the filter. The LPF should have a step response rise time that is significantly less than the length of a bit (3.33 msec) so that the LPF can respond quickly enough to changes in data. The step response overshoot should be as small as possible (less than 10%). With these design considerations in mind, Filsyn was used to design the LPF. The resulting LPF is third order with a maximally flat passband and an equal ripple stopband. The filter is implemented as a cascade of one second order section and one first order section. The coefficients were rounded to seven places using Filsyn and are listed in Table 13 in both decimal and canonical signed digit forms. The step response rise time is approximately 1 msec and the overshoot is approximately 5%. Figure 24 shows the LPF impulse response generated by a 256 point FFT on 256 output data samples from processor "modem." The impulse response of the actual microcoded version agrees well with the impulse response values calculated by Filsyn which are shown in Table 14.

VI-5. Demodulation Using a Digital Phase Locked Loop

Another possible method of demodulation is to use a Digital Phase Locked Loop (DPLL) [7,8]. A block diagram of this scheme is shown in Figure 25. This demodulation scheme was not coded into a design file; however, a brief description of how each functional block might be implemented is given below.

The phase detector could consist simply of subtracting the voltage controlled oscillator (VCO) output from the incoming signal. The loop filter could be the LPF used in the demodulator described in VI-4; however, this would have to be examined in more detail. The VCO should produce a reasonably band-limited signal so that aliasing terms do not significantly affect the

Section 1:	Decimal	Canonical Signed Digit
Scale Factor	0.09375	$2^{-4} + 2^{-5}$
$\alpha 1$	1.5625	$2^{0} + 2^{-1} + 2^{-4}$
$\alpha 2$	-0.6875	$-2^{-1} - 2^{-3} - 2^{-4}$
$\beta 1$	-0.25	-2^{-2}
$\beta 2$	1.0	2^{0}
Section 2:		
Scale Factor	0.125	2^{-3}
$\alpha 1$	0.65625	$2^{-1} + 2^{-3} + 2^{-5}$
$\alpha 2$	0	0
$\beta 1$	1.0	2^{0}
$\beta 2$	0	0

Table 13. Demodulator Lowpass Filter Coefficients

Frequency (Hz)	Loss (dB)
100	0.40
150	0.40
300	0.44
1500	30.06
2000	47.78
2500	50.72
3000	47.39
4500	60.38

Table 14. Demodulator Lowpass Filter Impulse Response

signal going through the LPF. The sawtooth wave generator (with sine wave shaping) described in Chapter IV could be used here. The step size for the VCO would have to be a variable in this case (loaded from a scaled LPF output) instead of one of several constants as before. A threshold comparator, programmable between two different thresholds depending on the mode of operation, would distinguish the size of the step driving the VCO, and therefore, the resulting data.

Since the mark and space frequencies in each mode are fairly close together, there would be only a small difference in the step sizes driving the VCO, and switching of data frequencies might be accompanied by a fairly large amount of bit jitter. This effect could be reduced by increasing the number of bits in the processor. Other considerations such as the DPLL lock range and response time would also have to be examined.

VII. MODEM TESTS AND CONCLUSIONS

A variety of tests were run on the "modem.df" design file. the modem was tested in both the full-duplex and self-test modes. Receive data attenuated by 36 dB, data at 400 bits per second, and the modulator squelch feature were all tested. Results of the tests are summarized in Table 15. In all cases, except when the data rate is higher than the designed rate of 300 bps, bit jitter is not more than one sample, which is equal to 104 usec. This performance is within the modem specifications.

The two modem design files in Appendices A and B were run through the DSP IC Layout Generator. Various chip layout plots (including a CIF plot [9]) are shown in Appendix D. The design file "modem.df" produces an awkward layout resulting in a more rectangular chip. The large finite state machine and large RAM in processor "modem" are the main problems in this case. The layout of the design file "modem3.df" produces a more compact chip that is reasonably square. It is expected that the modem using the sawtooth wave modulator has the same performance as the modem using the table look-up modulator. The reason for this equality is that the spectrums of the frequencies produced by the modulators are the same after being filtered by the transmit filter. Because of its desirable chip layout characteristics, the modem described by "modem3.df" is the best choice for actual fabrication.

The CAD tools of the DSP IC Layout Generator system worked well throughout the design of the 300-baud modem. However, writing the design

files is somewhat tedious and can take a relatively long time. Hopefully, the length of time it takes a designer to produce a chip using the DSP IC Layout Generator will be further shortened with the development of a higher level language compiler that will automatically generate the microcode instructions in the design files.

Test	Modem.df files				Bit jitter observed
	ctrlin	recin	transout	dataout	
1) SQT set true	64 samples of "512"	-	64 samples of 0	-	-
2) Full-duplex originate mode	512 samples of toggled data of 150 Hz (-2048,-1024)	512 toggled FSK data from transout of test 3	512 samples FSK toggled data for test 3	demodulated data from test 3. (negative numbers = mark, positive or zero = space)	0
3) Full-duplex answer mode	512 samples of toggled data at 150 Hz (0,1024)	512 toggled FSK data from transout of test 2	512 samples FSK toggled data for test 2	demodulated data from test 2	1 sample = 1/32 of a bit = 104 sec
4) 400 bps in answer and originate modes ALB set = self-test	512 samples of toggled data at 200 Hz (256,1280)	-	512 samples FSK data	demodulated data from transout	2 samples = 1/12 of a bit = 208 sec
5) Receive signal down by 36 db, originate mode	512 samples of toggled data at 150 Hz (-2048,-1024)	512 samples generated by pascal program. FSK toggled data	-	demodulated recin data	0
6) Self-test mode (ALB set) originate mode (use lowband)	512 samples pattern of 2 marks, 1 space (-1792,-768)	-	512 samples FSK data	demodulated data from transout	1 sample = 1/32 of a bit
7) self-test mode, originate mode	512 samples pattern of 2 spaces, 1 mark (-1792,-768)	-	512 samples FSK dta	demodulated data from transout	1 sample = 1/32 of a bit

Table 15. Modem Test Results

Test	Modem.df files				Bit jitter observed
	ctrlin	recin	transout	dataout	
8) Self-test mode, answer mode	512 samples pattern of 2 marks, 1 space (256,1280)	-	512 samples FSK data	demodulated data from transout	1 sample = 1/32 of a bit
9) Self-test mode, answer mode	512 samples pattern of 2 spaces, 1 mark (256,1280)	-	512 samples FSK data	demodulated data from transout	1 sample = 1/32 of a bit

Table 15. Modem Test Results (Continued)

VIII. REFERENCES

[1] Rabaey, Jan, Parallel Digital Signal Processors: An Automated Macrocell Approach, University of California, Berkeley, Electronics Research Lab, July 1984.

[2] 2920 Analog Signal Processor Design Handbook, Intel Corporation, August 1980.

[3] Chan, Anthony, et. al., CMOS 300 Baud Modem, National Semiconductor AN-349, September 1983.

[4] Szentirmai, George, Super-Filsyn User Manual, Version 1.0, Comsat General Integrated Systems, March 1982.

[5] Oppenheim, A. V. and Schafer, R. W. Digital Signal Processing, Prentice-Hall, Englewood Cliffs, NJ, 1975.

[6] Rittenhouse, John, Using the 2920 Signal Processor in Modem Applications, Application Note AP-117, Intel Corporation, June 1981.

[7] Langston, J. Leland, "μC Chip Implements High-Speed Modems Digitally," Electronic Design, June 24, 1982.

[8] Ziemer, R. E. and Tranter, W. H., Principles of Communications, Houghton Mifflin Co., USA, 1976.

[9] Mead, Carver and Conway, Lynn, Introduction to VLSI Systems, Addison-Wesley Co., Menlo Park, CA, 1980.

[10] Rabiner, Lawrence R. and Gold, Bernard, Theory and Application of Digital Signal Processing, Prentice-Hall, Englewood Cliffs, NJ, 1975.

[11] Takla, Ashraf, et. al., "A 300-Baud Frequency Shift Keying Modem," IEEE ISSCC Digest of Technical Papers, pp. 188-189, February, 1984.

[12] Yamamoto, Kazushige, "An Asynchronous FSK Modem," IEEE ISSCC Digest of Technical Papers, pp 190-191, February, 1984.

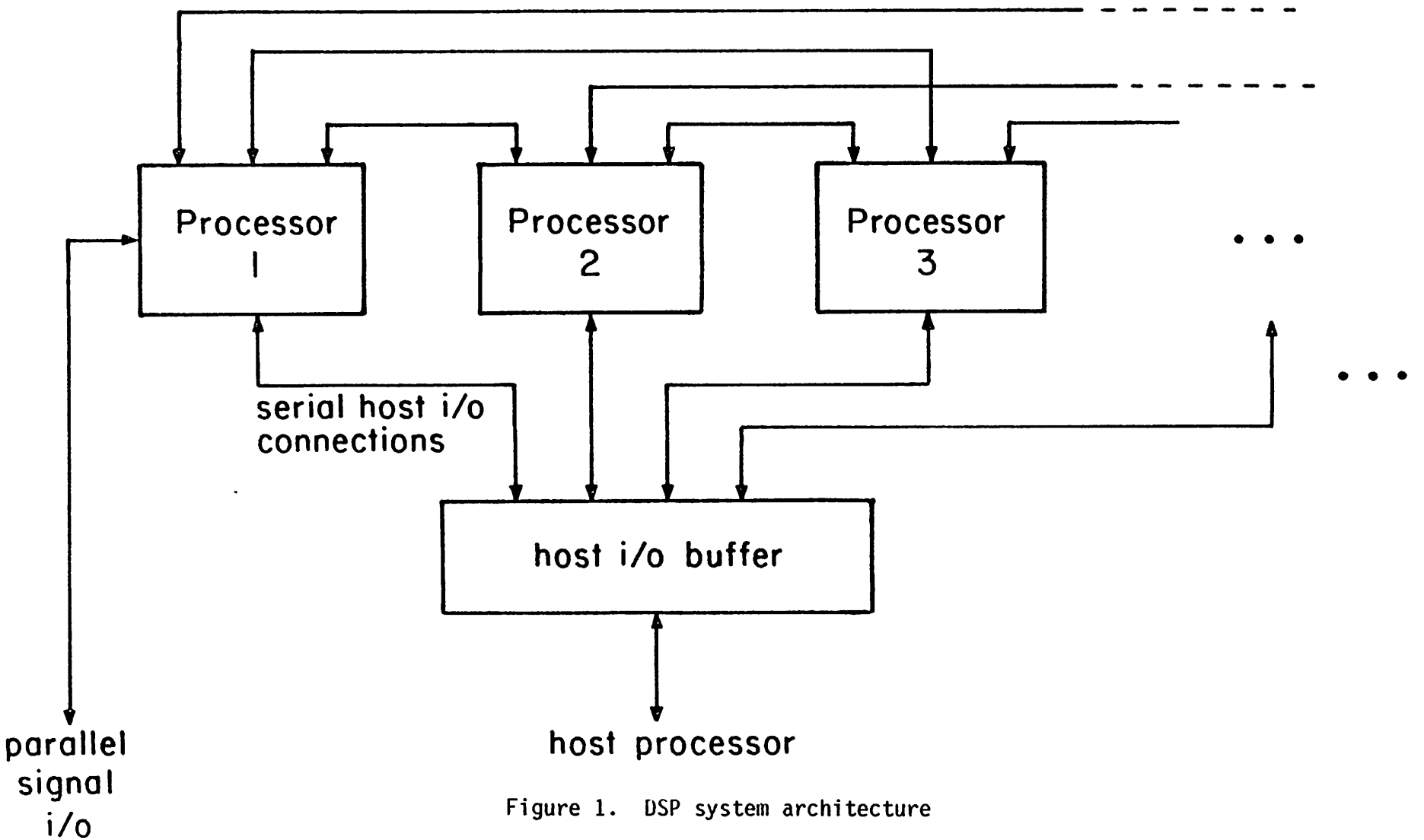


Figure 1. DSP system architecture

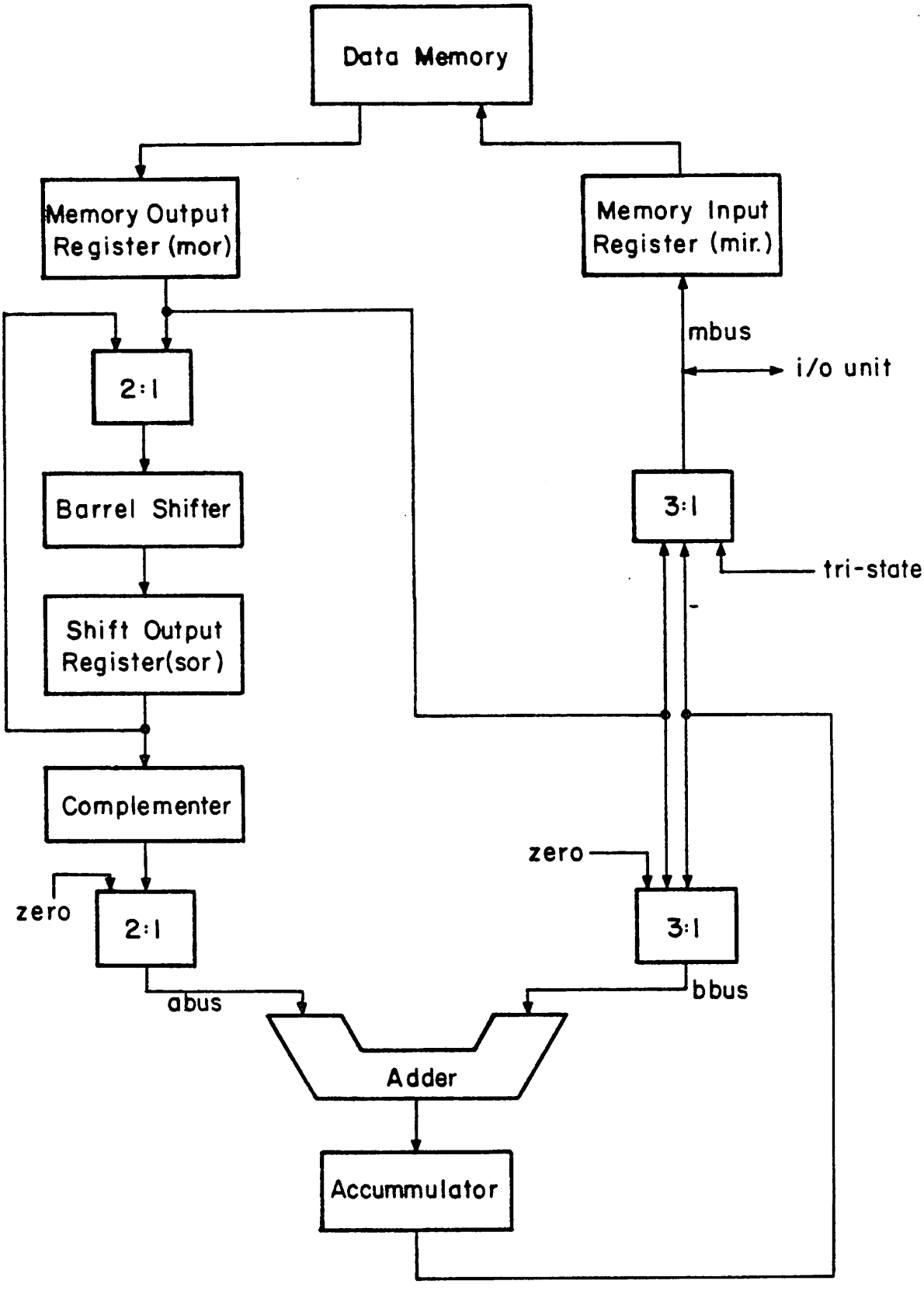
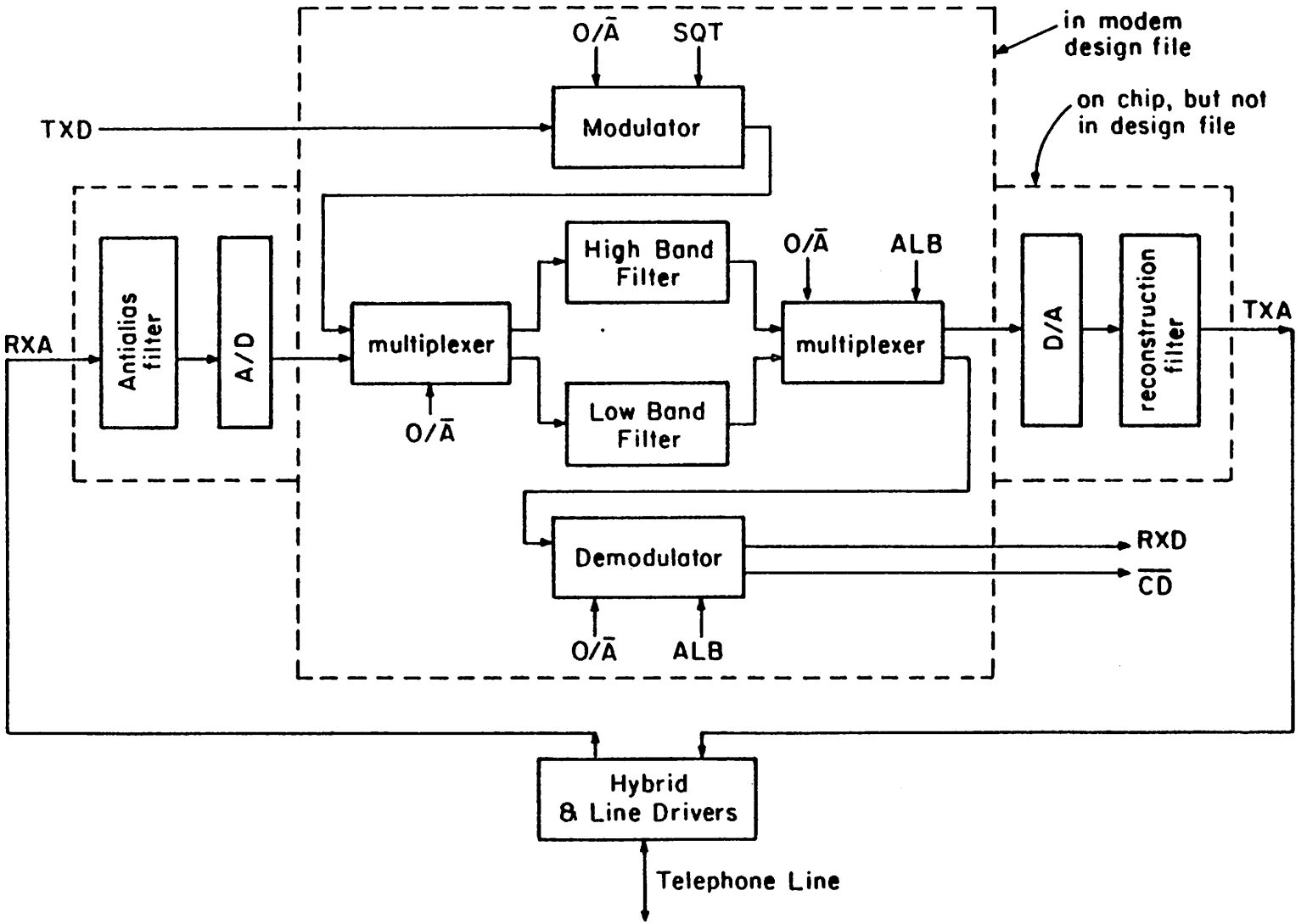


Figure 2. Processor arithmetic unit



O/Ā = mode
CD = carrier detect
ALB = self-test
SQT = squelch

TXD = txin = digital data to be transmitted
TXA = txout = FSK modulated data (analog)
RXA = recin = FSK modulated data (analog)
RXD = recout = demodulated digital data

Figure 3. Full-duplex 300-baud FSK modem

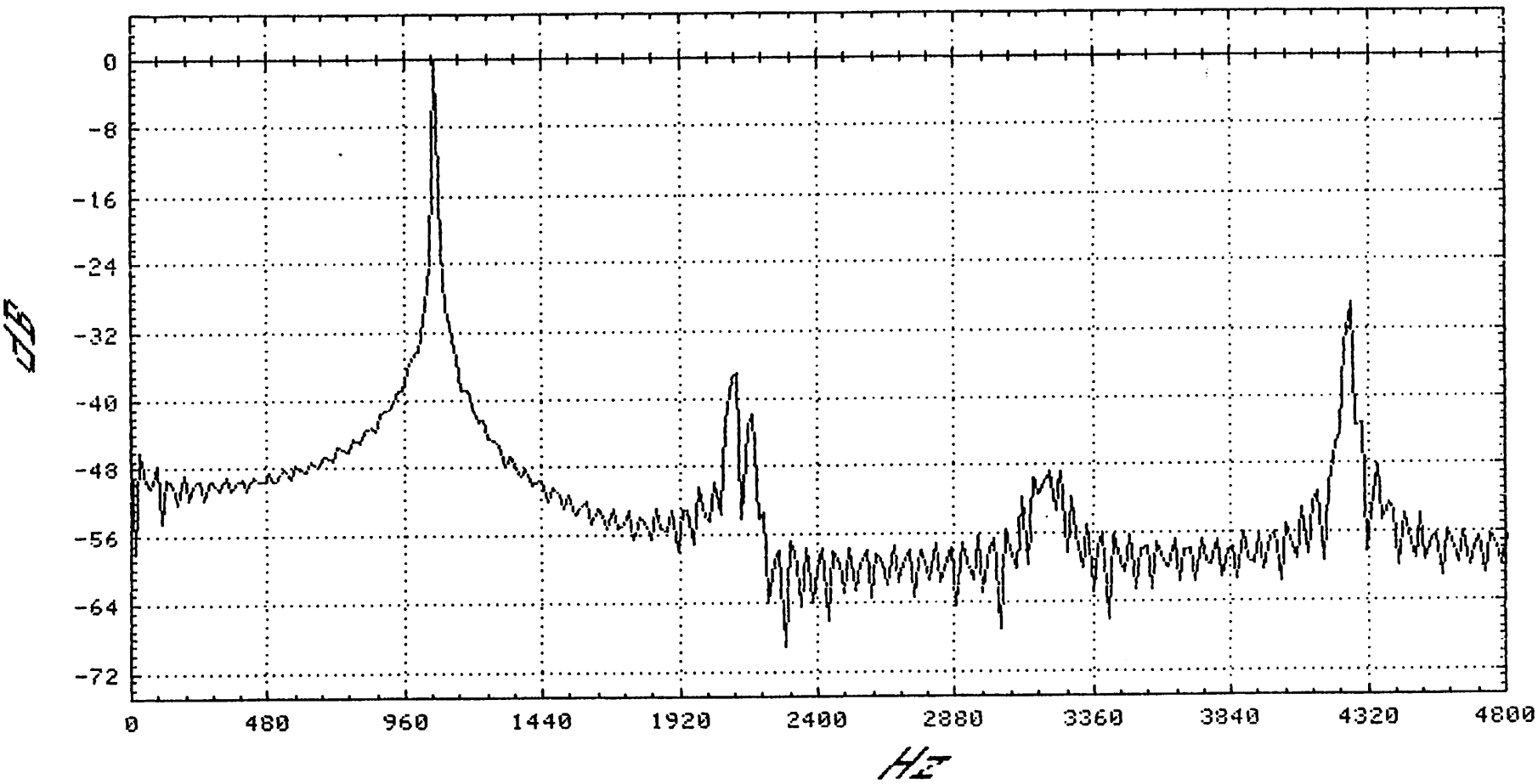


Figure 4. Sawtooth modulator 1070 Hz space frequency

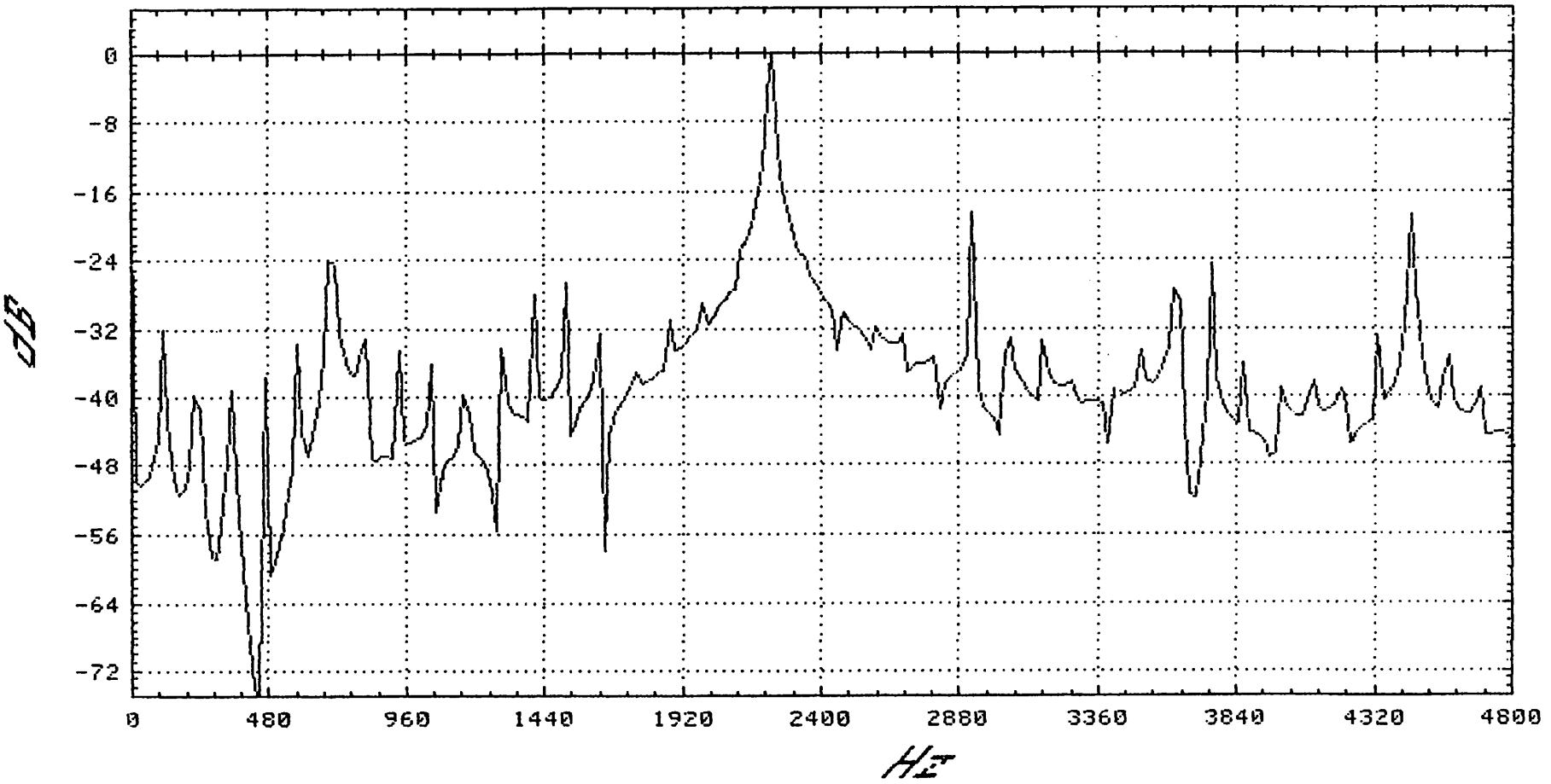


Figure 5. Sawtooth modulator 2225 Hz mark frequency

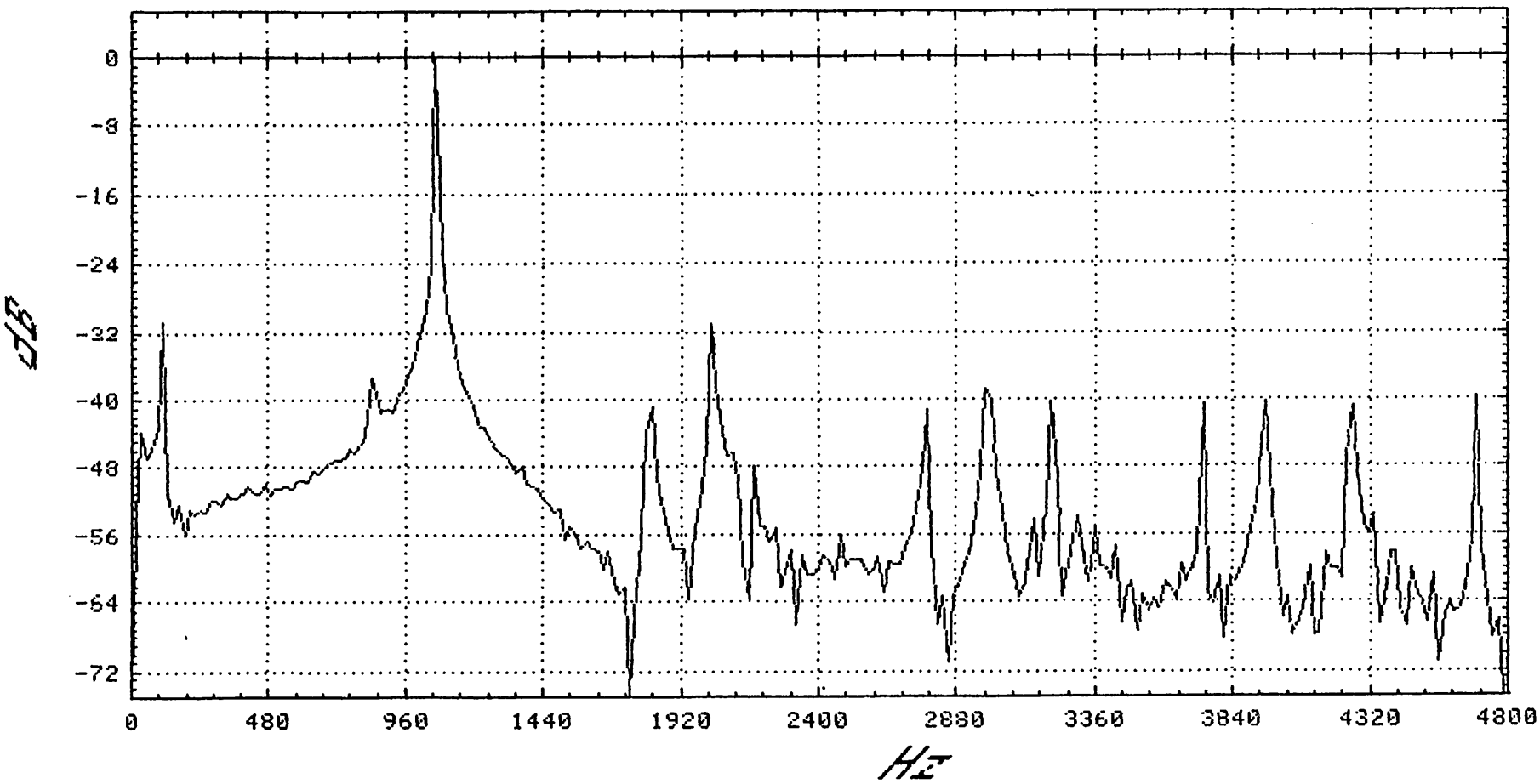


Figure 6. Table look-up modulator 1070 Hz

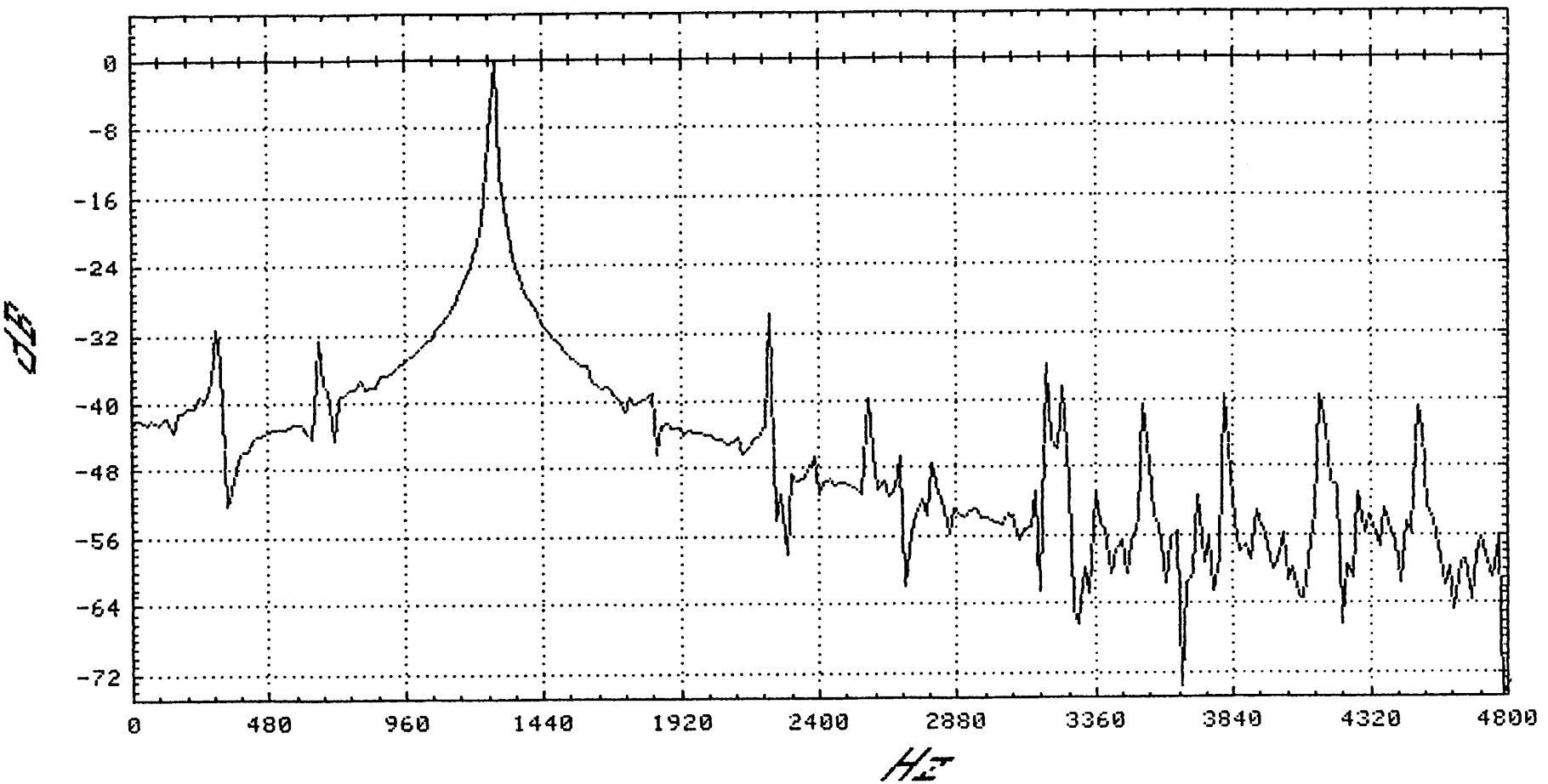


Figure 7. Table look-up modulator 1270 Hz

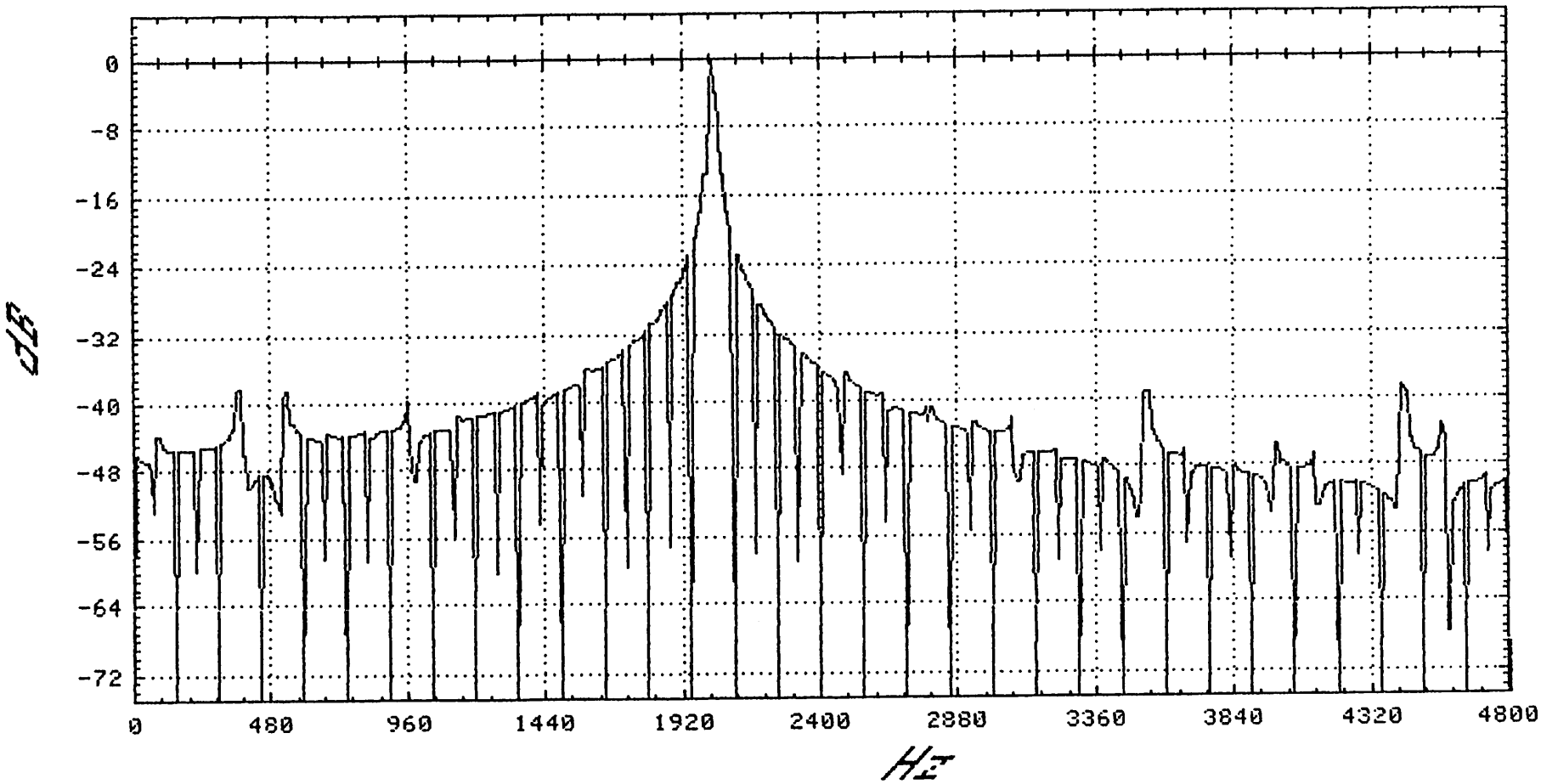


Figure 8. Table look-up modulator 2025 Hz

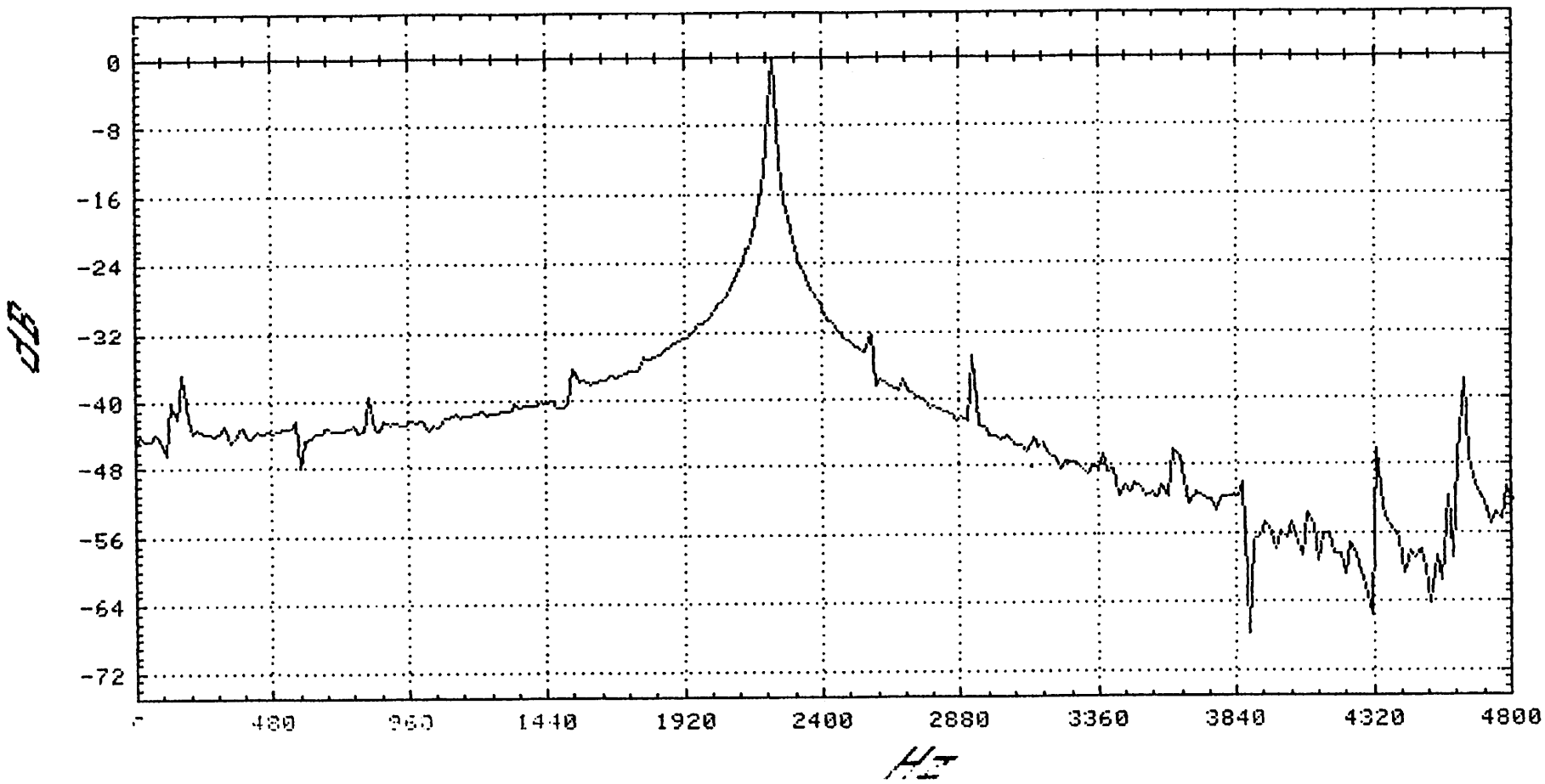


Figure 9. Table look-up modulator 2225 Hz

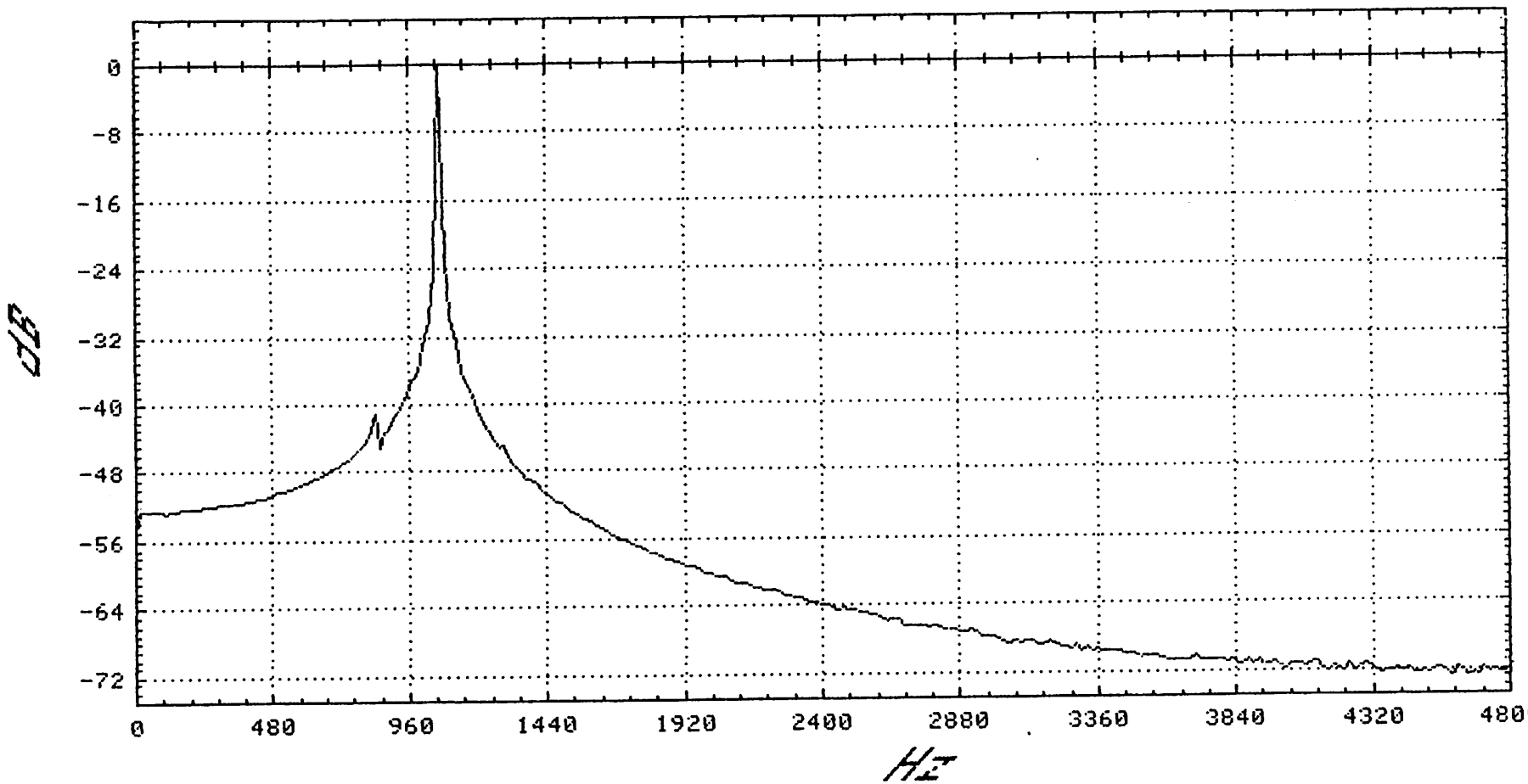


Figure 10. Table look-up modulator 1070 Hz after lowband filter

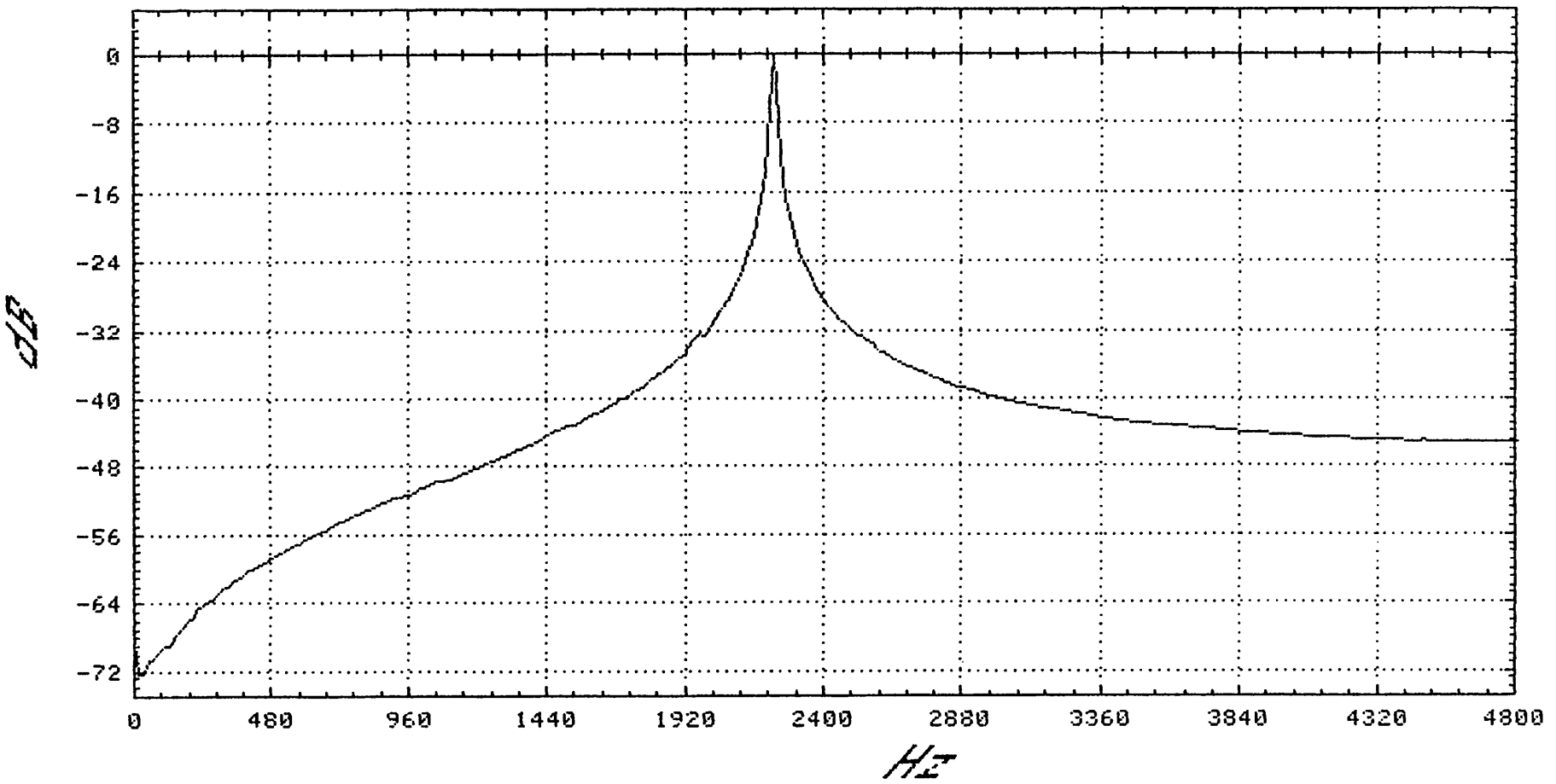


Figure 11. Table look-up modulator 2225 Hz after highband filter

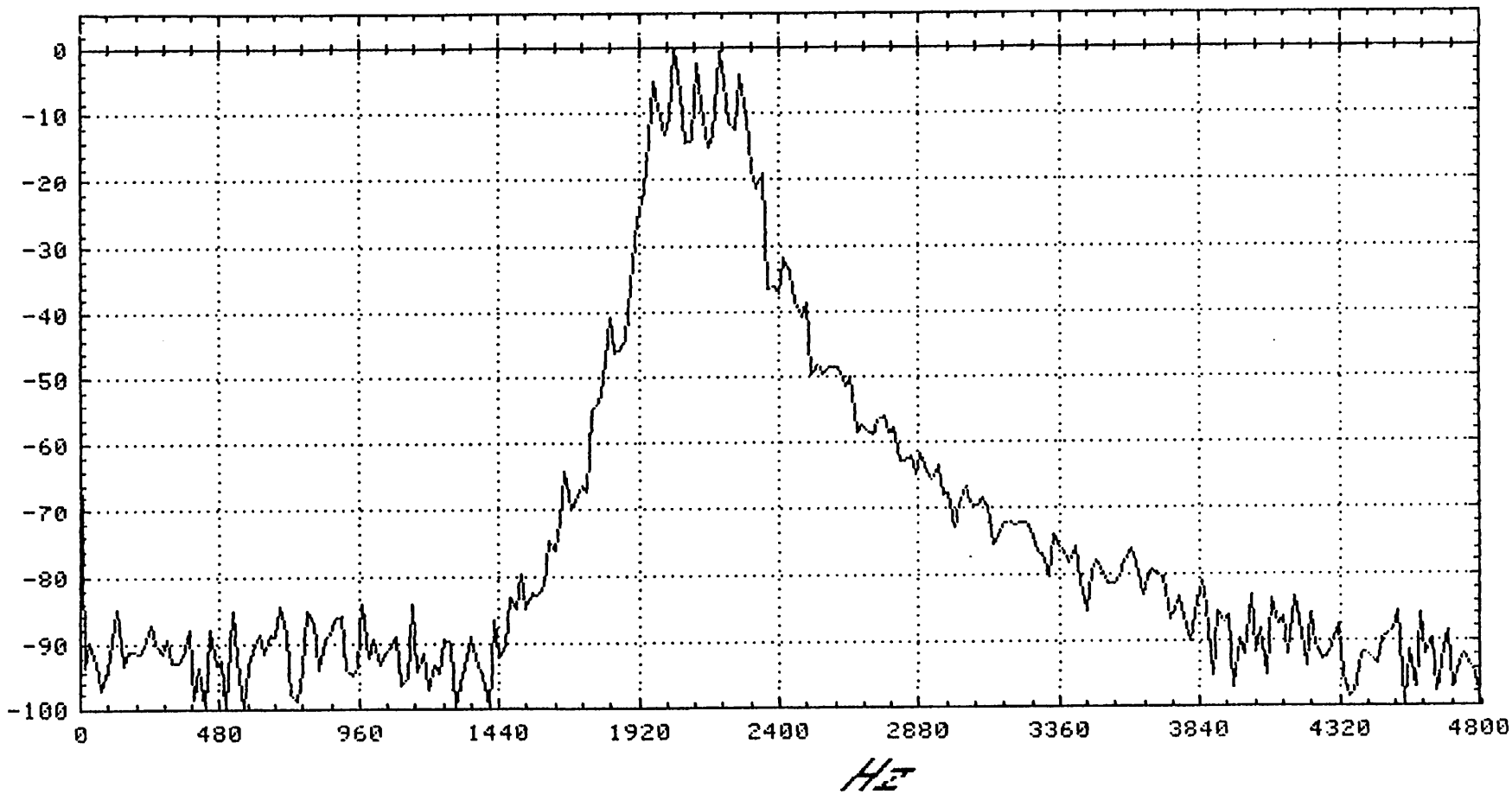


Figure 12. Toggled data after highband filter

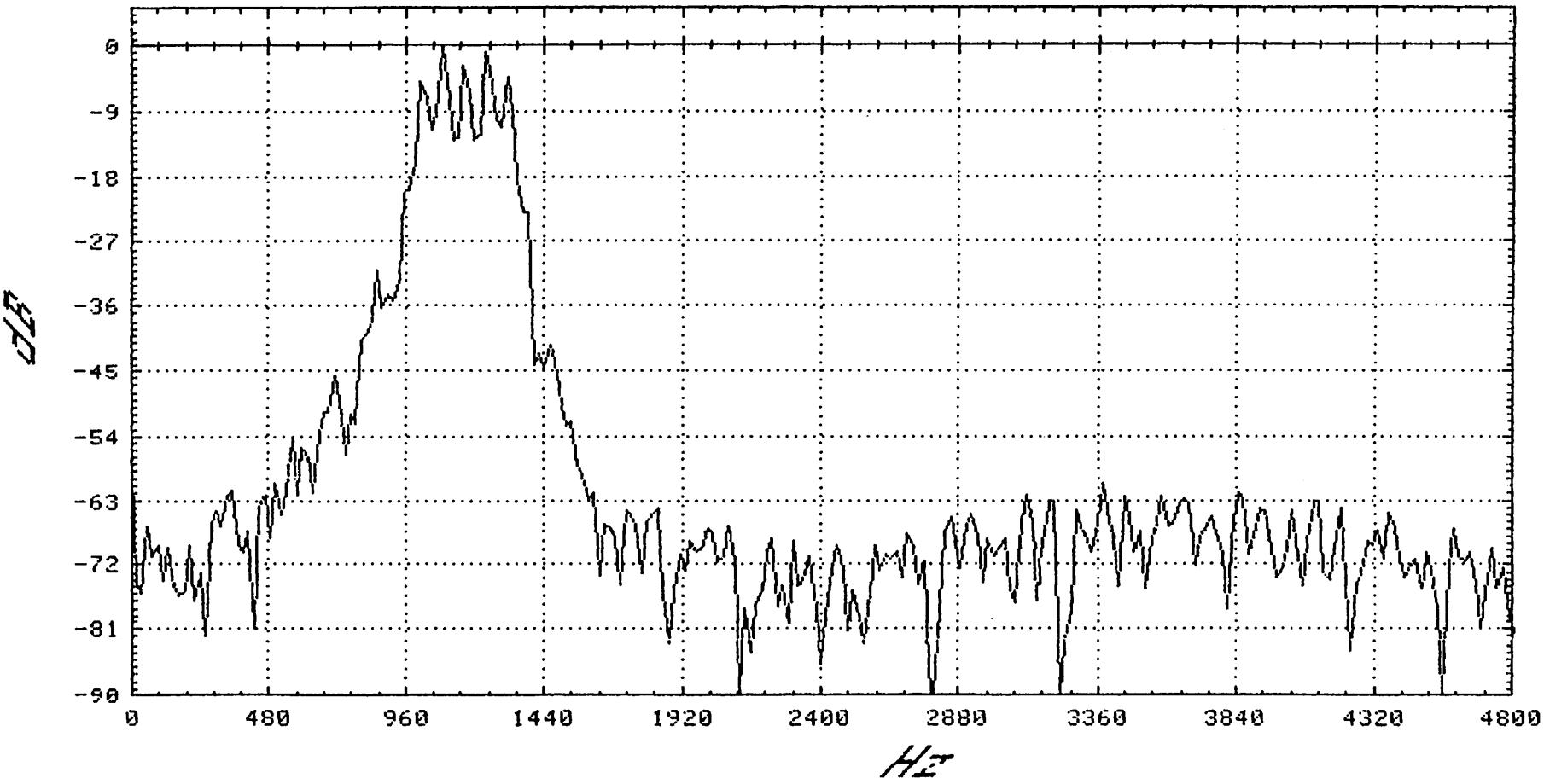
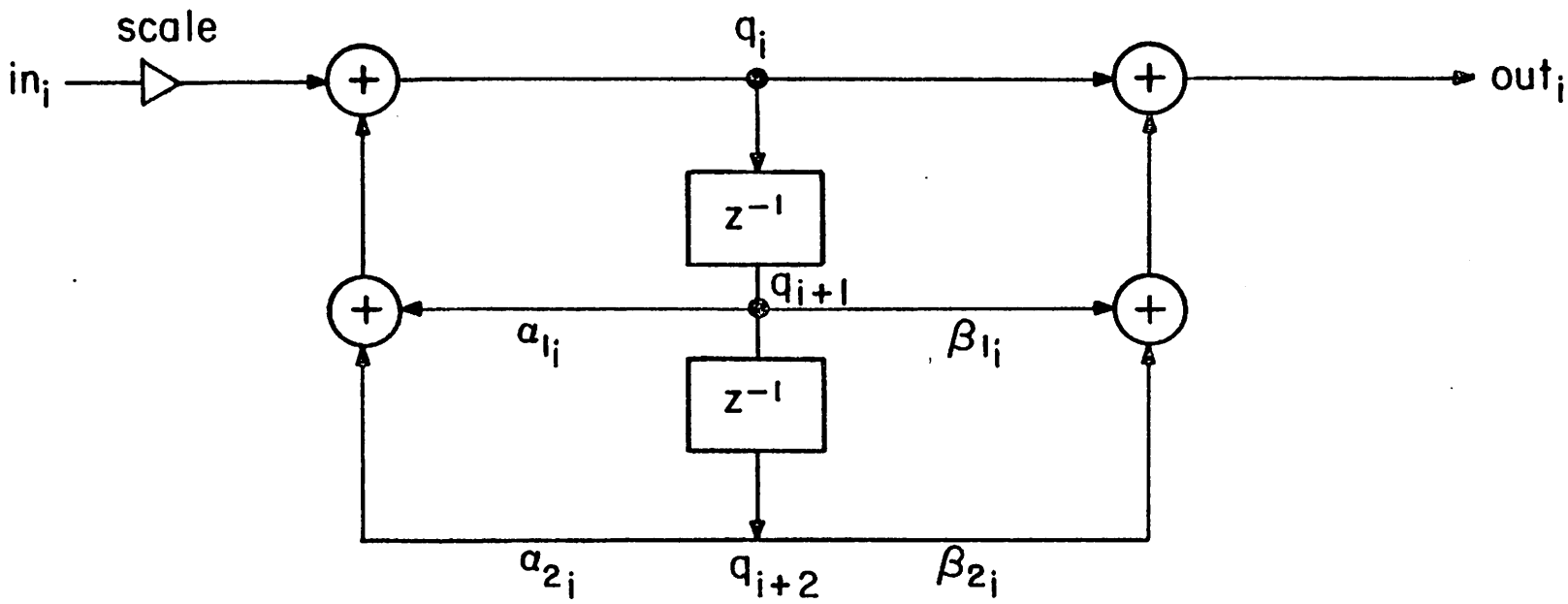


Figure 13. Toggled data after lowband filter



$$H(z) = \frac{out_i(z)}{in_i(z)} = \frac{1 + \beta_{1_i} z^{-1} + \beta_{2_i} z^{-2}}{1 - \alpha_{1_i} z^{-1} - \alpha_{2_i} z^{-2}}$$

Figure 14. Direct Form II second order section

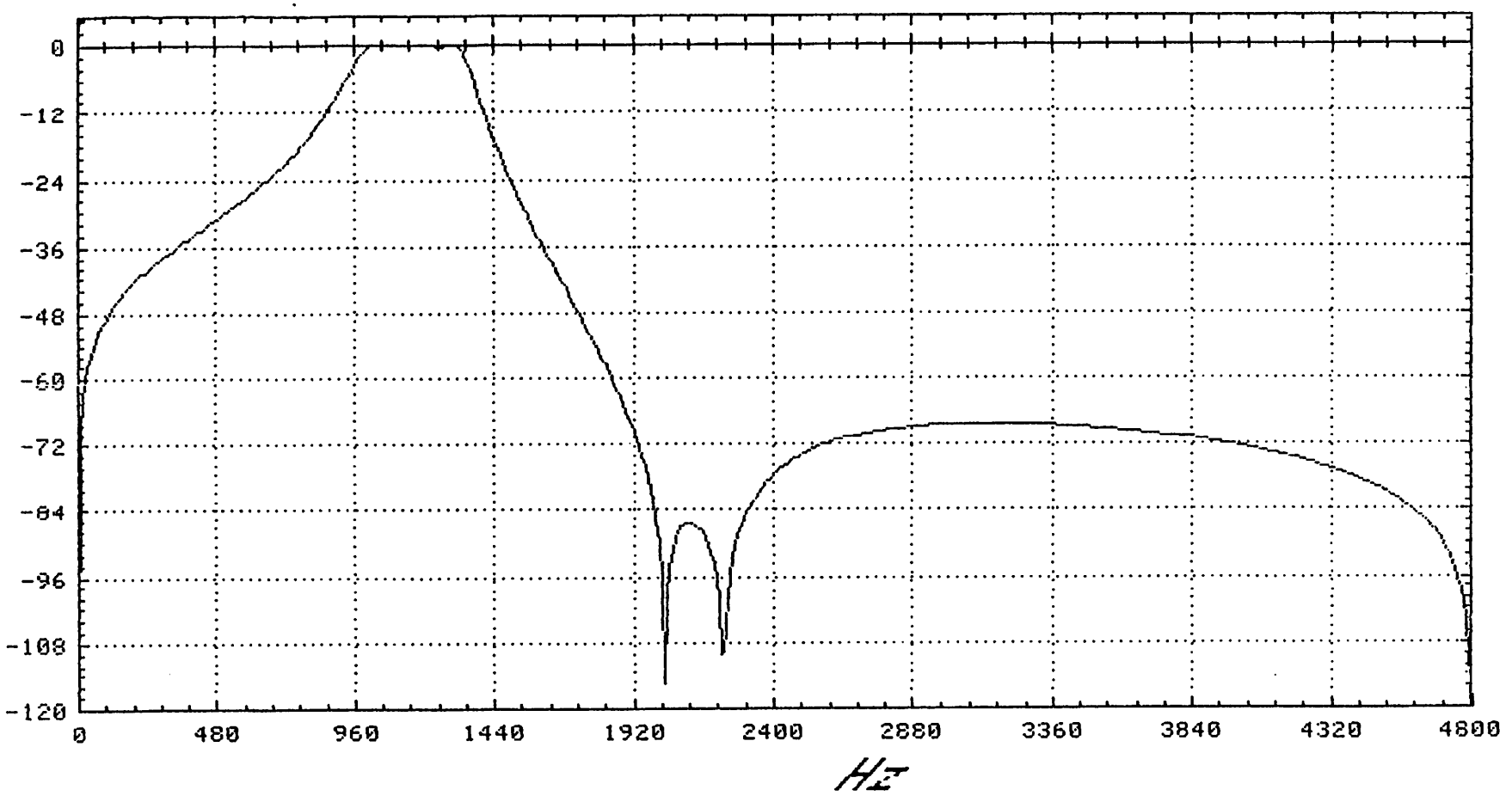


Figure 15. Lowband filter impulse response (32-bit wordlength)

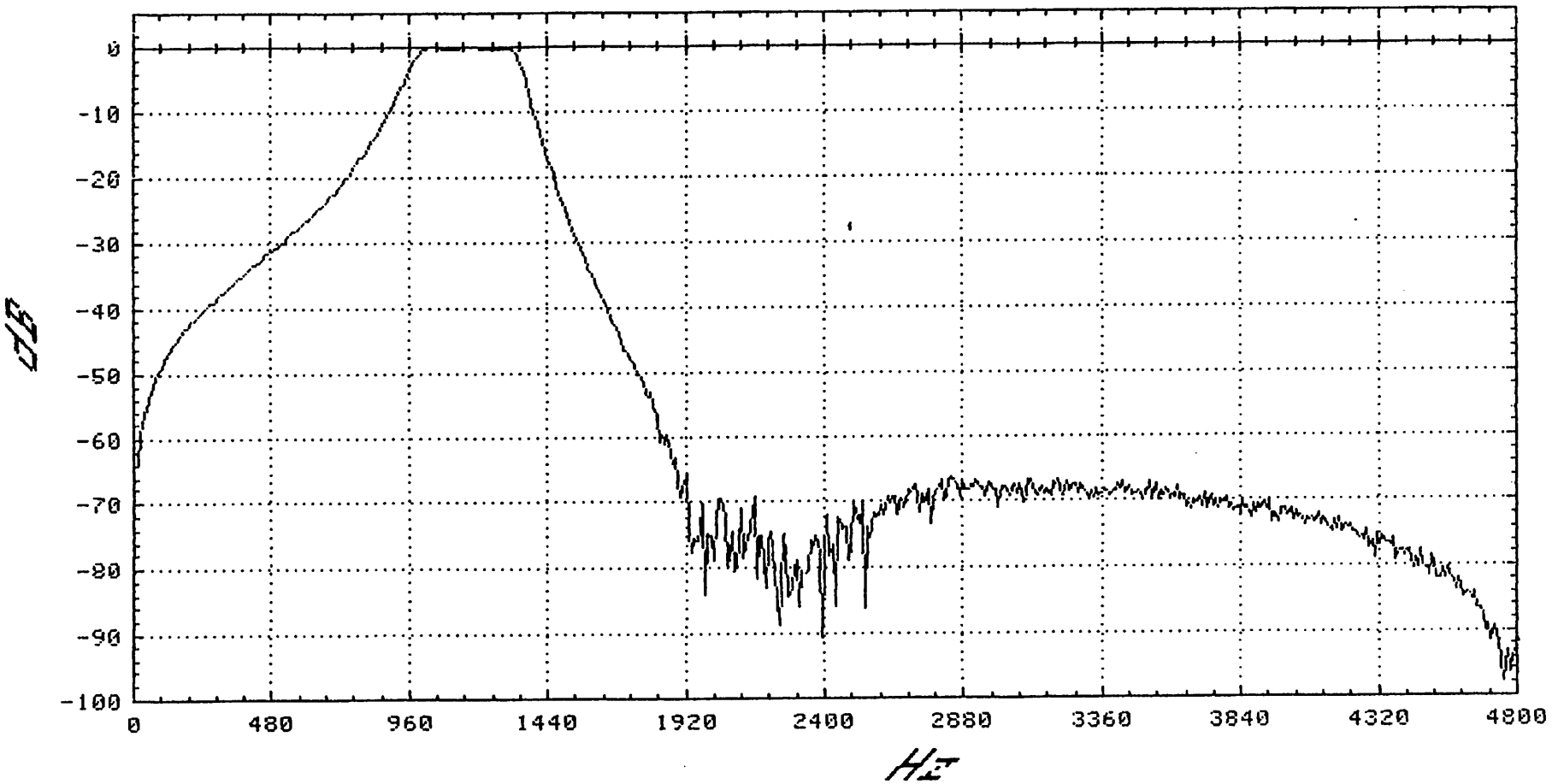


Figure 16. Lowband filter impulse response (20-bit wordlength)

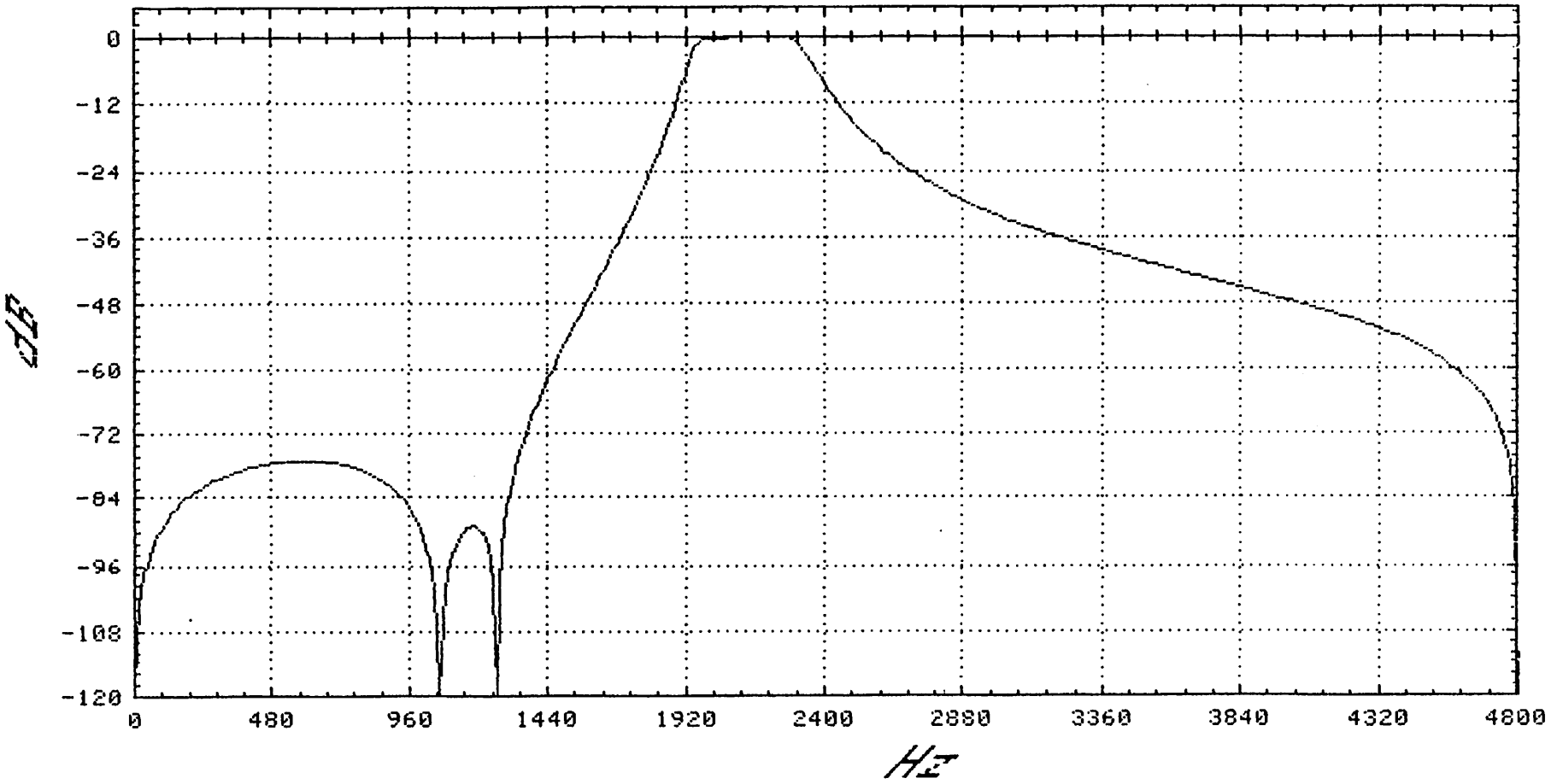


Figure 17. Highband filter impulse response (32-bit wordlength)

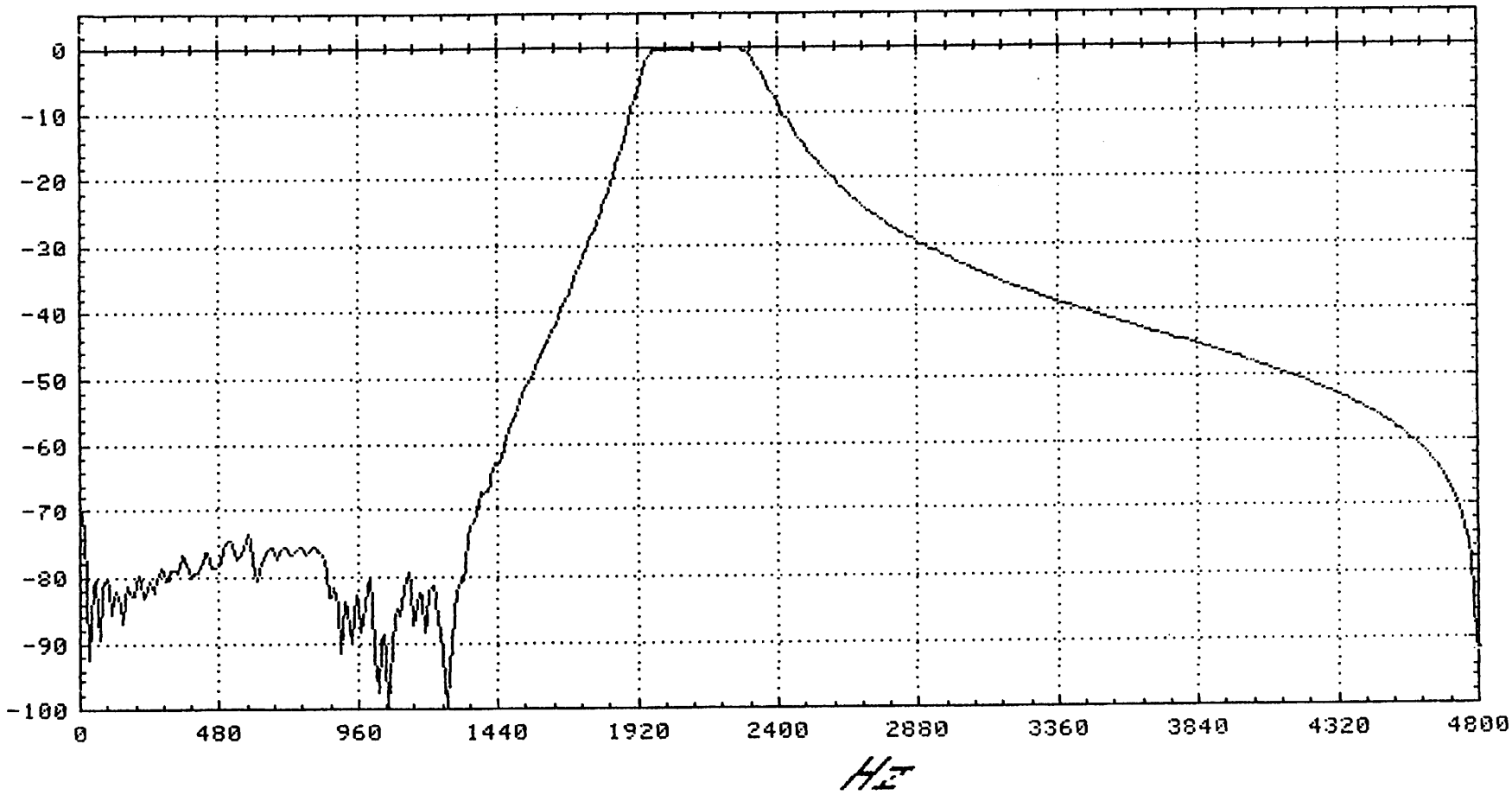
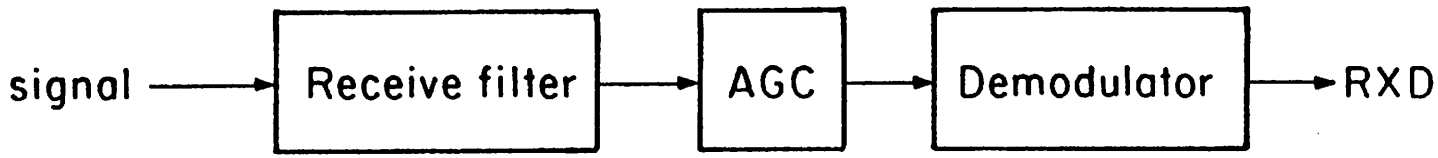
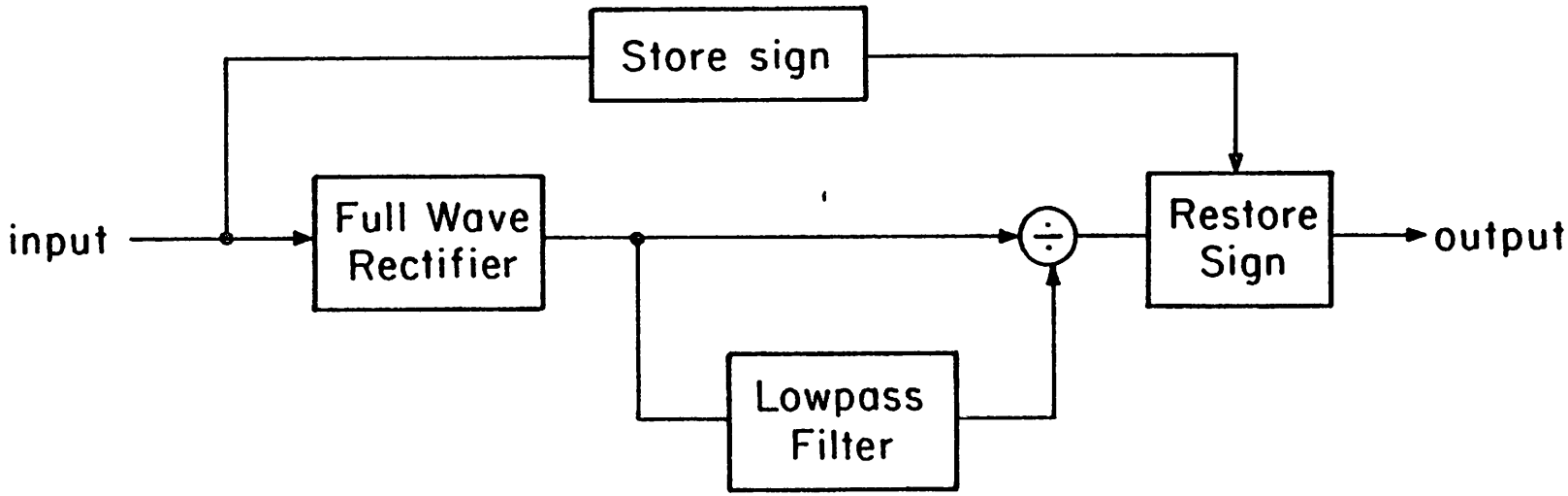


Figure 18. Highband filter impulse response (20-bit wordlength)



(a) Relation to Demodulator



(b) AGC Block Diagram

Figure 19. AGC function

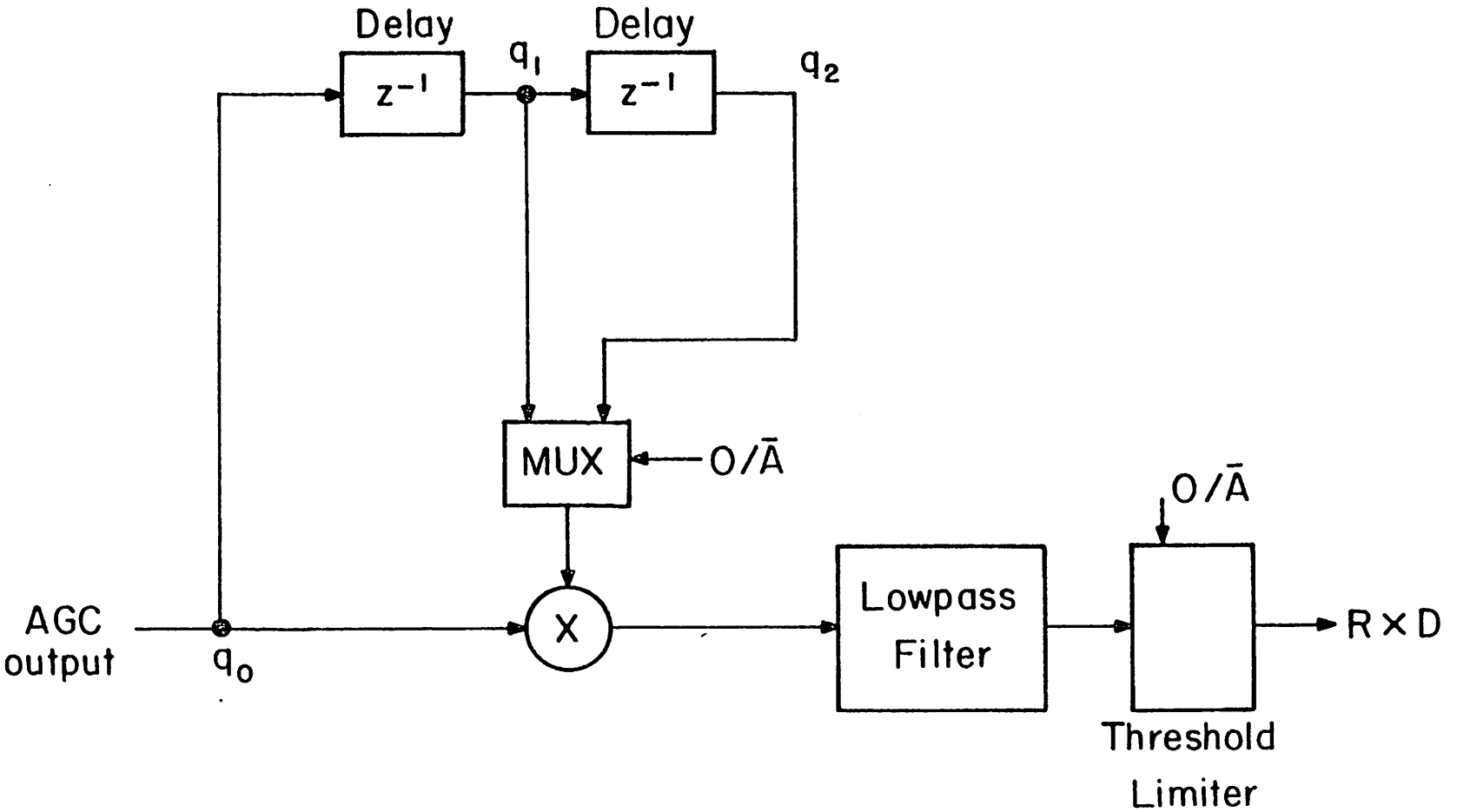


Figure 20. Delay line discriminator

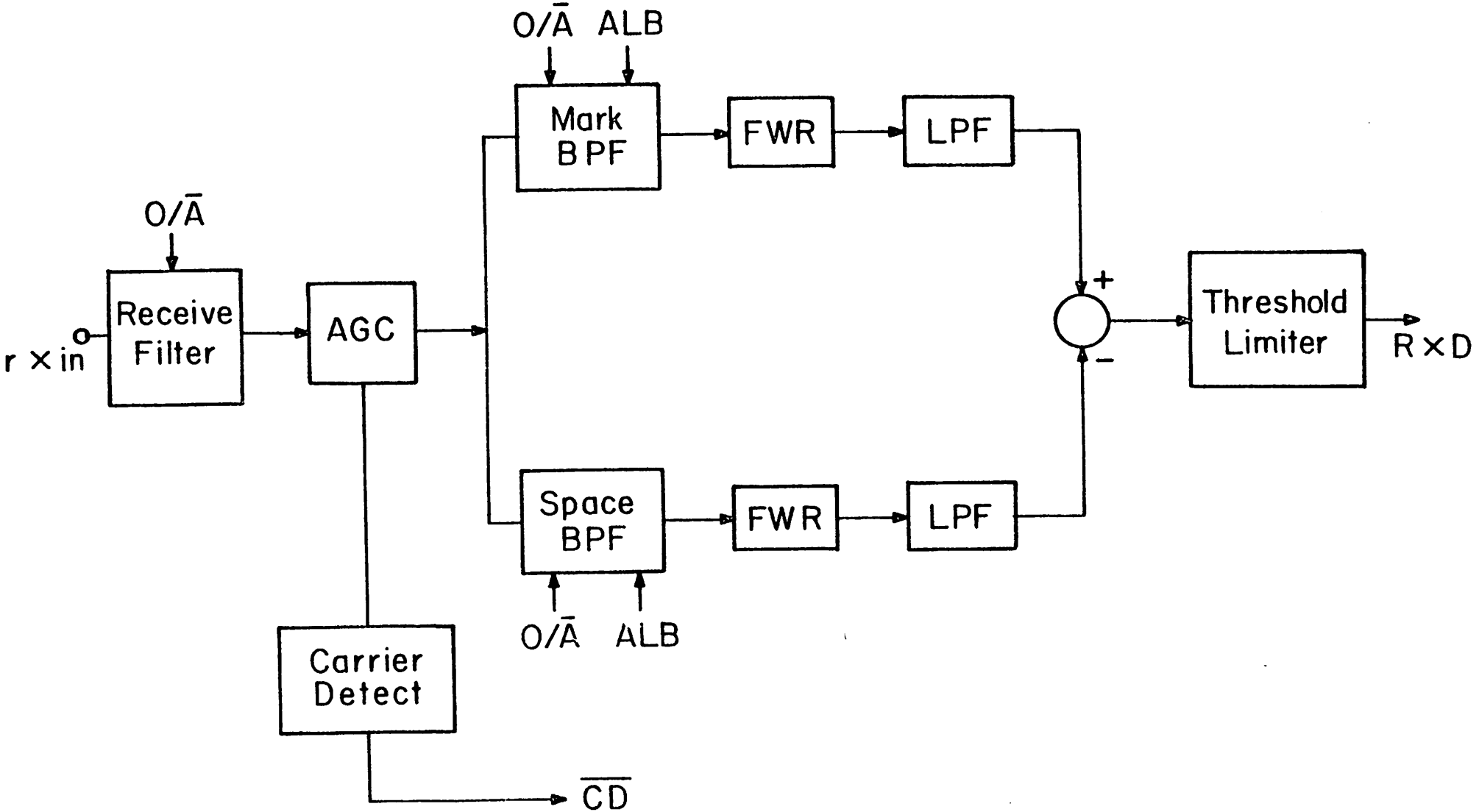


Figure 21. Bandpass filters demodulator

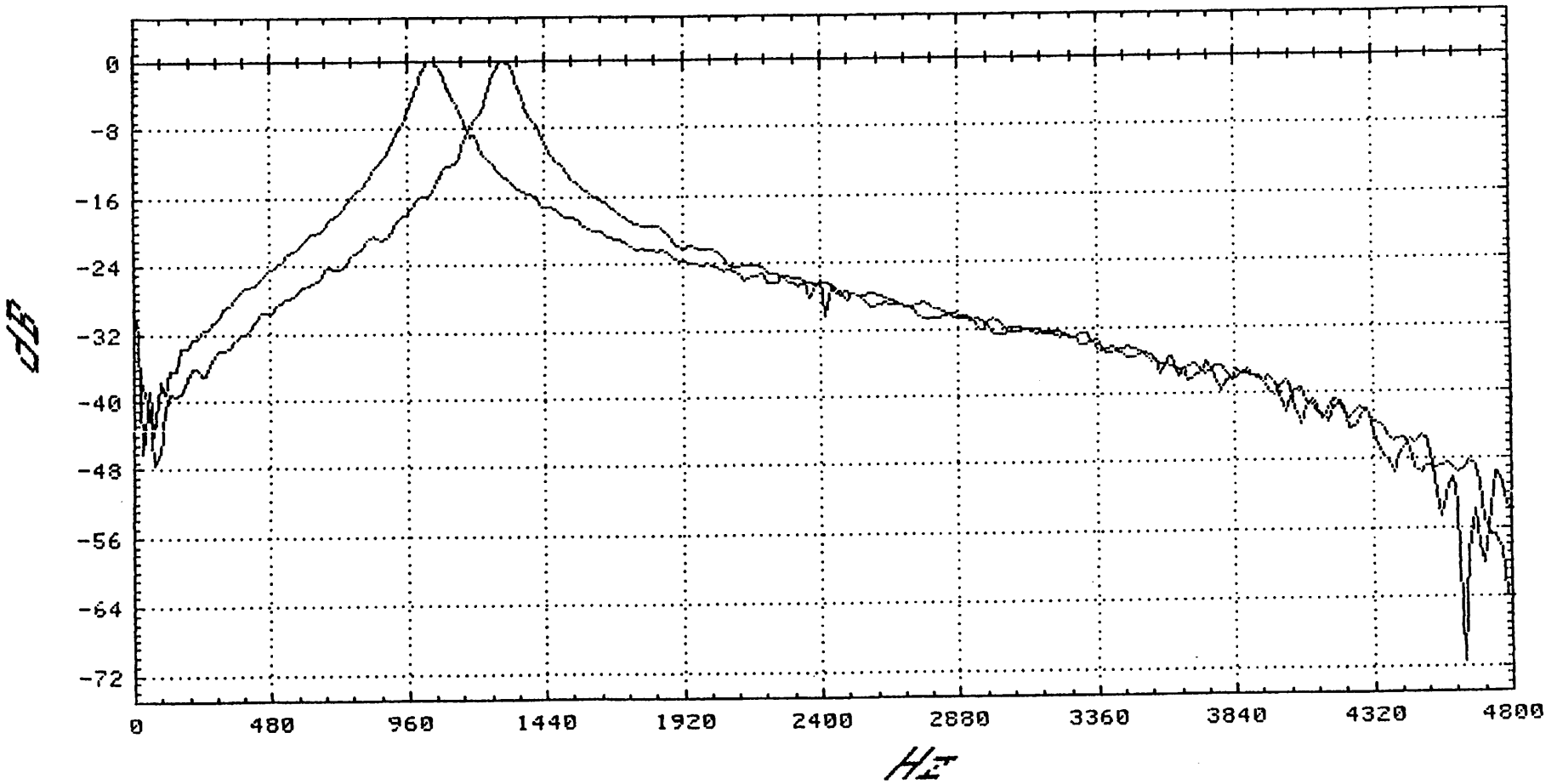


Figure 22. Demodulator answer mode bandpass filter impulse responses

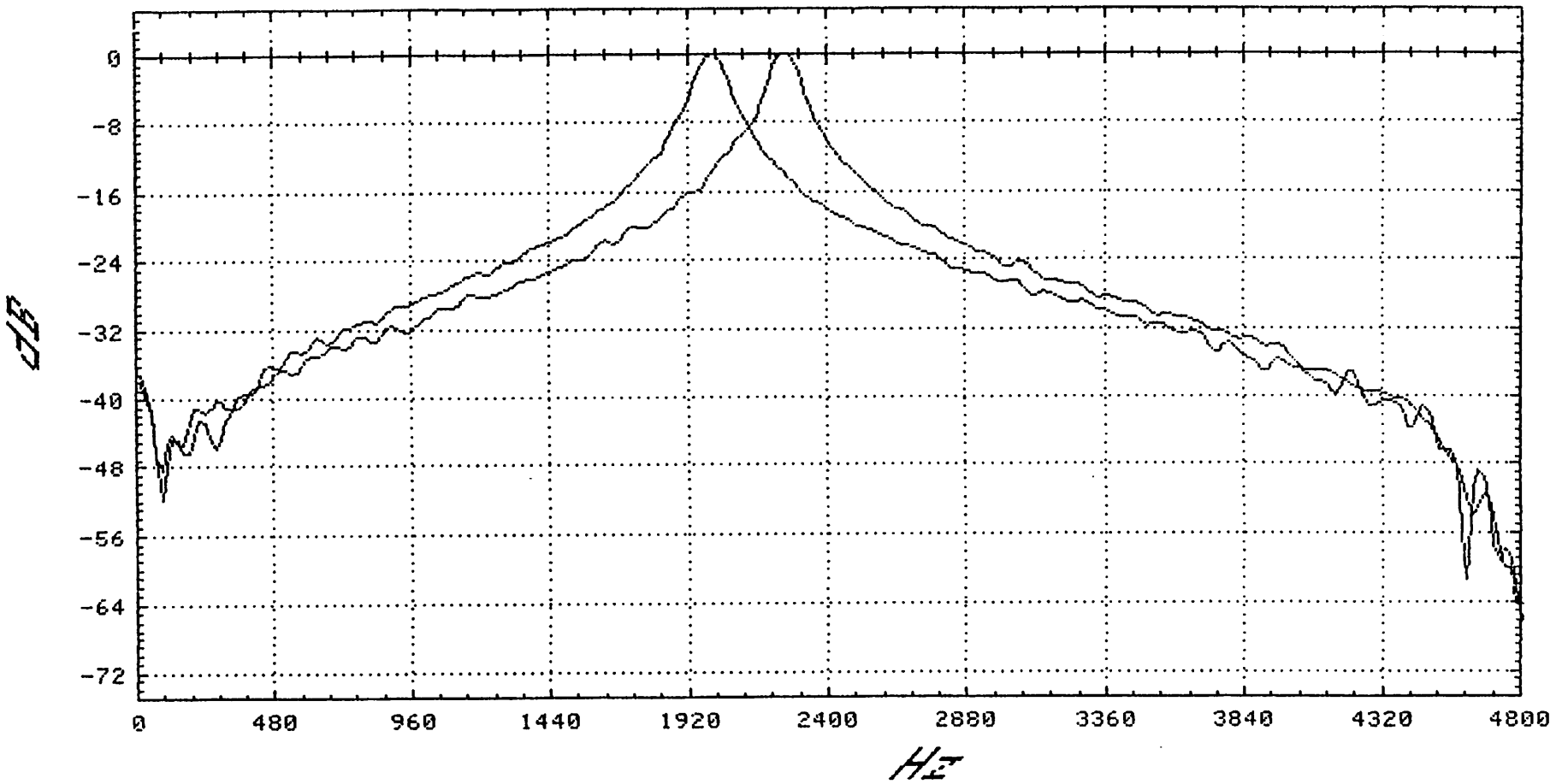


Figure 23. Demodulator originate mode bandpass filter impulse responses

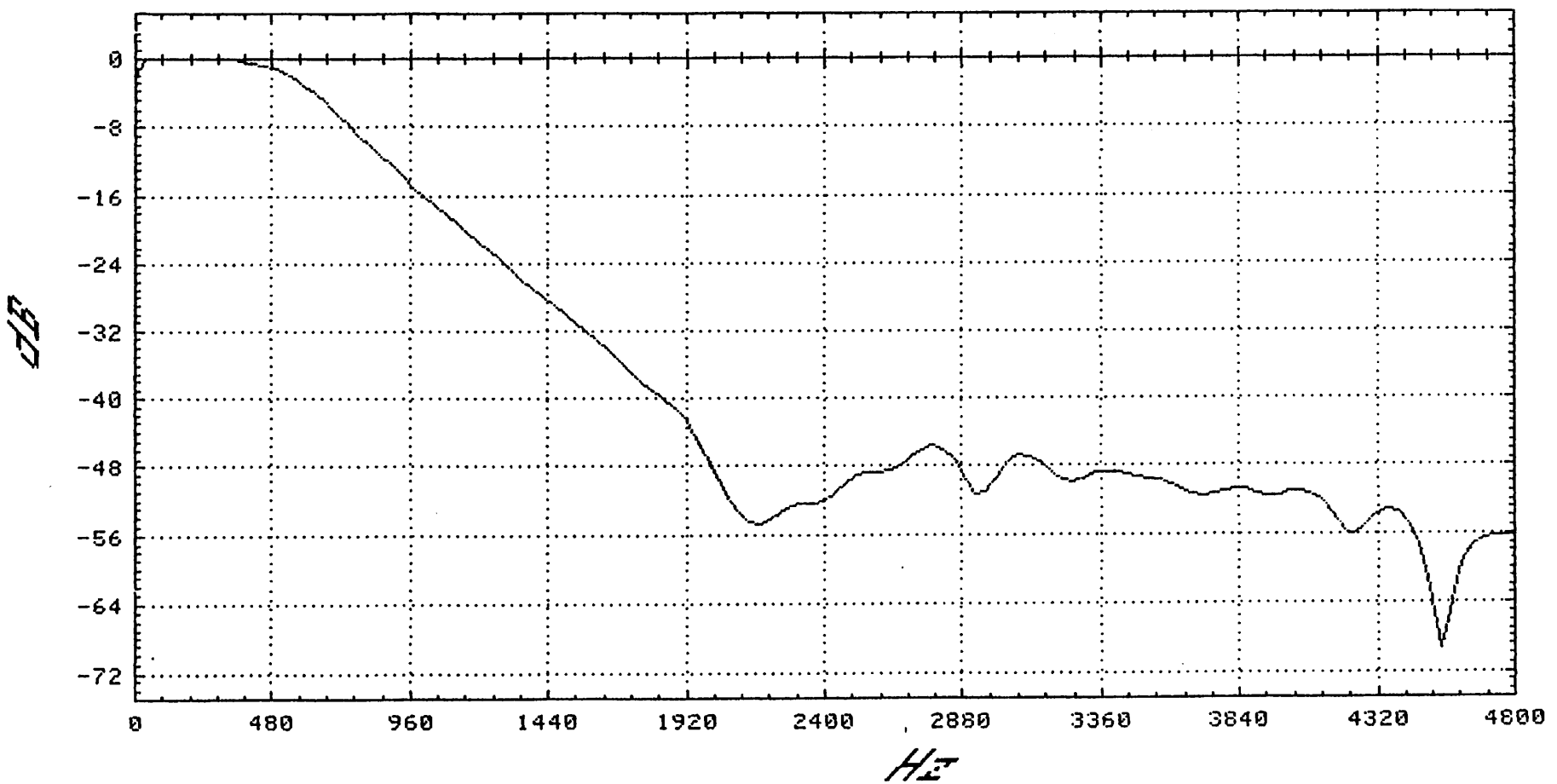


Figure 24. Demodulator lowpass filter impulse response

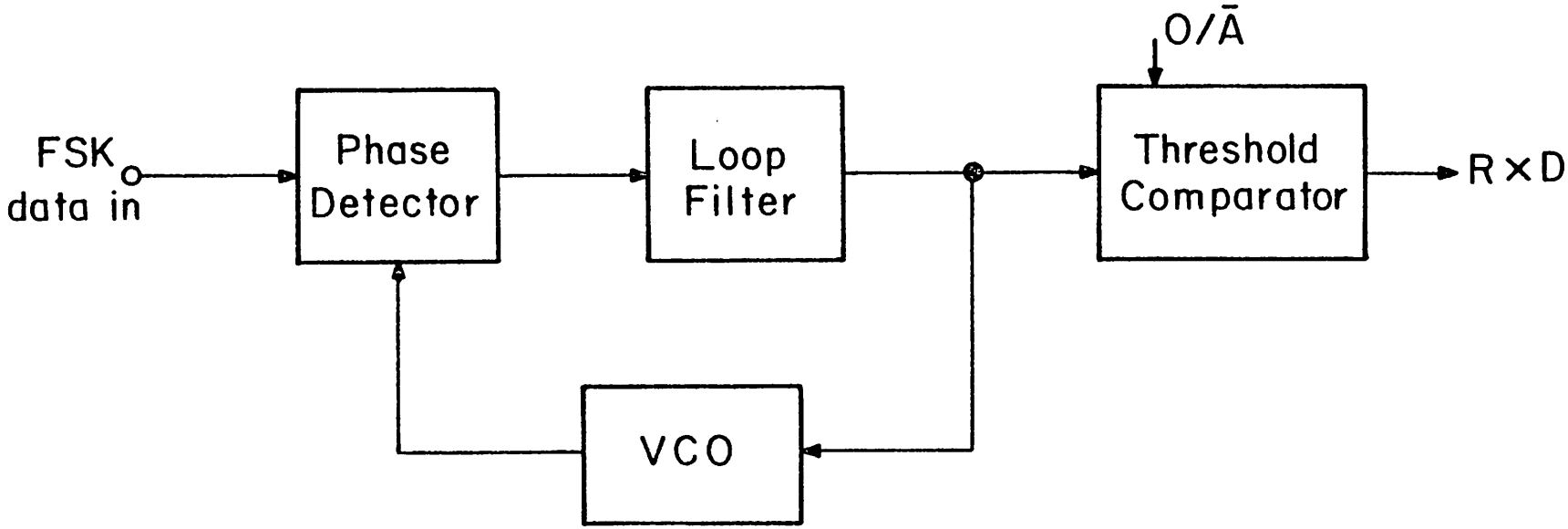


Figure 25. Demodulation using a digital phase locked loop

APPENDIX A
DESIGN FILE OF MODEM WITH TABLE LOOK-UP MODULATOR

```
/* 300 Baud FSK modem - Two processors - */
/* Processor modem contains the modulator and demodulator */
/* Processor filters contains two tenth order bandpass filters */
/* for filtering the transmit and receive signals */

.global
begin
rxin<12>;                /* FSK signal to be demodulated */
txout<12>;               /* FSK modulated data to be transmitted */
                         /*                                      */
wordin<12>;              /* wordin: |O/A|TXD|SQT|ALB|  */
                         /* O/A = mode of operation (A=0) */
                         /* TXD = digital data to be transmitted */
                         /* SQT = squelch modulator */
                         /* ALB = self-test mode, feeds modulated and */
                         /* filtered signal back to demodulator */
                         /*                                     */
wordout<12>;             /* wordout:|RXD|CD | - - -    */
                         /* RXD = demodulated digital data out */
                         /* CD=carrier detect signal(0=carrier present)*/
txmod<12>;               /* globals for interprocessor communication */
demod<12>;
cword<12>;
wd_out<12>;
tmp1<14>;                /* for divide operation in AGC circuit */
tmp2<14>;                /* for multiply operation in demodulator */
end

.io
begin
recin: rxin : signal_in;
ctrlin: wordin : signal_in;
transout: txout : signal_out;
dataout: wordout : signal_out;
end

.processor : modem<14>

begin

.local
begin
  temp;
  d[17];     /* storage for demodulator bandpass and lowpass filters */
  cmp[2];    /* cmp = demodulated signal power at space frequency */
             /* cmp[1] = demodulated signal power at mark frequency */
  p[5];      /* storage for lowpass filter in AGC circuit */
  step;      /* Index for table look-up. Step is incremented or */
             /* decremented according to desired frequency and part */
             /* of sine wave cycle */
  sine;      /* value from table */
  a_count;   /* counter to provide step adjustment every fourth count */
  o_count;   /* counter to provide step adjustment every tenth count */
  lc;        /* AGC output */
  alcin;     /* AGC input */
  lpfin;     /* input to lowpass filter in AGC circuit */
  lpfout;    /* output of lowpass filter in AGC circuit */
  cd;        /* carrier detect variables */
  count;
  c_past;
  outdata;   /* data to be loaded to wordout */
end

.constant
begin
```

```
  /* values in sinetable are equally spaced angles between */
  /* 0 and 90 degrees */

  sinetable[24] = 0,559,1115,1667,2210,2743,3263,3768,4256,
                  4724,5169,5591,5986,6354,6692,6999,7273,
                  7513,7718,7887,8020,8115,8172,8191;

  /* constants for correcting 1's complement subtractions */

  one=1;
  two=2;
  three=3;

  /* demodulator bandpass filter coefficients */

  ocoef[2]=4096, 1536;
  acoef[2]=4096, 2304;

  /* constants for computing step size */

  seven=7;
  eight=8;
  ten=10;
  m_four=-4;      /* for a_count test */
  m_ten=-10;      /* for o_count test */
  m_f_six=-46;
  m_test=-24;     /* for computing whether step has gone beyond */
                  /* table range */
  maxind=25;      /* for pivoting step around top to table */
  mask1=7168;     /* to get TXD value */
  mask2=3072;     /* to get SQT value */
  mask3=1024;     /* to get ALB value */
  odata=-8192;    /* for writing a "1" to RXD */

  /* Carrier detect constants */

  st1=43;         /* 20 msec on delay */
  st2=-128;       /* 10 msec off delay */
  th1=-33;        /* for -48 dB threshold */
  th2=-25;        /* for -43 dB threshold */
  max=8191;       /* for limiter */
  min=-8191;
end

.fsm
begin
  MSET: cc=!sign;
  SET:  cc=sign;
  MD: md=!sign;                          /* true if originate mode */
  MS: ms=!sign;                          /* true if space */
  ORIG: cc=((!md)&lb)|(md&(!lb));        /* for control of demodulator */
  ANSW: cc=((!md)&(!lb))|(md&lb);        /* bandpass filter coefficients */
  ALB: lb=sign;                          /* true for ALB set */
  SQT: sqt=sign;                         /* true if SQT set */
  SQTSET: cc=sqt;
  OS: os=md&ms;           /* true if originate and space */
  OM: om=md&(!ms);        /* etc. */
  AS: as=(!md)&ms;        /* true if answer and space */
  AM: am=(!md)&(!ms);     /* etc. */
  D1: d=cc|d;             /* for control of direction of stepping */
  U1: u=!cc&u;            /* through the table */
  D2: d=!cc&d;
  U2: u=cc|u;
```

```
TG: u=!d;                    /* makes sure only u or d is true */
TEST1: cc=(!sign)&u;         /* for testing whether step is > 23 */
TEST2: cc=sign&d;            /* for testing whether step is < 0 */
PN: pn=(!cc&pn)|(cc&!pn);    /* makes sine values negative for half */
PNSET: cc=pn;                /* a cycle */
OC: oc=cc;                   /* o_count flag */
AC: ac=cc;                   /* a_count flag */

/* The following states S1 to S16 are used in the modulator */
/* main_pr to set step to the correct value */

S1: cc=oc&os&u;
S2: cc=oc&om&u;
S3: cc=!oc&os&u;
S4: cc=!oc&om&u;
S5: cc=!ac&as&u;
S6: cc=ac&as&u;
S7: cc=!ac&am&u;
S8: cc=ac&am&u;
S9: cc=ac&am&d;
S10: cc=!ac&am&d;
S11: cc=ac&as&d;
S12: cc=!ac&as&d;
S13: cc=!oc&om&d;
S14: cc=!oc&os&d;
S15: cc=oc&om&d;
S16: cc=oc&os&d;

/* Carrier detect states */

CP: scp=sign;                          /* sign of c_past */
CT: sc=sign;                           /* sign of count */
XR: sch=((!scp)&(sc))|((scp)&(!sc));   /* EX-OR of scp and sc */
LM1: cc=sch&!sc;                       /* for limiter operation */
LM2: cc=sch&sc;
end




.main_pr<*>

/* MODULATOR SEGMENT BEGINS HERE */

begin
/* get MD, MS, ALB, SQT values */

mbus=cword, le, mor:=~mir;
acc:=mor, r(mask1), TG;
acc:=acc&mor, sor:=mor>1, MD;
acc:=acc+~sor, r(one);
acc:=acc+sor, sor:=mor, MS;
acc:=acc+sor, r(mask2);
acc:=acc&mor, sor:=mor>1;
acc:=acc+~sor, r(one);
acc:=acc+sor, sor:=mor, SQT;
acc:=acc+sor, r(mask3);
acc:=acc&mor, sor:=mor>1, r(step);

/* load step into iy to index table */

acc:=acc+~sor, mbus:=mor, iy:=mbus, sor:=mor, r(eight);
acc:=sor, sor:=mor, ALB;
```

```
acc:=acc+sor,  OS;
ry(sinetable), S1;                          /* step:=step+8 (1070 Hz) */
le, wc(step), acc:=mor, sor:=mor, PNSET;
r(one);
acc:=mor+~sor, le, w(sine), OM;
le, wc(sine), acc:=0, SQTSET;
le, wc(sine);                               /* sine:=0 if squelch set */
r(sine),       AS;
mbus=mor, txmod:=mbus, r(step), AM;  /* output sine value */

/* compute step for counting up through table */

sor:=mor, r(ten),S2;
acc:=sor+mor, r(one);
le, wc(step), sor:=mor;                     /* step:=step+10 (1270 Hz) */
acc:=acc+sor, r(two), S3;
le, wc(step), sor:=mor;                     /* step:=step+11 (1070 Hz) */
acc:=acc+sor, r(seven), S4;
le, wc(step), sor:=mor;                     /* step:=step+13 (1270 Hz) */
acc:=acc+sor, r(one), S5;
le, wc(step), sor:=mor;                     /* step:=step+20 (2025 Hz) */
acc:=acc+sor, r(one), S6;
le, wc(step), sor:=mor;                     /* step:=step+21 (2025 Hz) */
acc:=acc+sor, r(one), S7;
le, wc(step), sor:=mor;                     /* step:=step+22 (2225 Hz) */
acc:=acc+sor, r(m_f_six), S8;
le, wc(step), sor:=mor;                     /* step:=step+23 (2225 Hz) */
acc:=acc+sor, r(one), S9;

/* compute step for counting down through table */

le, wc(step), sor:=mor;                     /* step:=step-23 (2225 Hz) */
acc:=acc+sor, r(one), S10;
le, wc(step), sor:=mor;                     /* etc. */
acc:=acc+sor, r(one), S11;
le, wc(step), sor:=mor;
acc:=acc+sor, r(seven),  S12;
le, wc(step), sor:=mor;
acc:=acc+sor, r(two), S13;
le, wc(step), sor:=mor;
acc:=acc+sor, r(one), S14;
le, wc(step), sor:=mor;
acc:=acc+sor, r(two), S15;
le, wc(step), sor:=mor;
acc:=acc+sor, r(one), S16;
le, wc(step), sor:=mor, acc:=0;
r(o_count), SET;
acc:=sor+mor, r(m_ten), OC;        /* reset oc, ac */
le, w(o_count), sor:=mor, AC;
acc:=acc+sor, r(one);
acc:=0, sor:=mor, r(a_count), MSET;
le, wc(o_count), acc:=sor+mor, OC; /* set oc true every 10th count*/
r(m_four);
le, w(a_count), sor:=mor;
acc:=acc+sor, r(m_test);
acc:=0, sor:=mor, r(step), MSET;
le, wc(a_count), acc:=sor+mor, AC; /* set ac true every 4th count */
r(one), TEST1;
r(maxind), sor:=mor, D1;
acc:=acc+sor, sor:=mor, U1;
le, mor:=~mir;
acc:=mor+sor,  r(step);
```

```
le, wc(step), acc:=mor;          /* step:=25 - (step+(-23)) */
le, mor:=~mir, TEST2;
acc:=mor, U2;
le, wc(step), D2;                /* step:= -step */

/* DEMODULATOR SEGMENT BEGINS HERE */

/* Automatic Gain Control (AGC) */

mbus=demod, le, w(alcin), PN;    /* store input to remember sign */
r(alcin);
sor:=mor;
acc:=|sor|;                              /* FWR of alcin */
le, w(lpfin);                            /* store lpf input */

/* 3rd order lowpass filter for envelope detect */

r(lpfin);
sor:=mor>4;
acc:=sor, sor:=sor>1, r(p[2]);
acc:=acc+sor, sor:=mor>1;
acc:=acc+~sor, sor:=sor>2;
acc:=acc+~sor, sor:=sor>1, r(three);
acc:=acc+~sor, sor:=mor, r(p[1]);
acc:=acc+sor, sor:=mor;
acc:=acc+sor, sor:=sor>1;
acc:=acc+sor, sor:=sor>3, r(p[1]);
acc:=acc+sor, sor:=mor>2, r(one);
le, w(p), acc:=acc+~sor, sor:=mor;
acc:=acc+sor, r(p[2]);
sor:=mor, r(p[4]);
acc:=acc+sor, sor:=mor>1;
le, w(temp), acc:=sor, sor:=sor>2;
r(temp), acc:=acc+sor, sor:=sor>2;
sor:=mor>3, acc:=acc+sor, r(p[4]);
acc:=acc+sor, sor:=mor;
le, w(p[3]), acc:=acc+sor;
le, w(temp);
r(temp);                         /* scale output by 1.625 */
acc:=mor, sor:=mor>1;            /* so that denominator of divide */
acc:=acc+sor, sor:=sor>2;        /* opeation is greater than numerator */
acc:=acc+sor, r(one);
le, w(lpfout), sor:=mor;             /* store lpf output */
acc:=sor+mor, r(lpfin);
le, w(temp), sor:=mor;
r(temp);                                /* temp has -lpfout */
acc:=|sor|, sor:=mor>1;                 /* implement lpfin/lpfout */
sor:=sor>1, acc:=sor+acc, aip;          /* to normalize received signals*/
sor:=sor>1, acc:=sor+acc, aip;
sor:=sor>1, acc:=sor+acc, aip;
sor:=sor>1, acc:=sor+acc, aip;
sor:=sor>1, acc:=sor+acc, aip;
sor:=sor>1, acc:=sor+acc, aip; /* update lowpass filter variables */
sor:=sor>1, acc:=sor+acc, aip, r(p[1]);
sor:=sor>1, acc:=sor+acc, aip, mbus=mor, le, w(p[2]);
sor:=sor>1, acc:=sor+acc, aip, r(p);
sor:=sor>1, acc:=sor+acc, aip, mbus=mor, le, w(p[1]);
sor:=sor>1, acc:=sor+acc, aip, r(p[3]);
sor:=sor>1, acc:=sor+acc, aip, mbus=mor, le, w(p[4]);
acc:=sor+acc, aip;
tmp1:=quot, r(one);
mbus=tmp1, le, w(lc), sor:=mor;          /* lc has tmp=lpfin/lpfout */
```

```
acc:=sor+mor, r(alcin);                    /* ccc := -tmp */
le, acc:=mor;
SET;                                       /* recall sign of input to alc */
wc(lc);                                    /* lc := -tmp if cc set */
r(lc);

/* BEGIN CARRIER DETECT FUNCTION */


acc:=mor, r(cd);
le, w(d), sor:=mor;  /* write output of AGC to demodulator */
acc:=sor, r(lpfout); /* bandpass filter inputs */
r(th1), sor:=mor, SET;       /* sor:= lpfout , cc=sign of cd */
acc:=sor+mor, r(th2);        /* acc:= lpfout - 48 dB threshold */
le, w(cd), sor:=mor;
acc:=acc+sor, r(count);      /* acc:= lpfout - 43 dB threshold */
le, wc(cd), sor:=mor;        /* use 43 dB threshold if cd was <0 */
r(st1);
acc:=sor+mor, r(st2);        /* acc:= count + 43 for turn on delay*/
le, w(count), sor:=mor;
acc:=acc+sor, r(c_past);     /* acc:= count-128 for turn off delay*/
le, wc(count), acc:=mor;
r(count), CP;
acc:=mor;
CT;
r(max), XR;
acc:=mor, LM1, r(min);                     /* limit count if there */
le, wc(count), acc:=mor, LM2;              /* was a sign change between */
le, wc(count);                             /* count and c_past */
r(count);
mbus=mor, le, w(c_past), acc:=mor;
acc:=0, SET, r(ocoef);                     /* cc= sign of count */
le, w(outdata), acc:=mor;                  /* CD = 0 if carrier present */
le, wc(outdata);


/* Demodulator envelope detectors comparison */

r(cmp);
sor:=mor, r(cmp[1]);
acc:=mor+~sor, r(one);     /* cmp[1]-cmp > 0 : data=mark */
sor:=mor, r(outdata);      /* cmp[1]-cmp < 0 : data=space */
acc:=acc+sor, sor:=mor, r(odata);
acc:=sor+mor, MSET;
le, wc(outdata);
r(outdata);
mbus=mor, wd_out:=mbus;
end


/* Implements demodulator bandpass and lowpass filters. */
/* First iteration does space filtering in either O or A mode. */
/* Second iteration does mark filtering in either O or A mode. */

.sub_pr <2>
  begin

  /* implement 2nd order bandpass filters */

  rx(acoef), ORIG;
  mbus=mor, le, w(temp);
  rx(ocoef);
  mbus=mor, le, wc(temp);
  r(temp);
  rx(d[3]), mbus=mor, tmp2:=mbus;
```

```
    sor:=mor;
    sor:=scr>1, acc:=coef.~sor, coef:=tmp2;
    sor:=scr>1, acc:=acc+coef.sor;
    sor:=scr>1, acc:=acc+coef.sor;
    sor:=sor>1, acc:=acc+coef.sor;
    sor:=scr>1, acc:=acc+coef.sor;
    sor:=scr>1, acc:=acc+coef.sor;
    sor:=scr>1, acc:=acc+coef.sor;
    sor:=scr>1, acc:=acc+coef.sor;
    sor:=scr>1, acc:=acc+coef.sor;
    sor:=scr>1, acc:=acc+coef.sor,rx(d[9]);
    sor:=scr>1, acc:=acc+coef.sor,mbus=mor, le, wx(d[11]);
    sor:=scr>1, acc:=acc+coef.sor,rx(d[7]);
    sor:=scr>1, acc:=acc+coef.sor,mbus=mor, le, wx(d[9]);
    acc:=acc+coef.sor, rx(d[3]);
    sor:=mor;
    le, w(temp), acc:=acc+sor, ANSW;
    le, wc(temp);
    r(temp);
    sor:=mor, rx(d[5]);
    acc:=sor, sor:=mor;
    acc:=acc+~sor, sor:=sor>4, r(d);
    acc:=acc+sor, sor:=mor>6;
    acc:=acc+sor, sor:=sor>6, r(one);
    acc:=acc+sor, sor:=mor, rx(d[5]);
    acc:=acc+sor, sor:=mor, r(one);
    le, wx(d[1]), acc:=acc+~sor, sor:=mor;
    acc:=acc+sor;
    le, w(temp);


    r(temp);
    sor:=mor<1, rx(d[13]);
    acc:=|sor|, mbus=mor, le, wx(d[15]);  /* full wave rectify (FWR) */
    le, w(temp);                          /* of bandpass filter output*/

    /* 3rd order lowpass filter to remove components around 2fc */

    r(temp);
    sor:=mor>4;
    acc:=sor, sor:=sor>1, rx(d[11]);
    acc:=acc+sor, sor:=mor>1, rx(d[3]);
    acc:=acc+~sor, sor:=sor>2, mbus=mor, le, wx(d[5]);
    acc:=acc+~sor, sor:=sor>1, r(three);
    acc:=acc+~sor, sor:=mor, rx(d[9]);
    acc:=acc+sor, sor:=mor, rx(d[1]);
    acc:=acc+sor, sor:=sor>1, mbus=mor, le, wx(d[3]);
    acc:=acc+sor, sor:=sor>3, rx(d[9]);
    acc:=acc+sor, sor:=mor>2, r(one);
    le, wx(d[7]), acc:=acc+~sor, sor:=mor;
    r(temp), acc:=acc+sor, sor:=sor>2;
    sor:=mor>3, acc:=acc+sor, rx(d[15]);
    acc:=acc+sor, sor:=mor;
    le, wx(d[13]), acc:=acc+sor;
    le, wx(cmp);
    end
  end
```

```
/* PROCESSOR FILTERS IMPLEMENTS TWO TENTH ORDER BANDPASS FILTERS */

.processor : filters<20>

begin

.local
begin
  temp;
  lq[16];          /* storage for lowband filter */
  hq[16];          /* storage for highband filter */
  alb;             /* for self-test function */
  temp1;
  hresult;         /* highband filter output */
end

.constant
begin
  /* constants for correcting 1's complement subtractions */

  one=1;
  two=2;
  three=3;
  four=4;
  mask=65536;  /* to get ALB status from control word */
end

.fsm
begin
  SET: cc=sign;
  MSET: cc=!sign;
  MD: md=!sign;          /* true for originate */
  ORIG: cc=md;
  ANSW: cc=!md;
  ALB: lb=sign;          /* true if ALB set */
  ALBO: cc=lb&md;        /* states for controlling ALB function */
  ALBA: cc=lb&!md;
  ALBSET: cc=lb;
end


.main_pr

begin

/* get MD and ALB and perform signal multiplexing */

mbus=wordin, le, mor:=~mir, cword:=mbus; /* send wordin to processor*/
acc:=mor, r(mask);                       /* modem via cword */
acc:=acc&mor, sor:=mor>1, MD;
acc:=acc+~sor, ORIG;
mbus=txmod, le, wc(lq), ALB;    /* get txmod from processor modem */
ANSW;                           /* to filter it before transmitting */
wc(hq);
mbus=rxin, le, wc(lq), ORIG;    /* get signal to be filtered and */
```

```
wc(hq);                          /* demodulated */
mbus=wd_out, wordout:=mbus;      /* transfer processor modem variable */
                                 /* wd_out to wordout output */

/* LOWBAND FILTER */

/* First second order section - 2 poles, then 2 zeros */

r(lq);
sor:=mor>4, r(lq[2]);
acc:=sor, sor:=mor, r(lq[5]);
acc:=acc+sor, sor:=sor>2, mbus=mor, le, w(lq[6]);
acc:=acc+sor, sor:=sor>2, r(lq[3]);
acc:=acc+sor, sor:=mor;
acc:=acc+~sor, sor:=sor>3, r(lq[4]);
acc:=acc+sor, sor:=sor>2, mbus=mor, le, w(lq[5]);
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+~sor, sor:=mor, r(lq[2]);
acc:=acc+sor, sor:=mor;
le, w(lq[1]), acc:=acc+~sor, sor:=sor>1;
acc:=acc+~sor, sor:=sor>2;
acc:=acc+~sor, sor:=sor>3, r(four);
acc:=acc+~sor, sor:=mor, r(lq[3]);
acc:=acc+sor, sor:=mor, r(lq[8]);
acc:=acc+sor, sor:=sor>3, mbus=mor, le, w(lq[9]);
acc:=acc+sor, sor:=sor>2, r(lq[7]);
acc:=acc+sor, sor:=sor>1, mbus=mor, le, w(lq[8]);
acc:=acc+sor, r(lq[5]);

/* Second second order section */

le, w(temp), acc:=mor, sor:=mor>1;
r(temp), acc:=acc+sor, sor:=sor>6;
sor:=mor>2, acc:=acc+sor, r(lq[6]);
acc:=acc+sor, sor:=mor, r(lq[11]);
acc:=acc+~sor, sor:=sor>3, mbus=mor, le, w(lq[12]);
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+~sor, sor:=mor, r(lq[5]);
acc:=acc+sor, sor:=mor>1;
le, w(lq[4]), acc:=acc+~sor, sor:=sor>5;
acc:=acc+sor, r(one);
sor:=mor, r(lq[6]);
acc:=acc+sor, sor:=mor;
acc:=acc+sor, r(lq[8]);

/* Third second order section */

le, w(temp), acc:=mor, sor:=mor>2;
r(temp), acc:=acc+sor, sor:=sor>5;
sor:=mor>1, acc:=acc+~sor, r(lq[9]);
acc:=acc+sor, sor:=mor;
acc:=acc+~sor, sor:=sor>4, r(two);
acc:=acc+sor, sor:=mor, r(lq[8]);
acc:=acc+sor, sor:=mor;
le, w(lq[7]), acc:=acc+~sor, sor:=sor>1;
acc:=acc+~sor, sor:=sor>6, r(two);
acc:=acc+sor, sor:=mor, r(lq[9]);
acc:=acc+sor, sor:=mor, r(lq[10]);
acc:=acc+sor, sor:=sor>3, mbus=mor, le, w(lq[11]);
acc:=acc+sor, sor:=sor>1, r(lq[14]);
acc:=acc+sor, sor:=sor>2, mbus=mor, le, w(lq[15]);
acc:=acc+sor, r(lq[11]);
```

```
/* Fourth second order section */

le, w(temp), acc:=mor, sor:=mor>2;
r(temp), acc:=acc+sor, sor:=sor>5;
sor:=mor>2, acc:=acc+~sor, r(lq[12]);
acc:=acc+sor, sor:=mor, r(lq[13]);
acc:=acc+~sor, sor:=sor>3, mbus=mor, le, w(lq[14]);
acc:=acc+sor, sor:=sor>2;
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+sor, sor:=mor, r(lq[11]);
acc:=acc+sor, sor:=mor>2;
le, w(lq[10]), acc:=acc+~sor, sor:=sor>3;
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+~sor, sor:=mor, r(lq[12]);
acc:=acc+sor, sor:=mor;
acc:=acc+sor, r(lq[14]);

/* Fifth second order section */

le, w(temp), acc:=mor, sor:=mor>2;
acc:=acc+sor, sor:=sor>1, r(lq[2]);
acc:=acc+sor, sor:=sor>3, mbus=mor, le, w(lq[3]);
r(temp), acc:=acc+sor, sor:=sor>1;
sor:=mor>2, acc:=acc+sor, r(lq[15]);
acc:=acc+sor, sor:=mor, r(lq[1]);
acc:=acc+~sor, sor:=sor>3, mbus=mor, le, w(lq[2]);
acc:=acc+sor, sor:=sor>2;
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+~sor, sor:=mor, r(lq[15]);
acc:=acc+sor, sor:=mor, r(one);
le, w(lq[13]), acc:=acc+~sor, sor:=mor;
acc:=acc+sor;

/* output scaling and signal multiplexing */

le, w(temp);
r(temp);
acc:=mor, sor:=mor, r(hresult), ORIG;
acc:=acc+sor, sor:=sor>3, mbus=mor, le, wc(temp), ALBA;
acc:=acc+sor,  wc(alb), ANSW;
wc(temp1);
le, wc(temp), ORIG;
wc(temp1), ALBO;
wc(alb);
r(alb), ALBSET;
mbus=mor, le, wc(temp);
r(temp);
mbus=mor, demod:=mbus;     /* send data to be demodulated to */
                           /* processor modem */


/* HIGHBAND FILTER */

/* First second order section */

r(hq);
sor:=mor>3, r(hq[2]);
acc:=sor, sor:=mor>2, r(hq[5]);
acc:=acc+sor, sor:=sor>1, mbus=mor, le, w(hq[6]);
acc:=acc+sor, sor:=sor>4, r(hq[3]);
acc:=acc+~sor, sor:=mor;
acc:=acc+~sor, sor:=sor>3, r(hq[4]);
acc:=acc+sor, sor:=sor>2, mbus=mor, le, w(hq[5]);
acc:=acc+sor, sor:=sor>2, r(three);
acc:=acc+~sor, sor:=mor, r(hq[2]);
```

modem.df

```
acc:=acc+sor, sor:=mor>1;
le, w(hq[1]), acc:=acc+~sor, sor:=sor>3;
acc:=acc+~sor, sor:=sor>3, r(three);
acc:=acc+~sor, sor:=mor, r(hq[3]);
acc:=acc+sor, sor:=mor, r(hq[8]);
acc:=acc+sor, sor:=sor>2, mbus=mor, le, w(hq[9]);
acc:=acc+sor, sor:=sor>1, r(hq[7]);
acc:=acc+sor, sor:=sor>3, mbus=mor, le, w(hq[8]);
acc:=acc+sor, sor:=sor>1, r(hq[5]);
acc:=acc+sor, sor:=mor>2;

/* Second second order section */

le, w(temp), acc:=sor, sor:=sor>1;
r(temp), acc:=acc+sor, sor:=sor>2;
sor:=mor>3, acc:=acc+sor, r(hq[6]);
acc:=acc+sor, sor:=mor>1, r(hq[11]);
acc:=acc+~sor, sor:=sor>1, mbus=mor, le, w(hq[12]);
acc:=acc+~sor, sor:=sor>3;
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+sor, sor:=mor, r(hq[5]);
acc:=acc+sor, sor:=mor;
le, w(hq[4]), acc:=acc+~sor, sor:=sor>1;
acc:=acc+~sor, sor:=sor>4, r(three);
acc:=acc+~sor, sor:=mor, r(hq[6]);
acc:=acc+sor, sor:=mor, r(hq[8]);
acc:=acc+sor, sor:=mor>3;

/* Third second order section */

le, w(temp), acc:=sor, sor:=sor>4;
r(temp), acc:=acc+sor;
sor:=mor>1, r(hq[9]);
acc:=acc+sor, sor:=mor, r(hq[10]);
acc:=acc+~sor, sor:=sor>3, mbus=mor, le, w(hq[11]);
acc:=acc+sor, sor:=sor>2;
acc:=acc+~sor, sor:=sor>2, r(two);
acc:=acc+sor, sor:=mor, r(hq[8]);
acc:=acc+sor, sor:=mor>2;
le, w(hq[7]), acc:=acc+~sor, sor:=sor>3;
acc:=acc+~sor, sor:=sor>2, r(three);
acc:=acc+~sor, sor:=mor, r(hq[9]);
acc:=acc+sor, sor:=mor, r(hq[14]);
acc:=acc+sor, sor:=sor>2, mbus=mor, le, w(hq[15]);
acc:=acc+sor, sor:=sor>5, r(hq[11]);
acc:=acc+sor, sor:=mor>1;

/* Fourth second order section */

le, w(temp), acc:=sor, sor:=sor>3;
r(temp), acc:=acc+sor, sor:=sor>3;
sor:=mor>2, acc:=acc+~sor, r(hq[12]);
acc:=acc+sor, sor:=mor;
acc:=acc+~sor, sor:=sor>4, r(two);
acc:=acc+sor, sor:=mor, r(hq[11]);
acc:=acc+sor, sor:=mor;
le, w(hq[10]), acc:=acc+~sor, sor:=sor>2;
acc:=acc+~sor, sor:=sor>1, r(hq[13]);
acc:=acc+~sor, sor:=sor>2, mbus=mor, le, w(hq[14]);
acc:=acc+sor, sor:=sor>2, r(four);
acc:=acc+~sor, sor:=mor, r(hq[12]);
acc:=acc+sor, sor:=mor, r(hq[14]);
acc:=acc+sor, sor:=mor>2;
```

```
/* Fifth second order section */

le, w(temp), acc:=sor, sor:=sor>3;
r(temp), acc:=acc+~sor, sor:=sor>2;
sor:=mor>1, acc:=acc+sor, r(hq[15]);
acc:=acc+sor, sor:=mor, r(hq[2]);
acc:=acc+~sor, sor:=sor>2, mbus=mor, le, w(hq[3]);
acc:=acc+sor, sor:=sor>2;
acc:=acc+~sor, sor:=sor>2, r(three);
acc:=acc+sor, sor:=mor, r(hq[15]);
acc:=acc+sor, sor:=mor, r(one);
le, w(hq[13]), acc:=acc+~sor, sor:=mor;
acc:=acc+sor;

/* output scaling */

le, w(temp);
r(temp);
acc:=mor, sor:=mor>1, r(hq[1]);
acc:=acc+sor, sor:=sor>1, mbus=mor, le, w(hq[2]);
acc:=acc+sor;
le, w(hresult);
r(temp1);
mbus=mor, txout:=mbus;  /* output filtered FSK signal */
end
end
```

APPENDIX B
DESIGN FILE OF MODEM WITH SAWTOOTH MODULATOR

```
/* 300 Baud FSK modem - Two processors - */
/* Processor modem contains the modulator and demodulator */
/* Processor filters contains two tenth order bandpass filters */
/* for filtering the transmit and receive signals */

.global
begin
rxin<12>;                /* FSK signal to be demodulated */
txout<12>;               /* FSK modulated data to be transmitted */
                         /*                                      */
wordin<12>;              /* wordin: |O/A|TXD|SQT|ALB|  */
                         /* O/A = mode of operation (A=0) */
                         /* TXD = digital data to be transmitted */
                         /* SQT = squelch modulator */
                         /* ALB = self-test mode, feeds modulated and */
                         /* filtered signal back to demodulator */
                         /*                                     */
wordout<12>;             /* wordout:|RXD|CD | - - -    */
                         /* RXD = demodulated digital data out */
                         /* CD=carrier detect signal(0=carrier present)*/
txmod<12>;               /* globals for interprocessor communication */
demod<12>;
cword<12>;
wd_out<12>;
tmp1<14>;                /* for divide operation in AGC circuit */
tmp2<14>;                /* for multiply operation in demodulator */
end

.io
begin
recin: rxin : signal_in;
ctrlin: wordin : signal_in;
transout: txout : signal_out;
dataout: wordout : signal_out;
end

.processor : modem<14>

begin

.local
begin
  temp;
  d[17];     /* storage for demodulator bandpass and lowpass filters */
  cmp[2];    /* cmp = demodulated signal power at space frequency */
             /* cmp[1] = demodulated signal power at mark frequency */
  p[5];      /* storage for lowpass filter in AGC circuit */
  wave;
  lc;        /* AGC output */
  alcin;     /* AGC input */
  lpfin;     /* input to lowpass filter in AGC circuit */
  lpfout;    /* output of lowpass filter in AGC circuit */
  cd;        /* carrier detect variables */
  count;
  c_past;
  outdata;   /* data to be loaded to wordout */
end

.constant
begin

  /* constants for correcting 1's complement subtractions */

  one=1;
  three=3;
```

modem3.df

```
  /* demodulator bandpass filter coefficients */

  ocoef[2]=4096, 1536;
  acoef[2]=4096, 2304;

  /* constants for computing step size */

  half=-4096;
  k1=-913;        /* step for 1070 Hz, step=(1070/9600)*8191 */
                  /* The sample rate is F=9600 Hz */
  k2=-171;        /* additional step for 1270 Hz */
  k3=-644;        /* additional step for 2025 Hz */
  k4=-170;        /* additional step for 2225 Hz */
  mask1=7168;     /* to get TXD value */
  mask2=3072;     /* to get SQT value */
  mask3=1024;     /* to get ALB value */
  odata=-8192;    /* for writing a "1" to RXD */

  /* Carrier detect constants */

  st1=43;         /* 20 msec on delay */
  st2=-128;       /* 10 msec off delay */
  th1=-33;        /* for -48 dB threshold */
  th2=-25;        /* for -43 dB threshold */
  max=8191;       /* for limiter */
  min=-8191;
end

.fsm
begin
  MSET: cc=!sign;
  SET:  cc=sign;
  MD: md=!sign;                          /* true if originate mode */
  MS: ms=!sign;                          /* true if space */
  ORIG: cc=((!md)&lb)|(md&(!lb));    /* for control of demodulator */
  ANSW: cc=((!md)&(!lb))|(md&lb);    /* bandpass filter coefficients */
  ALB: lb=sign;                          /* true for ALB set */
  SQT: sqt=sign;                         /* true if SQT set */
  SQTSET: cc=sqt;
  OS: cc=md&ms;           /* true if originate and space */
  OM: cc=md&(!ms);        /* etc. */
  AS: cc=(!md)&ms;        /* true if answer and space */
  AM: cc=(!md)&(!ms);     /* etc. */

  /* Carrier detect states */

  CP: scp=sign;                          /* sign of c_past */
  CT: sc=sign;                           /* sign of count */
  XR: sch=((!scp)&(sc))|((scp)&(!sc));   /* EX-OR of scp and sc */
  LM1: cc=sch&!sc;                       /* for limiter operation */
  LM2: cc=sch&sc;
  end




.main_pr

  /* MODULATOR SEGMENT BEGINS HERE */

  begin
```

```
/* get MD, MS, ALB, SQT values */

mbus=cword, le, mor:=~mir;
acc:=mor, r(mask1);
acc:=acc&mor, sor:=mor>1, MD;
acc:=acc+~sor, r(one);
acc:=acc+sor, sor:=mor, MS;
acc:=acc+sor, r(mask2);
acc:=acc&mor, sor:=mor>1;
acc:=acc+~sor, r(one);
acc:=acc+sor, sor:=mor, SQT;
acc:=acc+sor, r(mask3);
acc:=acc&mor, sor:=mor>1;

acc:=acc+~sor, r(wave);
acc:=mor, r(max), ALB;
sor:=mor, SET;
acc:=acc+sor;
le, wc(wave);              /* wave:=wave+1.0 if wave was < 0 */

        /* wave:=wave-step */
        /* if OS is true, wave:=wave-913 */
        /* if OM is true, wave:=wave-1084 */
        /* if AS is true, wave:=wave-1728 */
        /* if AM is true, wave:=wave-1898 */

r(wave);
acc:=mor, r(k1);
sor:=mor, r(k2);
acc:=acc+sor, OS, sor:=mor, r(k3);
le, wc(wave), acc:=acc+sor, OM, sor:=mor;
r(k4);
le, wc(wave), acc:=acc+sor, AS, sor:=mor;
le, wc(wave), acc:=acc+sor, AM;
le, wc(wave);

     /* shape sawtooth wave into a sinewave approximation */

r(wave);
sor:=mor, r(half);
acc:=sor+mor;              /* wave:=wave-0.5 */

le, mor:=~mir;
sor:=mor;
acc:=acc+~sor;             /* wave:=wave*2 */

le, mor:=~mir;
sor:=mor;
acc:=|sor|;                /* wave:=|wave| */

r(half);
sor:=mor;
acc:=acc+sor;              /* wave:=wave-0.5 */

le, w(temp);
sor:=mor, r(temp);
acc:=acc+~sor, sor:=mor;
acc:=acc+sor;              /* wave:=wave*3 */
                           /* wave is now a clipped triangle */
                           /* sinewave approximation */
                           /* -- notice that the accumulator is */
                           /* purposely overflowed */
le, w(temp), acc:=0, SQTSET;
le, wc(temp);
r(temp);
```

```
mbus=mor, txmod:= mbus;
nop;

/* DEMODULATOR SEGMENT BEGINS HERE */

/* Automatic Gain Control (AGC) */

mbus=demod, le, w(alcin);          /* store input to remember sign */
r(alcin);
sor:=mor;
acc:=|sor|;                                  /* FWR of alcin */
le, w(lpfin);                               /* store lpf input */

/* 3rd order loupass filter for envelope detect */

r(lpfin);
sor:=mor>4;
acc:=sor, sor:=sor>, r(p[2]);
acc:=acc+sor, sor:= nor>1;
acc:=acc+~sor, sor: =sor>2;
acc:=acc+~sor, sor: =sor>1, r(three);
acc:=acc+~sor, sor: =mor, r(p[1]);
acc:=acc+sor, sor:= nor;
acc:=acc+sor, sor:= or>1;
acc:=acc+sor, sor:= or>3, r(p[1]);
acc:=acc+sor, sor:= nor>2, r(one);
le, w(p), acc:=acc+ sor, sor:=mor;
acc:=acc+sor, r(p[2 );
sor:=mor, r(p[4]);
acc:=acc+sor, sor:= nor>1;
le, w(temp), acc:=s r, sor:=sor>2;
r(temp), acc:=acc+s or, sor:=sor>2;
sor:=mor>3, acc:=a c+sor, r(p[4]);
acc:=acc+sor, sor:= nor;
le, w(p[3]), acc:=ac +sor;
le, w(temp);
r(temp);                          /* scale output by 1.625 */
acc:=mor, sor:=mor>1;             /* so that denominator of divide */
acc:=acc+sor, sor:= or>2;        /* opeation is greater than numerator */
acc:=acc+sor, r(one);
le, w(lpfout), sor:=n or;              /* store lpf output */
acc:=sor+mor, r(lpf n);
le, w(temp), sor:=m r;
r(temp);                                  /* temp has -lpfout */
acc:=|sor|, sor:=mor>1;                  /* implement lpfin/lpfout */
sor:=sor>1, acc:=so +acc, aip;           /* to normalize received signals*/
sor:=sor>1, acc:=so. +acc, aip;
sor:=sor>1, acc:=so. +acc, aip;
sor:=sor>1, acc:=so +acc, aip;
sor:=sor>1, acc:=so. +acc, aip;
sor:=sor>1, acc:=so. +acc, aip; /* update loupass filter variables */
sor:=sor>1, acc:=so +acc, aip, r(p[1]);
sor:=sor>1, acc:=so. +acc, aip, mbus=mor, le, w(p[2]);
sor:=sor>1, acc:=so. +acc, aip, r(p);
sor:=sor>1, acc:=so +acc, aip, mbus=mor, le, w(p[1]);
sor:=sor>1, acc:=so. +acc, aip, r(p[3]);
sor:=sor>1, acc:=so. +acc, aip, mbus=mor, le, w(p[4]);
acc:=sor+acc, aip;
tmp1:=quot, r(one);
mbus=tmp1, le, w(l ), sor:=mor;          /* lc has tmp=lpfin/lpfout */
acc:=sor+mor, r(alc n);                     /* acc := -tmp */
le, acc:=mor;
```

```
SET;                                    /* recall sign of input to alc */
wc(lc);                                 /* lc := -tmp if cc set */
r(lc);

/* BEGIN CARRIER DETECT FUNCTION */


acc:=mor, r(cd)
le, w(d), sor:=mor;  /* write output of AGC to demodulator */
acc:=sor, r(lpfout); /* bandpass filter inputs */
r(th1), sor:=mor, SET;      /* sor:= lpfout , cc=sign of cd */
acc:=sor+mor, r(th2);       /* acc:= lpfout - 48 dB threshold */
le, w(cd), sor:=mor;
acc:=acc+sor, r(count);     /* acc:= lpfout - 43 dB threshold */
le, wc(cd), sor:=mor;       /* use 43 dB threshold if cd was <0 */
r(st1);
acc:=sor+mor, r(st2);       /* acc:= count + 43 for turn on delay*/
le, w(count), sor:=mor;
acc:=acc+sor, r(c_past);    /* acc:= count-128 for turn off delay*/
le, wc(count), acc:=mor;
r(count), CT;
acc:=mor;
CT;
r(max), XR
acc:=mor, LM1, r(min);                  /* limit count if there */
le, wc(count), acc:=mor, LM2;           /* was a sign change between */
le, wc(count);                          /* count and c_past */
r(count);
mbus=mor, le, wc(c_past), acc:=mor;
acc:=0, SET, r(coef);                   /* cc= sign of count */
le, w(outdata), acc:=mor;               /* CD = 0 if carrier present */
le, wc(outdata);


/* Demodulator envelope detectors comparison */

r(cmp);
sor:=mor, r(cmp[1]);
acc:=mor+sor, r(one);    /* cmp[1]-cmp > 0 : data=mark */
sor:=mor, r(outdata);    /* cmp[1]-cmp < 0 : data=space */
acc:=acc+sor, sor:=mor, r(odata);
acc:=sor+mor, !SET;
le, wc(outdata);
r(outdata);
mbus=mor, wd_out:=mbus;
end


/* Implements demodulator bandpass and lowpass filters. */
/* First iteration does space filtering in either O or A mode. */
/* Second iteration does mark filtering in either O or A mode. */

sub_pr <2>
  begin

  /* implement 2nd order bandpass filters */

  rx(acoef), ORIG;
  mbus=mor, le, wc(temp);
  rx(ocoef);
  mbus=mor, le, wc(temp);
  r(temp);
  rx(d[3]), mbus=mor, tmp2:=mbus;
  sor:=mor;
  sor:=sor>1, acc:=coef.~sor, coef:=tmp2;
```

```
    sor:=sor>1, acc:=acc+coef sor;
    sor:=sor>1, acc:=acc+coef sor;
    sor:=sor>1, acc:=acc+coef sor;
    sor:=sor>1, acc:=acc+coef sor;
    sor:=sor>1, acc:=acc+coef sor;
    sor:=sor>1, acc:=acc+coef sor;
    sor:=sor>1, acc:=acc+coef sor;
    sor:=sor>1, acc:=acc+coef sor;
    sor:=sor>1, acc:=acc+coef sor,rx(d[9]);
    sor:=sor>1, acc:=acc+coef sor,mbus=mor, le, wx(d[11]);
    sor:=sor>1, acc:=acc+coef sor,rx(d[7]);
    sor:=sor>1, acc:=acc+coef sor,mbus=mor, le, wx(d[9])
    acc:=acc+coef.sor, rx(d[3]);
    sor:=mor;
    le, w(temp), acc:=acc+sor  ANSW;
    le, wc(temp);
    r(temp);
    sor:=mor, rx(d[5]);
    acc:=sor, sor:=mor;
    acc:=acc+~sor, sor:=sor>4, r(d);
    acc:=acc+sor, sor:=mor>6
    acc:=acc+sor, sor:=sor>6, r(one);
    acc:=acc+sor, sor:=mor, rx(d[5]);
    acc:=acc+sor, sor:=mor, r(one);
    le, wx(d[1]), acc:=acc+~scr, sor:=mor;
    acc:=acc+sor;
    le, w(temp);


    r(temp);
    sor:=mor<1, rx(d[13]);
    acc:=|sor|, mbus=mor, le, wx(d[15]);   /* full wave rectify (FWR) */
    le, w(temp);                                /* of bandpass filter output*/

    /* 3rd order lowpass filter to remove components around 2fc */

    r(temp);
    sor:=mor>4;
    acc:=sor, sor:=sor>1, rx(d[11]);
    acc:=acc+sor, sor:=mor>1, rx(d[3]);
    acc:=acc+~sor, sor:=sor>2, mbus=mor, le, wx(d[5]);
    acc:=acc+~sor, sor:=sor>1, r(three);
    acc:=acc+~sor, sor:=mor, rx(d[9]);
    acc:=acc+sor, sor:=mor, rx(d[1]);
    acc:=acc+sor, sor:=sor>1, mbus=mor, le, wx(d[3]);
    acc:=acc+sor, sor:=sor>3, rx(d[9]);
    acc:=acc+sor, sor:=mor>2  r(one);
    le, wx(d[7]), acc:=acc+~scr, sor:=mor;
    r(temp), acc:=acc+sor, sor:=sor>2;
    sor:=mor>3, acc:=acc+sor  rx(d[15]);
    acc:=acc+sor, sor:=mor;
    le, wx(d[13]), acc:=acc+sc-;
    le, wx(cmp);
    end
end
```

```
/* PROCESSOR FILTERS IMPLEMENTS TWO TENTH ORDER BANDPASS FILTERS */

.processor : filters<20>

begin

.local
begin
  temp;
  lq[16];        /* storage for lowband filter */
  hq[16];        /* storage for highband filter */
  alb;           /* for self-test function */
  temp1;
  hresult;       /* highband filter output */
end

.constant
begin
  /* constants for correcting 1's complement subtractions */

  one=1;
  two=2;
  three=3;
  four=4;
  mask=65536;  /* to get ALB status from control word */
end

.fsm
begin
  SET: cc=sign;
  MSET: cc=!sign;
  MD: md=!sign;        /* true for originate */
  ORIG: cc=md;
  ANSW: cc=!md;
  ALB: lb=sign;        /* true if ALB set */
  ALB0: cc=lb&md;      /* states for controlling ALB function */
  ALBA: cc=lb&!md;
  ALBSET: cc=lb;
end


.main_pr

begin

/* get MD and ALB and perform signal multiplexing */

mbus=wordin, le, mor:=~mir, cword:=mbus; /* send wordin to processor*/
acc:=mor, r(mask);                       /* modem via cword */
acc:=acc&mor, sor:=mor>1, MD;
acc:=acc+~sor, ORIG;
mbus=txmod, le, wc(lq), ALB;   /* get txmod from processor modem */
ANSW;                          /* to filter it before transmitting */
wc(hq);
mbus=rxin, le, wc(lq), ORIG;   /* get signal to be filtered and */
wc(hq);                        /* demodulated */
mbus=wd_out, wordout:=mbus;    /* transfer processor modem variable */
```

```
                                        /* wd_out to wordout output */
/* LOWBAND FILTER */

/* First second order section - 2 poles, then 2 zeros */

r(lq);
sor:=mor>4, r(lq[2]);
acc:=sor, sor:=mor, r(lq[5]);
acc:=acc+sor, sor:=sor>2, mbus=mor, le, w(lq[6]);
acc:=acc+sor, sor:=sor>2, r(lq[3]);
acc:=acc+sor, sor:=mor;
acc:=acc+~sor, sor:=sor>3, r(lq[4]);
acc:=acc+sor, sor:=sor>2, mbus=mor, le, w(lq[5]);
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+~sor, sor:=mor, r(lq[2]);
acc:=acc+sor, sor:=mor;
le, w(lq[1]), acc:=acc+~sor, sor:=sor>1;
acc:=acc+~sor, sor:=sor>2;
acc:=acc+~sor, sor:=sor>3, r(four);
acc:=acc+~sor, sor:=mor, r(lq[3]);
acc:=acc+sor, sor:=mor, r(lq[3]);
acc:=acc+sor, sor:=sor>3, mbus=mor, le, w(lq[9]);
acc:=acc+sor, sor:=sor>2, r(lq[7]);
acc:=acc+sor, sor:=sor>1, mbus=mor, le, w(lq[8]);
acc:=acc+sor, r(lq[5]);

/* Second second order section */

le, w(temp), acc:=mor, sor:=mor>1;
r(temp), acc:=acc+sor, sor:=sor>6;
sor:=mor>2, acc:=acc+sor, r(lq[6]);
acc:=acc+sor, sor:=mor, r(lq[11]);
acc:=acc+~sor, sor:=sor>3, mbus=mor, le, w(lq[12]);
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+~sor, sor:=mor, r(lq[5]);
acc:=acc+sor, sor:=mor>1;
le, w(lq[4]), acc:=acc+~sor, sor:=sor>5;
acc:=acc+sor, r(one);
sor:=mor, r(lq[6]);
acc:=acc+sor, sor:=mor;
acc:=acc+sor, r(lq[8]);

/* Third second order section */

le, w(temp), acc:=mor, sor:=mor>2;
r(temp), acc:=acc+sor, sor:=sor>5;
sor:=mor>1, acc:=acc+~sor, r(lq[9]);
acc:=acc+sor, sor:=mor;
acc:=acc+~sor, sor:=sor>4, r(two);
acc:=acc+sor, sor:=mor, r(lq[13]);
acc:=acc+sor, sor:=mor;
le, w(lq[7]), acc:=acc+~sor, sor:=sor>1;
acc:=acc+~sor, sor:=sor>6, r(two);
acc:=acc+sor, sor:=mor, r(lq[9]);
acc:=acc+sor, sor:=mor, r(lq[10]);
acc:=acc+sor, sor:=sor>3, mbus=mor, le, w(lq[11]);
acc:=acc+sor, sor:=sor>1, r(lq[14]);
acc:=acc+sor, sor:=sor>2, mbus=mor, le, w(lq[15]);
acc:=acc+sor, r(lq[11]);

/* Fourth second order section */

le, w(temp), acc:=mor, sor:=mor>2;
r(temp), acc:=acc+sor, sor:=sor>5;
```

```
sor:=mor>2, acc:=acc+~sor, r(lq[12]);
acc:=acc+sor, sor:=mor, r(lq[13]);
acc:=acc+~sor, sor:=sor>3, mbus=mor, le, w(lq[14]);
acc:=acc+sor, sor:=sor>2;
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+sor, sor:=mor, r(lq[11]);
acc:=acc+sor, sor:=mor>2;
le, w(lq[10]), acc:=acc+~sor, sor:=sor>3;
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+~sor, sor:=mor, r(lq[12]);
acc:=acc+sor, sor:=mor;
acc:=acc+sor, r(lq[14]);

/* Fifth second order section */

le, w(temp), acc:=mor, sor:=mor>2;
acc:=acc+sor, sor:=sor>1, r(lq[2]);
acc:=acc+sor, sor:=sor>3, mbus=mor, le, w(lq[3]);
r(temp), acc:=acc+sor, sor:=sor>1;
sor:=mor>2, acc:=acc+sor, r(lq[15]);
acc:=acc+sor, sor:=mor, r(lq[1]);
acc:=acc+~sor, sor:=sor>3, mbus=mor, le, w(lq[2]);
acc:=acc+sor, sor:=sor>2;
acc:=acc+sor, sor:=sor>2, r(two);
acc:=acc+~sor, sor:=mor, r(lq[15]);
acc:=acc+sor, sor:=mor, r(one);
le, w(lq[13]), acc:=acc+~sor, sor:=mor;
acc:=acc+sor;

/* output scaling and signal multiplexing */

le, w(temp);
r(temp);
acc:=mor, sor:=mor, r(hresult), ORIG;
acc:=acc+sor, sor:=sor>3, mbus=mor, le, wc(temp), ALBA;
acc:=acc+sor,  wc(alb), ANSW;
wc(temp1);
le, wc(temp), ORIG;
wc(temp1), ALBO;
wc(alb);
r(alb), ALBSET;
mbus=mor, le, wc(temp);
r(temp);
mbus=mor, demod:=mbus;     /* send data to be demodulated to */
                           /* processor modem */


/* HIGHBAND FILTER */

/* First second order section */

r(hq);
sor:=mor>3, r(hq[2]);
acc:=sor, sor:=mor>2, r(hq[5]);
acc:=acc+sor, sor:=sor>1, mbus=mor, le, w(hq[6]);
acc:=acc+sor, sor:=sor>4, r(hq[3]);
acc:=acc+~sor, sor:=mor;
acc:=acc+~sor, sor:=sor>3, r(hq[4]);
acc:=acc+sor, sor:=sor>2, mbus=mor, le, w(hq[5]);
acc:=acc+sor, sor:=sor>2, r(three);
acc:=acc+~sor, sor:=mor, r(hq[2]);
acc:=acc+sor, sor:=mor>1;
le, w(hq[1]), acc:=acc+~sor, sor:=sor>3;
acc:=acc+~sor, sor:=sor>3, r(three);
acc:=acc+~sor, sor:=mor, r(hq[3]);
```

```
acc:=acc+sor, sor:=mor, r(hq[8]);
acc:=acc+sor, sor:=scr>2, mbus=mor, le, w(hq[9]);
acc:=acc+sor, sor:=scr>1, r(hq[7]);
acc:=acc+sor, sor:=scr>3, mbus=mor, le, w(hq[8]);
acc:=acc+sor, sor:=scr>1, r(hq[5]);
acc:=acc+sor, sor:=mor>2;

/* Second second order section */

le, w(temp), acc:=sor, sor:=sor>1;
r(temp), acc:=acc+so , sor:=sor>2;
sor:=mor>3, acc:=acc+sor, r(hq[6]);
acc:=acc+sor, sor:=mor>1, r(hq[11]);
acc:=acc+~sor, sor:= or>1, mbus=mor, le, w(hq[12]);
acc:=acc+~sor, sor:= or>3;
acc:=acc+sor, sor:=scr>2, r(two);
acc:=acc+sor, sor:=mor, r(hq[5]);
acc:=acc+sor, sor:=mor;
le, w(hq[4]), acc:=acc+~sor, sor:=sor>1;
acc:=acc+~sor, sor:= or>4, r(three);
acc:=acc+~sor, sor:=nor, r(hq[6]);
acc:=acc+sor, sor:=mor, r(hq[8]);
acc:=acc+sor, sor:=mor>3;

/* Third second order section */

le, w(temp), acc:=sor, sor:=sor>4;
r(temp), acc:=acc+so ;
sor:=mor>1, r(hq[9]);
acc:=acc+sor, sor:=mor, r(hq[10]);
acc:=acc+~sor, sor:= or>3, mbus=mor, le, w(hq[11]);
acc:=acc+sor, sor:=scr>2;
acc:=acc+~sor, sor:= or>2, r(two);
acc:=acc+sor, sor:=mor, r(hq[8]);
acc:=acc+sor, sor:=mor>2;
le, w(hq[7]), acc:=acc+~sor, sor:=sor>3;
acc:=acc+~sor, sor:= or>2, r(three);
acc:=acc+~sor, sor:=nor, r(hq[9]);
acc:=acc+sor, sor:=mor, r(hq[14]);
acc:=acc+sor, sor:=scr>2, mbus=mor, le, w(hq[15]);
acc:=acc+sor, sor:=scr>5, r(hq[11]);
acc:=acc+sor, sor:=mor>1;

/* Fourth second order section */

le, w(temp), acc:=sor, sor:=sor>3;
r(temp), acc:=acc+so , sor:=sor>3;
sor:=mor>2, acc:=acc+~sor, r(hq[12]);
acc:=acc+sor, sor:=mor;
acc:=acc+~sor, sor:= or>4, r(two);
acc:=acc+sor, sor:=mor, r(hq[11]);
acc:=acc+sor, sor:=mor;
le, w(hq[10]), acc:=acc+~sor, sor:=sor>2;
acc:=acc+~sor, sor:= or>1, r(hq[13]);
acc:=acc+~sor, sor:= or>2, mbus=mor, le, w(hq[14]);
acc:=acc+sor, sor:=scr>2, r(four);
acc:=acc+~sor, sor:=nor, r(hq[12]);
acc:=acc+sor, sor:=mor, r(hq[14]);
acc:=acc+sor, sor:=mor>2;

/* Fifth second order section */

le, w(temp), acc:=sor, sor:=sor>3;
r(temp), acc:=acc+~sor, sor:=sor>2;
sor:=mor>1, acc:=acc+sor, r(hq[15]);
```

modem3.df

```
acc:=acc+sor, sor:=mor, r(hq[2]);
acc:=acc+~sor, sor:=sor>2, mbus=mor, le, w(hq[3]);
acc:=acc+sor, sor:=sor>2;
acc:=acc+~sor, sor:=sor>2, r(three);
acc:=acc+sor, sor:=mor, r(hq[15]);
acc:=acc+sor, sor:=mor, r(one);
le, w(hq[13]), acc:=acc+~sor, sor:=mor;
acc:=acc+sor;

/* output scaling */

le, w(temp);
r(temp);
acc:=mor, sor:=mor>1, r(hq[1]);
acc:=acc+sor, sor:=sor>1, mbus=mor, le, w(hq[2]);
acc:=acc+sor;
le, w(hresult);
r(temp1);
mbus=mor, txout:=mbus;   /* output filtered FSK signal */
end
end
```

APPENDIX C
DELAY LINE DEMODULATOR

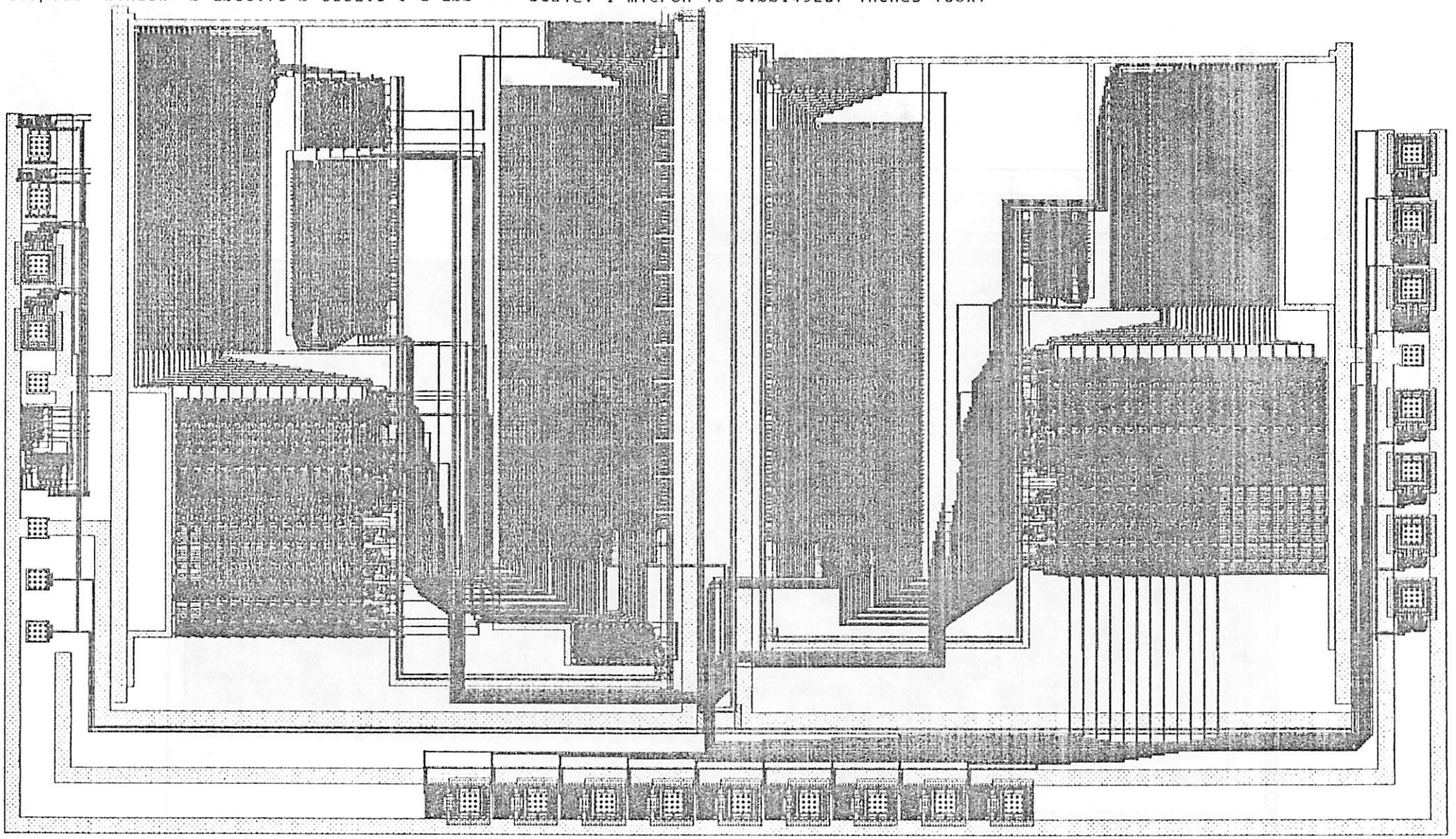
```
.global
begin
  tmp<14>;                          /* for multiply operation */
  wordin<12>;                       /* for getting mode -      */
                                    /* wordin: |O/A| - - - |   */
  in<12>, out<12>;                  /* datain is data in fsk form */
                                    /* to be demodulated - it has been */
                                    /* through the receive bandpass */
                                    /* filter */
end

.io
begin
  ctrlin: wordin : signal_in;
  dm3in: in : signal_in;
  dm3out: out : signal_out;
end

.processor : dm3<14>

/* This demodulator is the delay-line discriminator version. */
/* The signal is multiplied by a one sample delayed signal     */
/* for the 1170 Hz threshold (answer mode), and by a two       */
/* sample delayed signal for the 2125 threshold (originate    */
/* mode). */

begin

.local
  begin
  temp;
  q[3];             /* for the delay storage */
  p[5];             /* for the 3rd order loupass filter */
  rxd;              /* demodulated data */
  end

.constant
  begin
  one=1;
  three=3;
  THO=-680;         /* threshold for comparing mark and space */
                    /* when operating in originate mode */
  THA=519;          /* -680+519=-161 is threshold for comparing */
                    /* mark and space when operating in answer mode */
  dataone=-8192; /* for data output */
  end

.fsm
  begin
  MD: md=!sign;             /* true for originate mode */
  ORIG: cc=md;                 /* originate mode flag */
  ANSW: cc=!md;                /* answer mode flag */
  SET: cc=sign;
  end

.main_pr
  begin
  mbus=wordin, le, mor:=~mir;
  acc:=mor;
  MD;                                   /* set mode of operation */

  /* get input signal and load into tmp for multiply */

  mbus=in, le, w(q), tmp:=mbus;
```

```
/* multiply incoming signal by itself delayed by 1 sample or */
/* two samples depending on mode */

r(q[1]);
mbus=mor, le, w(temp);
r(q[2]), ANSW;
mbus=mor, le, wc(temp);
r(temp);
sor:=mor;
sor:=sor>1, acc:=coef.~sor, coef:=tmp;
sor:=sor>1, acc:=acc+coef.sor;
sor:=sor>1, acc:=acc+coef.sor;
sor:=sor>1, acc:=acc+coef.sor;
sor:=sor>1, acc:=acc+coef.sor;
sor:=sor>1, acc:=acc+coef.sor;
sor:=sor>1, acc:=acc+coef.sor; /* update states of lowpass filter */
sor:=sor>1, acc:=acc+coef.sor, r(p[1]);
sor:=sor>1, acc:=acc+coef.sor, mbus=mor, le, w(p[2]);
sor:=sor>1, acc:=acc+coef.sor, r(p);
sor:=sor>1, acc:=acc+coef.sor, mbus=mor, le, w(p[1]);
sor:=sor>1, acc:=acc+coef.sor, r(p[3]);
sor:=sor>1, acc:=acc+coef.sor, mbus=mor, le, w(p[4]);
acc:=acc+coef.sor;
le, w(temp);

/* Result of multiply is input to lowpass filter. */
/* The lowpass filter removes the frequency components */
/* at twice the signal frequency which are generated by */
/* the multiply. */

/* Lowpass filter is 3rd order - implemented as a cascade of */
/* one second order section and one first order section */

r(temp);
sor:=mor>4;
acc:=sor, sor:=sor>1, r(p[2]);
acc:=acc+sor, sor:=mor>1, r(q[1]);
acc:=acc+~sor, sor:=sor>2, mbus=mor, le, w(q[2]);
acc:=acc+~sor, sor:=sor>1, r(three);
acc:=acc+~sor, sor:=mor, r(p[1]);
acc:=acc+sor, sor:=mor, r(q);
acc:=acc+sor, sor:=sor>1, mbus=mor, le, w(q[1]);
acc:=acc+sor, sor:=sor>3, r(p[1]);
acc:=acc+sor, sor:=mor>2, r(one);
le, w(p), acc:=acc+~sor, sor:=mor;
acc:=acc+sor, r(p[2]);
sor:=mor, r(p[4]);
acc:=acc+sor, sor:=mor>1;
le, w(temp), acc:=sor, sor:=sor>2;
r(temp), acc:=acc+sor, sor:=sor>2;
sor:=mor>3, acc:=acc+sor, r(p[4]);
acc:=acc+sor, sor:=mor, r(THO);
le, w(p[3]), acc:=acc+sor, sor:=mor;

/* compare with originate threshold */

acc:=acc+sor, r(THA), ORIG;
sor:=mor, le, wc(rxd);

/* compare with answer threshold */

acc:=acc+sor, ANSW;
le, wc(rxd);
r(rxd), acc:=0;
```

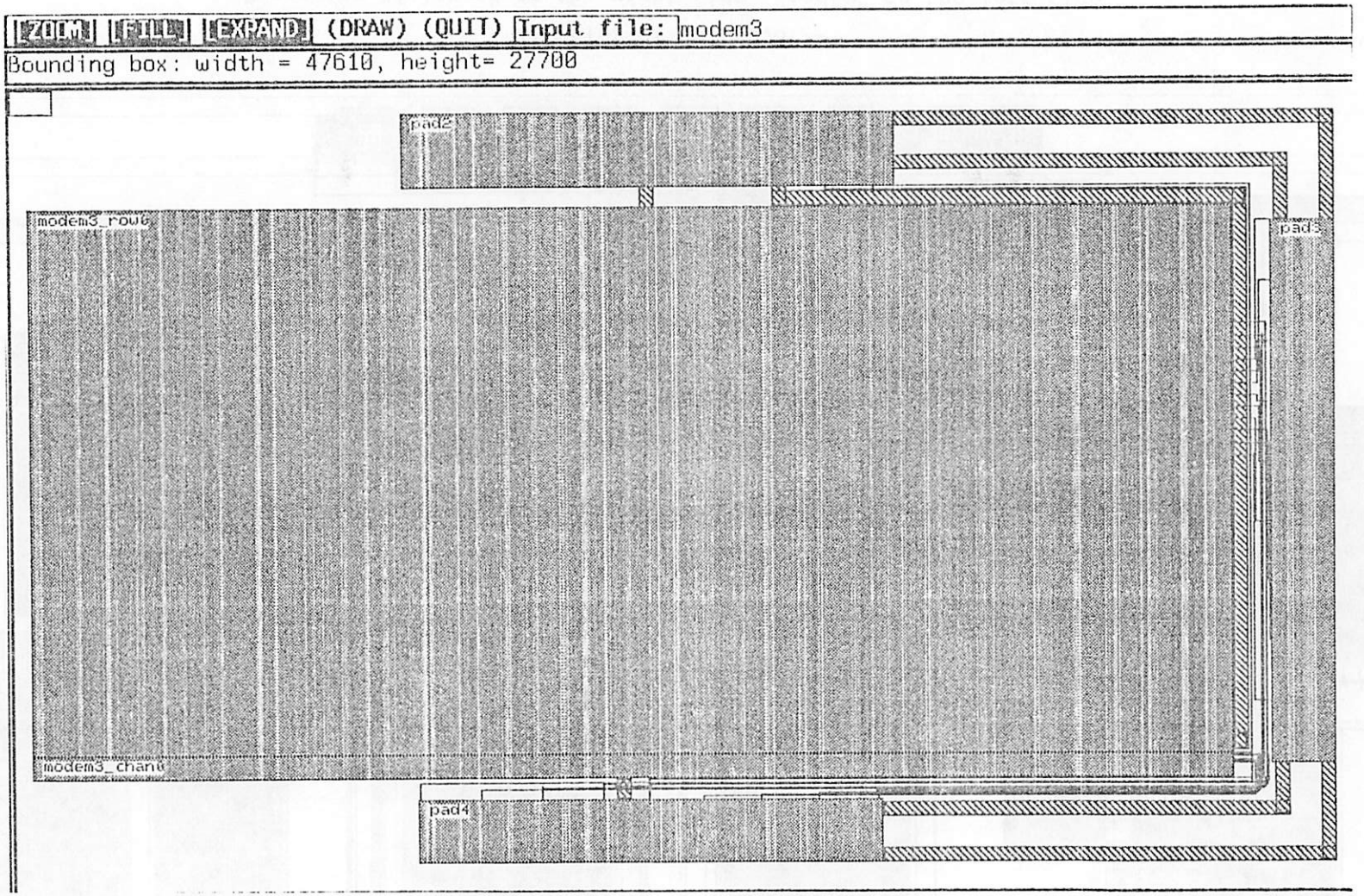
dm3.df

```
    le, w(rxd), acc:=mor;      /* write 0 to out if space */
    r(dataone), SET;
    acc:=mor;
    le, wc(rxd);               /* write 1 to out if mark */
                               /* note that the data is in the */
                               /* MSB position of the output word */
                               /* so that a 1 is represented by -8192 */
    r(rxd);
    mbus=mor, out:=mbus;
    nop;
    end
  end
```

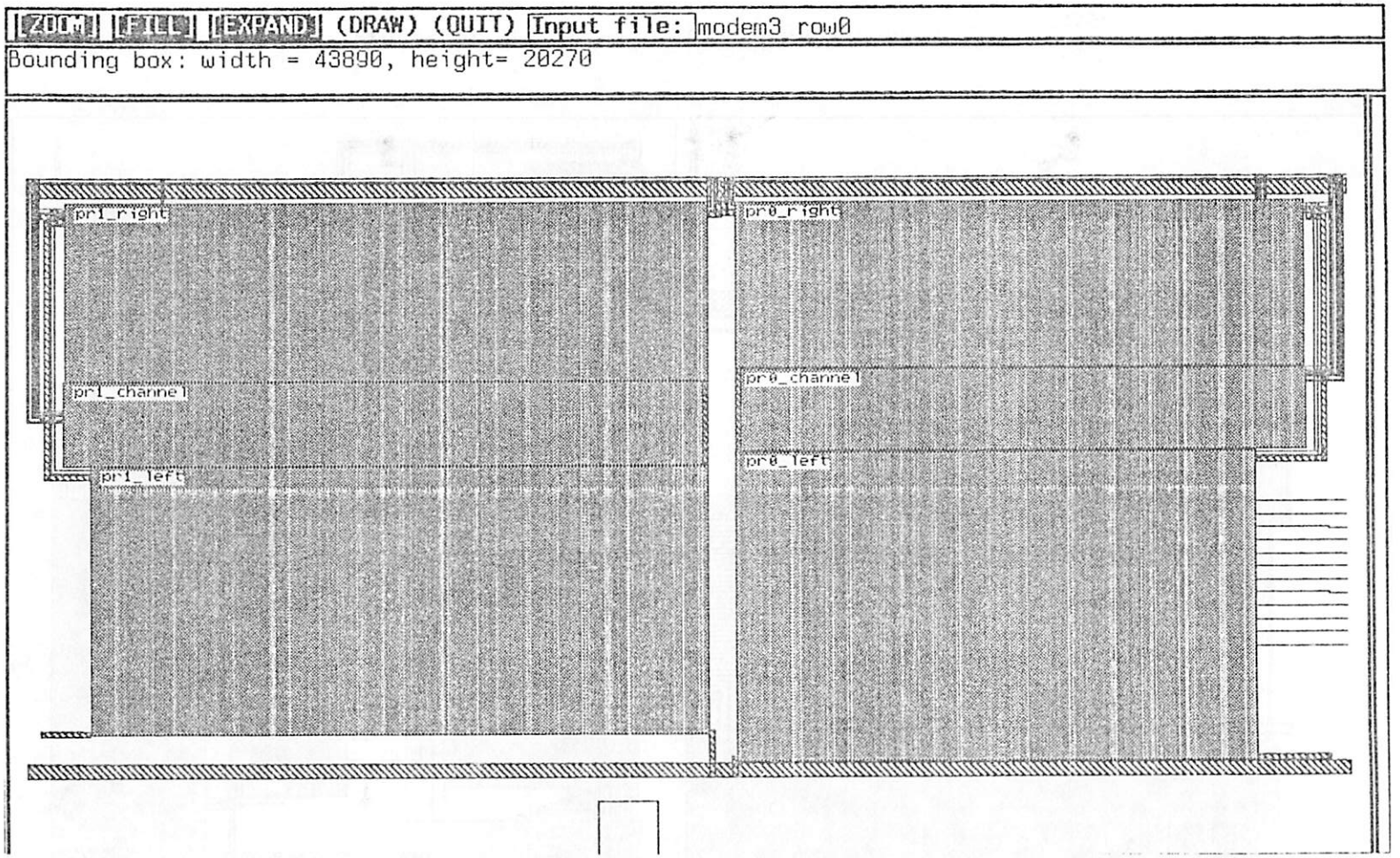
APPENDIX D
CHIP LAYOUT PLOTS



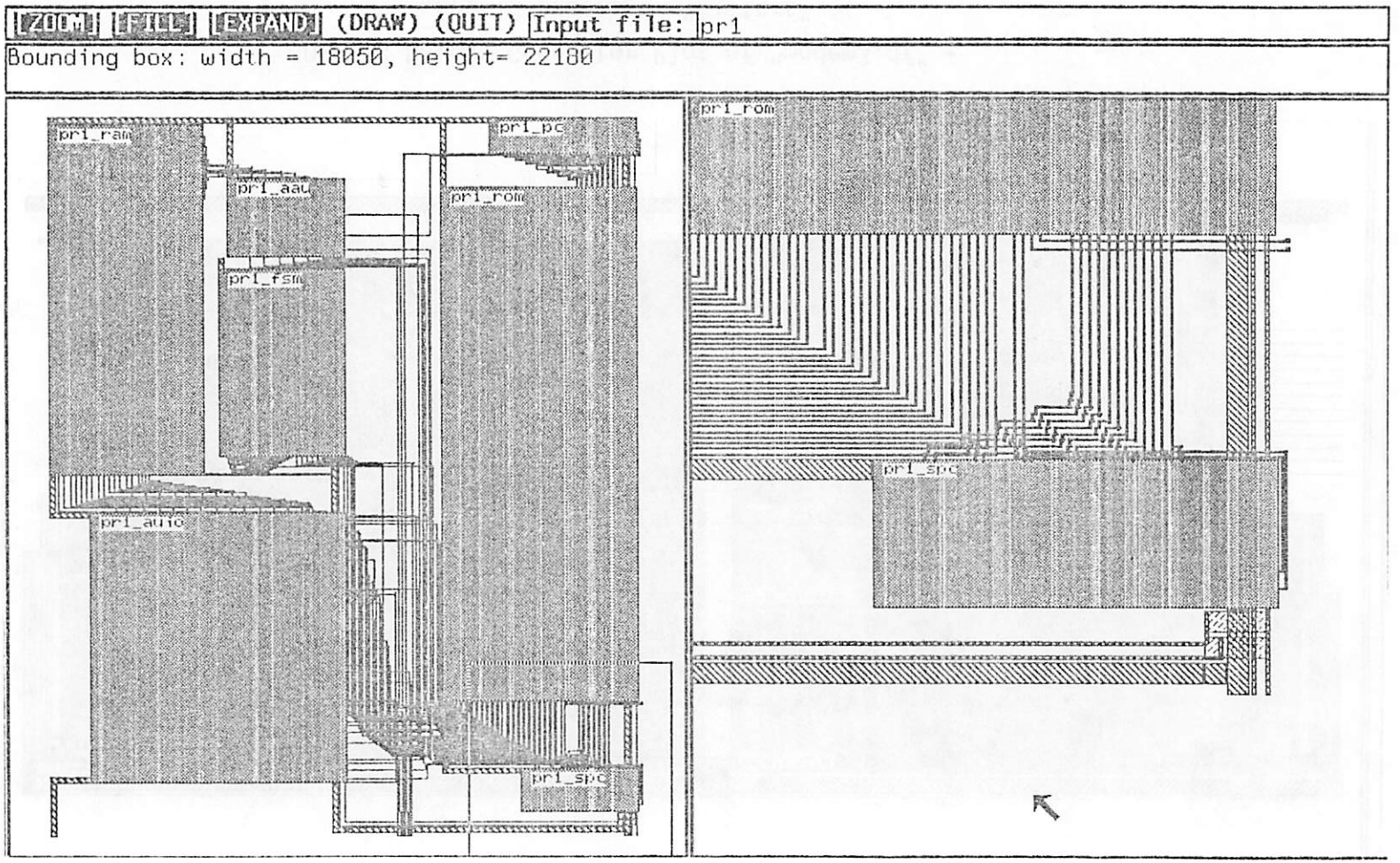
CIF Plot of Modem3.df -
Processor "filters" on the right side, "modem" on the left side



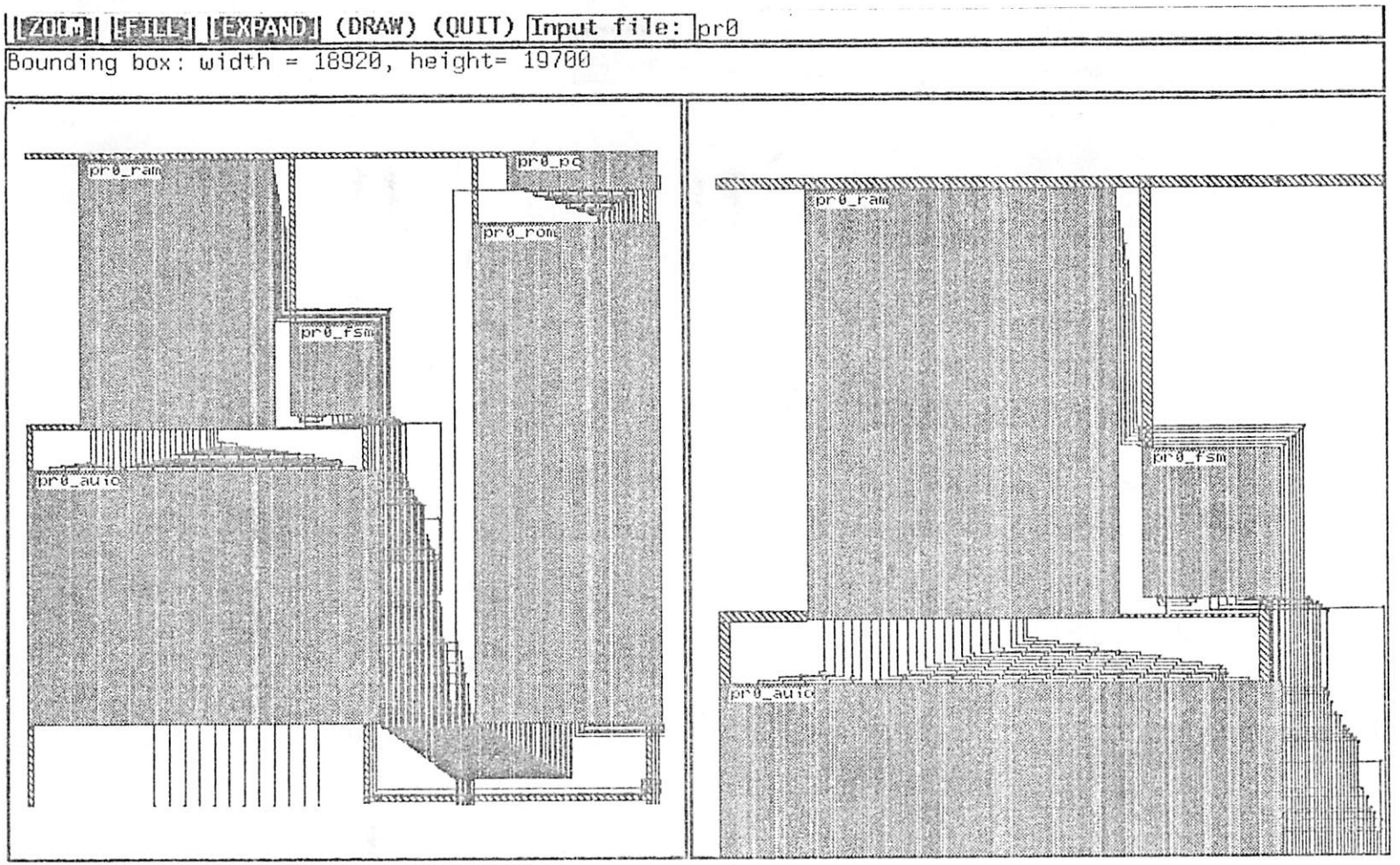
DSP IC Layout Generator Plot of "modem3.df"



DSP IC Layout Generator Plot of "modem3.df" -
pr0 is processor "filters"



Processor "modem" - detail shown on the right



Processor "filters" - detail shown on the right