IMPLEMENTING TABLE LOOKUP OSCILLATORS FOR MUSIC WITH THE MOTOROLA DSP56000 FAMILY

2716 (A-6)

John Strawn
S Systems
San Rafael, California

Presented at the 85th Convention 1988 November 3-6 Los Angeles

AN AUDIO ENGINEERING SOCIETY PREPRINT

Implementing Table Lookup Oscillators for Music with the Motorola DSP56000 Family

John Strawn[1]
S Systems
Post Office Box 623
San Rafael, California 94915-0623, USA

Abstract

The Motorola DSP56000 chip family is starting to appear in a number of digital musical and audio applications. Various claims have been made about the number of oscillators that can be implemented on the chip. Working within a prototypical hardware architecture that will be briefly discussed, I have implemented some table lookup oscillators in 56000 assembler. The oscillator can be used either as a sine-wave oscillator or for playback in a sampling synthesizer. The preprint for this paper includes code examples. In the preprint and in the oral presentation, the musical requirements for an oscillator will be presented, especially concerning updating the frequency and amplitude terms. There will be a discussion of the numbers of oscillators that can be achieved under varying conditions, and a discussion of possible software and hardware design tradeoffs.

Introduction

Chip manufacturers [1, 2] have published application notes concerning the implementation of oscillators for their DSP chip families. Unfortunately, these articles have not yet treated *musical* requirements for oscillators. Several groups are starting to use the 56000 in music processors (see, for example, [3]). In this paper, we will examine the requirements for a real-time wavetable lookup oscillator implemented on the Motorola 56000 chip, a state-of-the-art DSP chip. As one might expect, the chip's architecture is being steadily improved by the manufacturer; the recently announced floating-point 96000 chip family will not be treated here. Indeed, given that many of the operations in table lookup (such as address calculation) are of an integer nature, those planning to implement an oscillator with a more expensive floating-point chip should study whether the floating-point arithmetic capabilities warrant the increased cost.

---

[1] Current address: Yamaha Music Technologies USA, Inc. (YMT); Wood Island Building, Suite 2B; 80 East Sir Francis Drake Boulevard; Larkspur Landing CA 94939; tel. (415) 925 0206; uucp: ...ucbvax!pixar!ymt!john.

Requirements of a lookup oscillator for music

Waveform generators are used in a variety of audio applications. For music, various kinds of waveform generators corresponding to the amazing variety of synthesis techniques have been implemented over the years. Here we will limit ourselves to wavetable lookup oscillators. Note that a wavetable lookup can be used either for generating sinusoidal waves, or for a sampling synthesizer.

Following the usual additive synthesis, FM, and sampling models, the generalized lookup oscillator needs to have the following capabilities:

1. perform, obviously, a table lookup---but not so obviously, possibly interpolate between adjacent samples. Interpolation can increase the signal/noise ratio of the signal tremendously [4, 5, 6];

2. update the current position in the wavetable according to a frequency input;

3. modify the frequency input by a frequency modulation (FM) [7] input;

4. multiply the sample from the table by an amplitude envelope value;

5. update that amplitude envelope;

6. write the result to somewhere in memory.

A model oscillator: the Samson Box

The goal of this study, prepared originally for the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford, was to determine how many Samson-Box style oscillators [8, 9, 10] could be implemented on one 56000 chip. The specifications of the Samson Box oscillator are summarized in Figure 1. This oscillator is generalized enough that it is useful as a model for a comparison implementation. A few quirks of the Samson Box oscillator will need to be discussed briefly, since terminology from that oscillator will be used in the code given here. The Samson Box specification uses the upper-case codes given in the first column of the figure. I adopted the C-style names given in the third column.

The oscillator performs one lookup, with *angle* being the current position inside the wavetable. *freq* is the radian frequency; *freqSweep* specifies the frequency envelope, and is the amount by which to change freq on each sample.

For the amplitude envelope, the *exponent* is the current stored value. *decay* is added to exponent during each sample. We are working with linear envelopes here. The Samson Box has provisions for exponential envelopes which we will not consider. The Samson Box has the further provision for *asymptote*; contrary to what its name implies, asymptote is added into the updated exponent value, so that asymptote acts as a kind of positive or negative offset to the envelope.

For passing data between oscillators, the Samson Box implements "sum memory." That is, when a quantity is written out to memory, whatever is in memory will be added in. A further trick is that the datum to be written out is right-shifted by one bit, as a precaution against overflow. Sum memory addresses here will be notated in the comments as mem[x], where x is the sum memory address. (The Samson Box distinctions between last-pass and this-pass memory will not be covered here, although hardware designers should examine bank switching as one way for updating parameters in parameter memory).

Figure 2 summarizes, from [10], the processing steps inside the Samson Box. These steps will be used in the 56000 code as comments. For reasons of hardware implementation which need not concern us here, the Samson Box used quantities with the bit widths shown in the figures. We will be extravagent here and use 24-bit or 48-bit quantities which match or exceed the lengths used in the Samson Box. Where convenient, a long quantity (48 bits) will be divided between lower and upper halves such that the lower half represents a fractional part, and the upper half represents an integer part.

Hardware considerations

The 56000 chip

The internal architecture of the 56000 family is summarized in Figure 3, taken from the manual [11].

Internally, there are three 24-bit buses (x, y, and p for "program") which concern the programmer. These buses connect internal and external memory with the internal address and data ALUs. The data ALU and address ALU may be performing calculations independently and simultaneously while data is being shifted on these three buses. This parallelism is one of the advantages of this particular chip over certain other DSP chips on the market. Experience shows that this parallelism can be exploited in many useful ways by the programmer.

The Address ALU and addressing considerations

Memory is divided (Figure 4) into internal and external portions at address (hex) 1FF in the data memory. (Details of the oscillator parameters will be discussed below). The internal portion is further divided into on-chip RAM and on-chip ROM areas. The upper part of the "external portion" is divided off into "peripheral" memory space, for which there are certain instructions. Many of the "x peripheral" locations are in fact on-chip registers used for i/o. There are thus about 64K external X and Y data locations for parameters and lookup tables.

The memory space can be accessed with absolute addresses, although this is inefficient on the 56000 chip. The address ALU, shown in Figure 5, is broken into two parts, each of which contains a) an ALU and b) banks of three parallel registers. Register R*n* for addressing memory is matched with register N*n* for offsets and increments

larger than 1, and with register M*n* for modulus and bit-reverse (FFT) operations. Both halves of the address ALU may be used simultaneously. The corollary is that there are certain restrictions on which pairs of R*n* may be used for x/y memory accesses during parallel data moves.

The Data ALU

The data ALU is shown in Figure 6. The 48-bit registers x and y are broken into 24-bit halves (x0 and x1, y0 and y1, respectively). The 56-bit accumulator registers a and b are broken into 8-bit extension registers (a2 and b2) along with two 24-bit registers each (a1 and a0, b1 and b0). Any of the registers may be read or written to using either the x or the y bus. A limiter prevents arithmetic overflow on reads from the a and b registers. Simultaneously, the "multiply-accumulate and logical unit" in the figure can perform one integer operation (add, mpy, xor, multiply with round/accumulate, and the like) during each instruction cycle.

Input/output (i/o) considerations

There are three i/o ports ("host", SCI, and SSI) on the chip, the details of which need not concern us here, except to say that the SSI serial interface lends itself well to AES/EBU interfacing, and/or to interconnecting with other 56000 chips. A prototypical hardware architecture is shown in Figure 7. A host CPU updates parameters and code inside the 56000 chip, controls start and stop, and the like. External data and/or program memory is dual-ported. In this paradigm, the CPU fills parameter memory with all startup values, then starts the 56000. The CPU passes updates in real time to the 56000, either on an interrupt basis into the 56000's internal memories, or through the dual-ported external memory. Some handshaking considerations will be covered below. Audio out comes either from the CPU or from the 56000's (SSI) serial port.

Software for the lookup oscillator

Data structures

A data structure for the parameters

Figure 8 shows one solution for packing the parameters into data memory. Here, we exploit the 56000's capability for two-word transfers. Where possible, a double-precision quantity is of course packed into x and y addresses at the same location. The FM and sum memory address entries are pointers into sum memory, the location of which will be discussed in the next section. The exact order of the parameters is optimized to match the code, which will be discussed below. With 7 locations per oscillator, a maximum of 36 oscillators could fit into the 256-word internal data RAM.

Memory map for patching and the lookup tables

Figure 4 shows where the oscillator data can fall. Normal external data memory starts with location 0x1FF, and goes to 0xFFC0 in both X and Y memory spaces. Starting

with 0x1FF, we stack up parameter frames for as many oscillators as are needed. In Y memory, there is room for lookup tables. Above the lookup tables are the sum memory locations. The choice of Y memory for the lookup tables and sum memory locations is again dictated by convenience of addressing in the code, as will be discussed below.

Program memory map and control structure

Figure 9 shows three possible paradigms for embedding the oscillator code into an overall software package. "OVERHEAD" in all three parts of the figure includes the time alloted for changing the parameters in program memory. There must be a careful interlock mechanism so that the parameters for at least the oscillator currently being executed will not be overwritten. The safest method is to allow parameter updates for all oscillators only when OVERHEAD is executing. This has the disadvantage that more updates may need to be handled, on the average, than what OVERHEAD allows.

In Figure 9a), the code is compacted into one central loop, which is repeated *N* times. This has the advantage that it is easy to change the number of oscillators that are needed, simply by modifying the *N* quantity in the code in program memory. None of the parameters is addressed by absolute addresses. During each pass, the addressing registers are incremented to point to the parameters for the next oscillator. Parameter blocks for successive oscillators must thus be next to each other in data memory. Part of OVERHEAD is initializing all of the address registers.

In Figure 9b), the code for the oscillators is repeated *N* times in program memory. *N* oscillators are always performed in each sample time. There is no loop overhead associated with each oscillator, as in Figure 9a), so this code is slightly faster but takes up more program memory space. Parameters fall into contiguous data space, or absolute addressing may be used.

Figure 9c) is the method that was used in the Lucasfilm ASP (also known as the SoundDroid) [12, 13]. Here, the code for each block is followed by a jump instruction. At the end of each sample time, the OVERHEAD block has the option of re-patching by overwriting, in program memory, the JMP instructions. The final block to be executed jumps to OVERHEAD itself. OVERHEAD jumps back "out" to the first oscillator. This scheme can be implemented with pre-assembled macro calls, which allows for absolute addresses in data memory, if that is desired. One advantage of this scheme is that each block can be highly pipelined (an example will be discussed with Figure 11). In the ASP, each block would implement *n* copies of one function to exploit pipelining. The disadvantage of this method is that a memory manager is required that is intelligent enough to 1) keep track of which functions are in which blocks in memory; 2) load a new block when the current block is out of functional units; 3) keep track of the jumps; 4) keep track of any absolute data memory addresses in the (relocatable) program blocks.

Code for the "complicated" oscillator

The code examples given in this preprint are almost completely stand-alone. They follow standard 56000 formats, except that the code fragments are numbered in the left margin to make referencing easier in this preprint. Figure 10 gives code for an oscillator with the full "bells and whistles" described above. The steps given in Figure 2 are used a comments. There is no attempt at optimizing for execution time; a more compact version of the code will be given later. In fact, in this figure, each line of code executes only one step. Experience shows that a good way to write tight code is to first enumerate the steps as is done here.

A walk through the code

Data Structures. Lines 7-10 are labels that correspond to the boundaries in the memory map (see Figure 4). Lines 12-28 specify one oscillator's worth of the parameters, given in Figure 8. The variable summem in line 14 is a pointer to the sum memory block which starts in line 36 of the code. sintab in line 17 is the base of the sine (or lookup) table. The file sintab.asm, not shown here, fills a lookup table starting in line 32. The SETUP of Figure 9 is limited to the initialization in line 42. The quantity 1 in y1 is used for incrementing the current address to get one of the two lookup values needed for interpolation.

Parameters. In this 56000 implementation, freq is a 48-bit quantity. The high-order part (freq hi from x memory in Figure 10) is the right-adjusted "integer" part, up to 16 bits as given by the 16-bit length of the R registers in the 56000's address ALU. The low-order 24 bits contain the "fractional" part of frequency. freqSweep is also right-adjusted.

The FM input is scaled to match the frequency input. That is, the FM input is right-adjusted and scaled such that a modulating sine wave whose maxima reach ±(table length/2) corresponds to a modulator with a deviation of c Hz, where c is the carrier frequency.

In this 56000 implementation, exponent and decay terms are left-adjusted, signed, 24 bits.

Calculating the interpolated sample. Lines 46-50 calculate the frequency-modulated frequency increment. The *address* of the frequency modulation input is read in line 47 into r1, but can't be used (due to 56000 pipelining) until line 49.

By line 56, we have calculated the frequency term in register b as well as stored it in memory for the next sample. When freqSweep is read in line 53, r0 is *decremented* so that freq, the old value of which was read with auto-incrementing in the long move of line 48, can be overwritten in line 55. By (auto-)incrementing and decrementing the address register in this manner, the code can arbitrarily pick up and deposit parameters in the parameter frame of lines 12-28. The extra auto-increment in line 56 may seem extravagent, but experience shows that certain extravagences at an initial coding stage

can be made to disappear when one starts to compact the code, as will be discussed below.

Two (usually adjacent) samples from the lookup table are needed to implement interpolation. In lines 59-61, we start to set up interpolation. R1 will be used to grab the first sample, and r2 the second, so we store the lookup table base address into n1 and n2. Register y0 has the table length (minus 1), for calculating modulus by hand. By the way, there are other possibilities for performing interpolation, such as storing samples in one bank of memory and a parallel set of differences in another bank. Also, if we're dealing only with sine waves, there are other methods of generating sines besides table lookup.

Anyway, in lines 64-65 we calculate the new position inside the lookup table. In particular, we now need to examine what happens when the incremented lookup position falls off the end of the table in memory. Nominally we would like to store the base address of the lookup table in register N*n* and the length of the table (minus 1) in register M*n*; then R*n* would contain the offset. One problem is that there is no easy way to divide the address register into integer and fractional parts, so that it becomes impossible to preserve fractional information for doing interpolation later. The other sticky part is that the increment can be negative, especially in FM [7], which means that the lookup pointer can fall backwards off the *beginning* of the table. The modulus feature of the on-chip address ALU does not handle this possibility.

Therefore, we have to handle the lookup table boundaries by hand. Note that this code assumes that exactly one copy of the wavetable is stored in memory. As Bernard Mont-Reynaud pointed out in a discussion at CCRMA, one could have a partial copy of the wavetable duplicated before and after a central, complete copy, which might simplify the code in some architectures. For example, if one calculates the modulus explicitly for the first of the two interpolated samples, then the second sample could be guaranteed to always fit into the expanded table. In the code, then, line 68 tests whether the updated angle is now less than 0. If so, lines 69-70 increment the angle by the length of the table. Note that the quantity in y0 is the highest-order address in the table; to get the table *length*, we need to add in the quantity 1 in register y1, in line 70. Note also there there is a "feature" in line 69---if you've specified a negative increment whose absolute value is larger than the table length, then line 69 won't add in enough correction. The "and" in line 71 takes care of falling off the end of the table, and also takes care of a chip "feature"---if, during the additions in lines 65, 69, and/or 70, overflow occurs, then limiting, which we *don't* want, will happen during the move in line 72.

When the address for the second sample (for interpolation) is calculated in lines 76-78, it's not necessary to worry about underflow.

Note that the updated angle in line 65 is a long quantity stored in accumulator a. The operations on accumulator a in lines 68-78 affect only a2 and a1; the lower-order part in a0, which will give the "fraction" used in interpolation, remains unchanged. That fractional part is stored into x0 in line 82, to be used in the macr instruction in line 88.

Lines 83 and 84 use the base address of the table, stored way back in lines 59 and 60, to get the two samples for interpolation. Lines 85-86 calculate the difference between the two samples, which must be in a register such as y0 for multiplication. The first sample is needed again for line 88. An alternative, when the code is compacted, is to do a copy or register-to-register transfer from register x1 to register b, somewhere in lines 84-87. The interpolated sample is stored in register x1 in line 89 to prepare for multiplication by the amplitude.

The amplitude envelope. The updated amplitude envelope, calculated in lines 92, 95, and 97, must be limited to avoid numeric overflow and underflow. Fortunately, the 56000's ALU handles the numeric overflow automatically [11, section 3.3.6]. Of the several ways to test for underflow, I chose to clear accumulator b in line 96 and then to test for "less than 0" in line 98. With the tlt instruction, if (in this case) the result in accumulator a is less than 0, then the quantity in accumulator b is shifted into accumulator a. The clr instruction in line 96 will allow for a double move when the code is compacted, so this will be an efficient way to get 0 into the accumulator.

When the asymptote term is added in line 104, the result must also be limited for arithmetic overflow; and the 56000 performs this for free in the move from register a to register x0 (line 105).

In line 108 we finally multiply the interpolated sample by its envelope. Remember that in this particular architecture, the oscillator output is first right-shifted, then summed with whatever is in sum memory. For fun, we can also round up the sample multiplied by its envelope. An easy way to do the rounding is to add a 1 into the low-order bit of the enveloped sample, then to right-shift the sum. This happens in lines 109-110. In lines 112-116, the new sample is summed and written out to sum memory. Again, the nop of line 113, needed for the address ALU in the 56000 chip, will disappear when the code is compacted.

Compacting 56000 code

On the 56000, the main vehicle for compacting code is to exploit the parallel move facility. As discussed above, when the data ALU is performing some operation, the X and Y data busses (Figure 3) can be moving data between memory and the data ALU (certain penalties will be discussed at the end of this section). In some cases, data may be moved between data ALU or address ALU registers as well. Furthermore, the address registers may be (auto-)incremented. Since all these registers latch their data, a datum may be fetched and stored long before it is needed, if the fetch can be hidden in an otherwise unused parallel move slot. The limit on compacting code is ultimately the number of data ALU operations to be performed. Experience has shown that with careful hand-coding with real-world audio and musical algorithms, the length of the code can be brought very close to this limit.

A compacted oscillator

Since the pipelined and compacted code of Figure 11 follows closely the code in Fig-

ure 10, let us limit this discussion to giving examples of the techniques mentioned in the last paragraph. The memory maps and data structures remain unchanged, so they are omitted here.

In lines 59-60 of Figure 10, we grab the base address of the lookup table twice. In Figure 11, this quantity is first loaded into n1 in line 23. Loading into n2 is hidden in the add instruction in line 32, with a register-to-register transfer inside the address ALU. As in Figure 10, neither of these registers is used until lines 48 and 49 of Figure 11.

Stepping backwards and forwards in the parameter frame is more complicated in Figure 11. The quantity 2 has now been stored in register n0 (line 10). In line 23 of Figure 11, r0 is pointing at the base of the table (line 17 in the parameter frame of Figure 10). With the move in line 23, we decrement r0 by the quantity 2 in register n0, so that r0 now points back at freq (line 15 in the parameter frame of Figure 10). In the very next instruction (Figure 11, line 25), we write out the updated frequency value and increment r0 by the 2 in n0, so that r0 is again correctly pointing at (table length -1), which will be read in the next instruction (line 28). By storing 2 in n0, we can eliminate the double (r0)+ in lines 55 and 56 of Figure 10. The other subtle difference is that r0 must be incremented in line 21 of Figure 11, in order for the jump back by 2 to work, whereas it was *decremented* in line 53 of Figure 10.

There are many examples in Figure 11 where a single move happens during an ALU operation. Line 57 shows a double move. This line combines lines 89, 92, and 96 of Figure 10. Note that register x1 in line 89 of that figure has now been replaced by y0 in Figure 11. This kind of switch is frequent when one is compacting code.

A register-to-register data ALU move in line 50 of Figure 11 eliminates the need for the memory access in line 87 of Figure 10.

In order to tighten the code even further, the instructions of lines 47-49 of Figure 10 are pipelined. Register r1 of Figure 10 is replaced by register r6, due to the 56000's rules governing which address ALU registers can be used simultaneously. The three instructions are first used at lines 15-17 in Figure 11, before the first pass through the inner loop. During that first pass, the initialization for the *second* oscillator is hidden at the bottom of the figure. Line 15 is hidden in line 76; line 16 in line 79; and line 17 in line 81.

To compress this code even further, the compulsive programmer will look for 1) more possibilities for pipelining and 2) improvements to be gained by rearranging the parameter frame, with corresponding changes in the order of the rough blocks of code.

A word of warning about a quirk of the 56000 is appropriate here. Note that there is a penalty for doing x/y or long moves from external memory when the program is also in external memory. Of the three memory cycles available (x, y, and/or program), two may be external "for free." But if all three items come from external memory, then an additional one-cycle pipelining penalty is imposed.

A simple oscillator

Many of the luxuries in the code discussed thus far can be dispensed with for a "bare-bones" oscillator, such as the one shown in Figure 12. Here we take advantage of the sine-wave lookup table available starting at y:0 in the 56001 chip (line 8), with length of 256 parameterized as #tab_len. So this oscillator is just a sine oscillator, not a generalized sampling oscillator. The parameter frame in lines 14-23 show that the time-varying parts of the frequency and amplitude envelopes have been removed. Now there are freq and amplitude inputs which must be changed elsewhere when it is time for them to change. The FM input remains, and it continues to be read on each sample.

In the setup code, tab_len is stored into y0 (line 28). In the earlier code, a different tab_len was read in for each oscillator, but here only one table is used for all oscillators. The base address of that one sine table is stored into n1 forever as well (line 29).

We further assume that this oscillator is the only thing being executed in the chip. Thus, start in line 32 goes to the beginning of the parameter block for every sample. Note that the code given here is as uncompacted as the code in Figure 10; presumably the nop in line 32 in Figure 12 can be removed during compaction.

The blocks of code should now look familiar, but their order has been rearranged. Lines 56 and 59 are the major difference. Instead of calculating an updated amplitude envelope, we multiply the sample by the amplitude value in the parameter frame. Note that no interpolation or roundoff has been done, and the right-shift by 1 is omitted before summing into sum memory.

Conclusion: How many oscillators on one chip?

Suppose that we're operating at a 50 kHz sample rate. The current instruction cycle time of the chip is 95 nsec, although this time is expected to go down. That means 210 instructions per sample.

The 56000 software simulator reports that the simple oscillator of Figure 12 requires 16 instruction cycles, which would allow for an absolute maximum of 13 oscillators per chip.

For the compacted oscillator, with internal program memory and external x and y data memories, the 56000 software simulator reports that one sample takes 37 instruction cycles, which would allow a maximum of 5 oscillators per chip. As a limiting case, if we take just the arithmetic operations in the complicated oscillator, there are 21 instruction cycles. At 50 kHz, that would result in 9 or possibly 10 oscillators per sample. (Note that the limit in the chip is thus given by the execution time, at least at this sample rate, and not by the length of the parameter frame in Figure 8, of which 36 could be fit into internal RAM). Therefore, I would guesstimate that this chip can generate 8 musical oscillators in real time at 50 kHz; information from others in the music industry working with this chip shows that this estimate is reasonable.

Acknowledgments and Disclaimer

I am grateful to Yamaha Music Technologies for making it possible for me to appear at this AES Convention. The work in this paper was inspired by a request from Max Mathews to give a talk on this topic at CCRMA, Stanford University. This work was completed long before my employment at Yamaha Music Technologies USA and was supported by none of my consulting clients at that time. This work is entirely my own and I am solely responsible for errors, omissions, and self-deceptions.

References

[1]. Chrysafis, Andreas. *Digital sine-wave synthesis using the DSP56001.* Motorola application note APR1, 1987.

[2]. *Precision digital sine-wave generation with the TMS32010.* Texas Instruments application note SPRA007, 1984.

[3]. Snell, John. "Professional real-time signal processor for synthesis, sampling, mixing, and recording." AES preprint 2508 (M-4), 1987 New York convention.

[4]. Hartmann, W. M. "Digital waveform generation by fractional addressing." *Journal of the Acoustical Society of America* 82(6):1883-91, 1987.

[5]. Mehrgardt, S. "Noise spectra of digital sine generators using the table-lookup method." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 31:1037-39, August 1983.

[6]. Moore, F. Richard. "Table Lookup Noise for Sinusoidal Digital Oscillators." *Computer Music Journal* 1(2):26-29, 1977. Reprinted in Curtis Roads and John Strawn, eds. *Foundations of Computer Music.* Cambridge, Massachusetts: MIT Press, 1985, pp. 326-334.

[7]. Chowning, John M. "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation." *Journal of the Audio Engineering Society* 21(7):526-34, 1973. Reprinted in Curtis Roads and John Strawn, eds. *Foundations of Computer Music.* Cambridge, Massachusetts: MIT Press, 1985.

[8]. Samson, Peter R. "A General-Purpose Digital Synthesizer." *Journal of the Audio Engineering Society* 28(3):106-13, 1980.

[9]. Samson, Peter R. "Architectural issues in the design of the Systems Concepts digital synthesizer." In John Strawn, ed. *Digital Audio Engineering: An Anthology.* Los Altos, CA: Kaufmann, 1985, pp. 61-94.

[10]. "Systems Concepts digital synthesizer programming specification." San Francisco, CA: Systems Concepts, Inc., version of 9/16/77. Manuscript.

[11]. Motorola. *DSP56000 Digital Signal Processor User's Manual.* Version 1.1, 1987.

[12]. Moorer, James A. "The Lucasfilm Digital Audio Facility." In John Strawn, ed. *Digital Audio Engineering: An Anthology.* Los Altos, CA: Kaufmann, 1985, pp. 95-135.

[13]. Abbott, Curtis. "System level software for the Lucasfilm ASP System." San Rafael, CA: Lucasfilm, Technical Memo #58, August 1982.

Figure 1. Samson Box sine-wave oscillator with linear envelope (from [10]).

name from [10]	width (bits)	name used in this article	function
GO	20	freqSweep	frequency sweep rate
GJ	28	freq	oscillator frequency
GK	20	angle	oscillator angle
GP	20	decay	decay rate
GQ	24	exponent	decay exponent
GL	12	asymptote	
GSUM		adr	sum memory output address
GFM		FMadr	sum memory FM input address

Figure 2. Steps in the Samson Box oscillator. The terminology and usage of temp variables is taken from [10]. The numbers at the left refer to the steps in [10], some of which are intentionally omitted here.

1.	Temp0	<-	20 bits from mem[FMadr] + high-order 20 bits of freq
2.	freq	<-	freq + freqSweep, right-adjusted and sign-extended
4.	angle	<-	angle + Temp0
8.	Temp5	<-	sin(angle)
10.	Temp7	<-	high-order 12 bits of exponent
11.	decay	<-	exponent + right-adjusted decay
13.	Temp8	<-	asymptote + decay
14.	Temp9	<-	Temp8 * Temp5 sum right-shifted high-order 19 bits into sum memory

Figure 3. The 56000 architecture (from [11]).

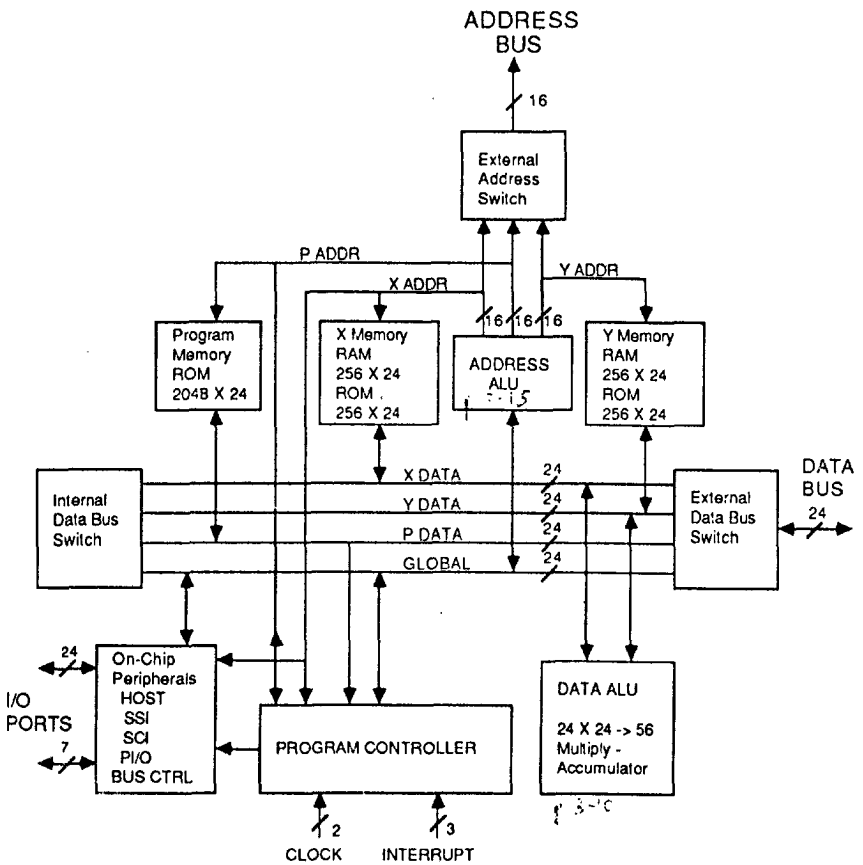


Figure 4. DSP56000 Memory Map (after [11], Fig. 9.1.2) with parameters for wavetable lookup oscillators.

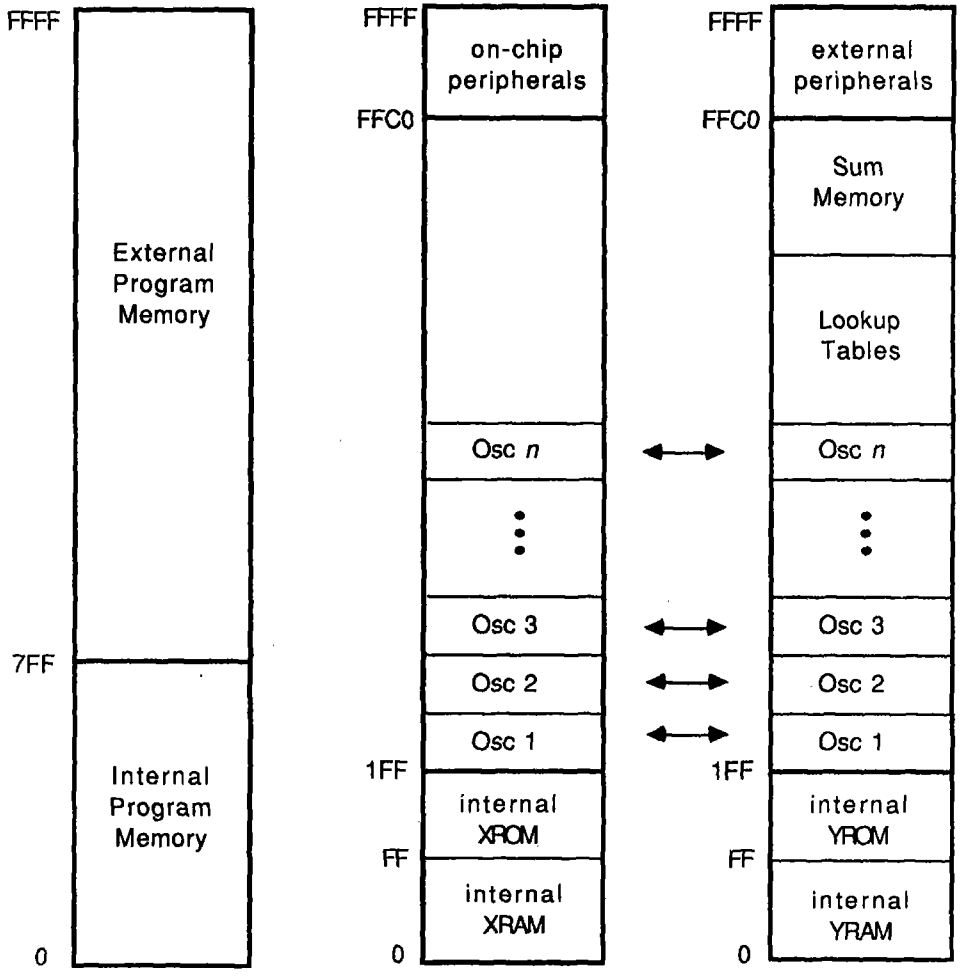


Figure 5. The 56000 address ALU (from [11]).

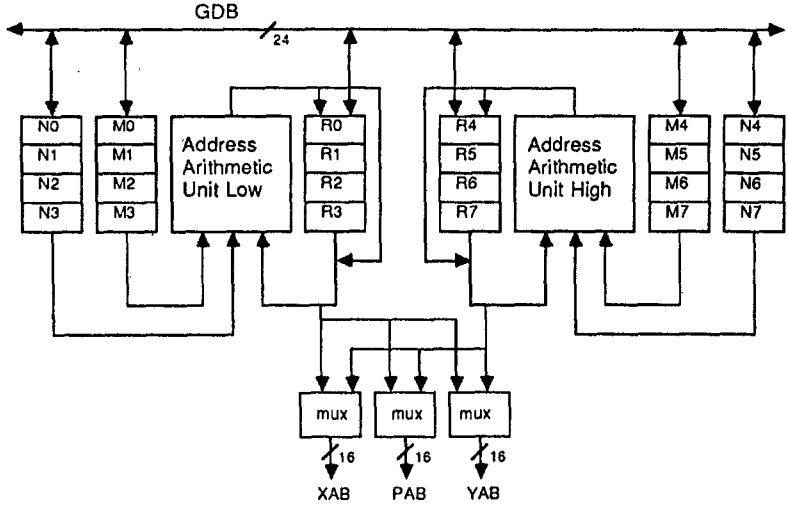
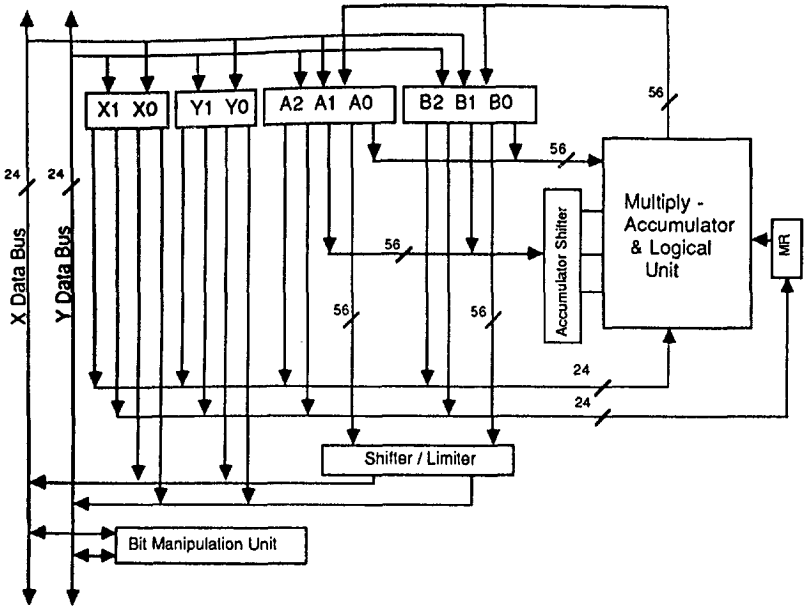


Figure 6. The 56000 data ALU (from [11]).



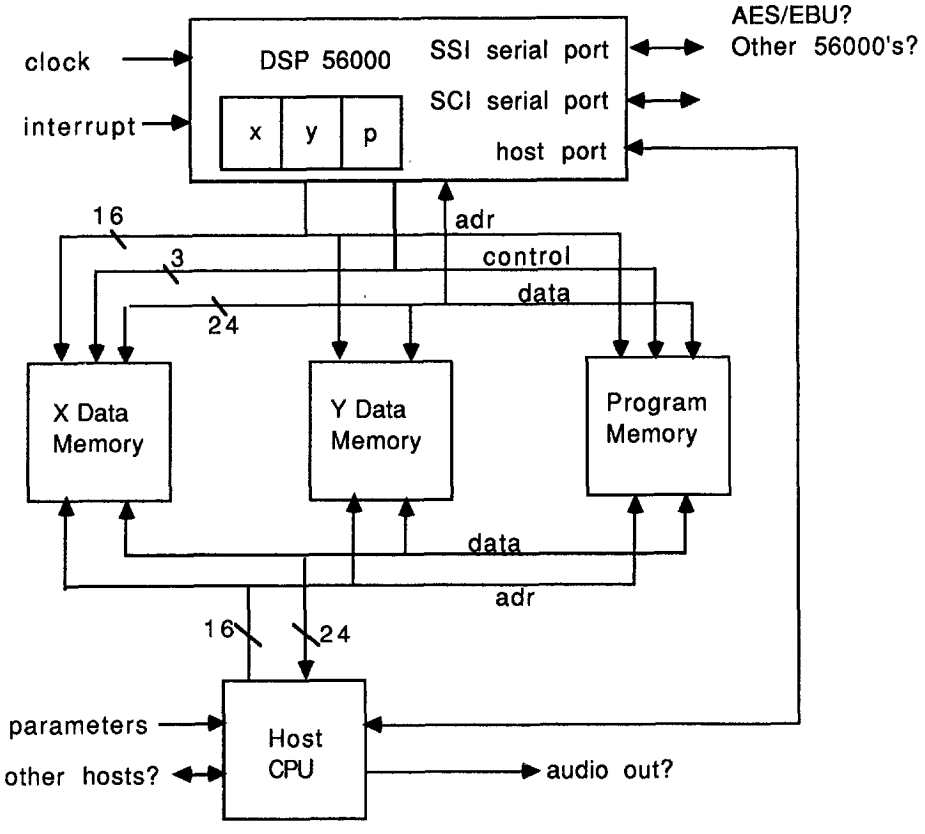


Figure 7. Prototypical hardware architecture with host, 56000, and external memory.

Figure 8. Parameter data structure

Parameter no.	x data memory	y data memory
1.	FMadr	empty
2.	freq hi	freq lo
3.	freqSweep hi	freqSweep lo
4.	base of table	table length
5.	angle hi	angle lo
6.	exponent	decay
7.	asymptote	sum memory address

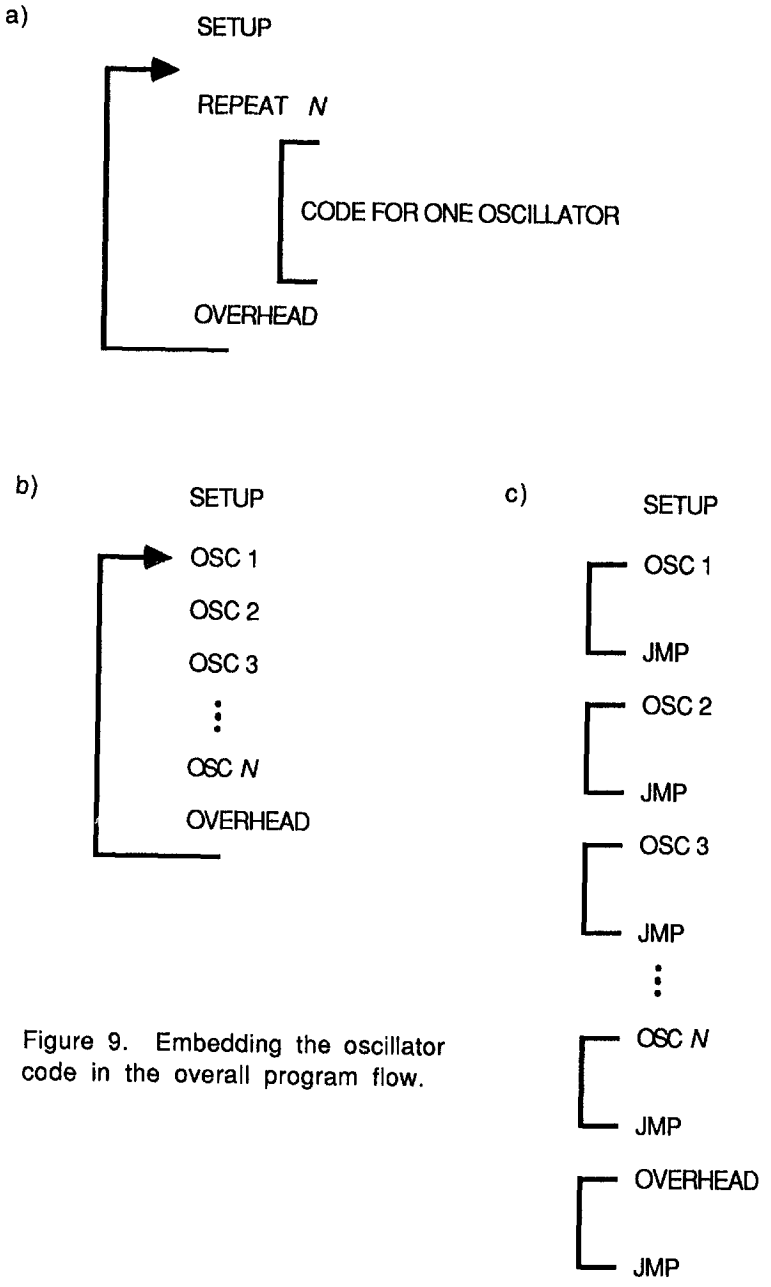


Figure 9. Embedding the oscillator code in the overall program flow.

```
1   ;; Figure 10. Interpolating "sampler"/sine wave oscillator.
2   ;; DRAFT---uncompacted version
3   ;; John Strawn, S Systems
4   ;; 2/9/88
5   ;; Copyright S Systems
6
7   lookup_org  set             $3000    ; y external
8   summem_org  set             $2000    ; y external
9   param_org   set             $200     ; x and y external
10  prog_org    set             $0       ; p
11
12      ; parameters for one oscillator
13      org     x:param_org
14      dc summem+2         ; FM input
15      dc $20              ; freq hi
16      dc 0                ; freqSweep hi
17      dc sintab           ; base of table
18      dc 0                ; angle hi
19      dc 0                ; exponent
20      dc $7fffff          ; asymptote
21      org     y:param_org
22      dc 0                ; empty
23      dc 0                ; freq lo
24      dc 0                ; freqSweep lo
25      dc tab_len-1        ; table length - 1, used as mask
26      dc 0                ; angle lo
27      dc 0                ; decay
28      dc summem+1         ; sum memory address
29
30      org     Y:lookup_org    ; lookup table
31  tab_len     set 256
32      include 'sintab'        ; has label sintab in sintab.asm
33
34      org     Y:summem_org    ; sum memory locations
35  summem_len set      16
36  summem      dup summem_len
37      dc      0
38      endm
39
40      org     P:prog_org
41      ; setup
42      move    #>1,y1          ; increment of 1 stays here forever
43  start       move    #param_org,r0
44      nop
45
46  ; 1.  Temp0 <- mem[FMadr] + freq
47      move    x:(r0)+,r1      ; get FMadr
48      move    l:(r0)+,x       ; get freq
49      move    y:(r1),a        ; get mem[FMadr]
50      add     x,a             ; Temp0 in a
51
52  ; 2.  freq <- freq + freqSweep
53      move    l:(r0)-,b       ; get freqSweep
54      add     x,b             ; freq + freqSweep
55      move    b,l:(r0)+       ; update freq
56      move    (r0)+
57
58  ; set up addressing (need r0 locally for get, update angle below)
59      move    x:(r0),n1       ; base address into n1
60      move    x:(r0),n2       ; base address into n2
61      move    y:(r0)+,y0      ; (table length - 1) into y0
62
63  ; 4.  angle <- angle + Temp0
64      move    l:(r0),x        ; get angle
```

Figure 10 (continued)

```
65      add     x,a             ; angle + Temp0 in a
66
67  ; create modulus for address
68      jge     posFreq         ; if angle now < 0,
69      add     y0,a            ;       jump ahead one table's length
70      add     y1,a
71  posFreq and y0,a            ; y0 has mask for length of lookup table
72      move    a,l:(r0)+       ; store updated angle
73      move    a1,r1           ; integer portion of angle to r1
74
75  ; create modulus for address+1 (for interpolation)
76      add     y1,a
77      and     y0,a
78      move    a1,r2
79
80  ; 8.  Temp5 <- sin(angle)
81  ;           with interpolation
82      move    a0,x0           ; fractional part of address in x0
83      move    y:(r1+n1),x1    ; get first sample
84      move    y:(r2+n2),a     ; get second sample
85      sub     x1,a            ; get sample slope
86      move    a,y0
87      move    y:(r1+n1),b     ; get first sample again
88      macr    x0,y0,b         ; interpolated Temp5 in b
89      move    b,x1            ; Temp5 in x1 for later multiply
90
91  ; 10. Temp7 <- exponent
92      move    x:(r0),a
93
94  ; 11. decay <- exponent + decay
95      move    y:(r0),x0       ; get decay
96      clr     b               ; 0 into b for tlt
97      add     x0,a
98      tlt     b,a             ; a <- a MAX 0
99                              ;    (overflow handled automatically by chip)
100     move    a,y:(r0)+       ; store updated decay, with automatic limiting
101
102 ; 13.  Temp8 <- asymptote + decay
103     move    x:(r0),x0       ; get asymptote
104     add     x0,a            ; Temp8 in a
105     move    a,x0            ; Temp8 in x0, again with limiting
106 ;
107 ; 14.  Temp9 <- Temp8 * Temp5
108     mpy     x0,x1,a
109     add     y1,a            ; round (the 1 from y1 will become .5 after shift)
110     asr     a               ; right-shift by 1 bit with sign extension
111 ;   sum into sum memory
112     move    y:(r0)+,r1      ; get sum memory address
113     nop
114     move    y:(r1),x1       ; get old value from sum memory
115     add     x1,a
116     move    a,y:(r1)        ; write out to sum memory
117
118     jmp     start
```

```
;; Figure 11. Interpolating "sampler"/sine wave oscillator.
;; DRAFT---compacted version
;; John Strawn, S Systems
;; 2/10/88
;; Copyright S Systems

    org     P:prog_org
    ; setup
    move    #>1,y1          ; increment of 1 stays here forever
    move    #>2,n0          ; for incrementing r0 by 2 sometimes
    move    #param_org,r0
    nop
    ; pipelining initialization
; 1.  Temp0 <- 20 bits from mem[FMadr] + high-order 20 bits of freq
    move    x:(r0)+,r6      ; get FMadr
    move    l:(r0)+,x       ; get freq
    move    y:(r6),a        ; get mem[FMadr]

; main loop starts here
; 2.  freq <- freq + freqSweep, right-adjusted and sign-extended
start       add     x,a     l:(r0)+,b       ; Temp0 in a
                                            ; get freqSweep in b
    add     x,b     x:(r0)-n0,n1    ; freq + freqSweep
                                            ; base address into n1
    move    b,l:(r0)+n0     ; update freq, skip over freqSweep

; set up more addressing
    move    y:(r0)+,y0      ; table length into y0

; 4.  angle <- angle + Temp0
    move    l:(r0),x        ; get angle
    add     x,a     n1,n2   ; angle + Temp0 in a
                            ; base address into n2

; create modulus for address
    jge     posFreq         ; if angle now < 0,
    add     y0,a            ;       jump ahead one table's length
    add     y1,a
posFreq and y0,a            ; y0 has mask for length of lookup table

; create modulus for address+1 (for interpolation)
    add     y1,a    a,l:(r0)        ; store updated angle
    and     y0,a    x:(r0),r1       ; integer portion of angle to r1
    move    a1,r2

; 8.  Temp5 <- sin(angle)
;           with interpolation
    move    y:(r1+n1),y0            ; get first sample
    move    y:(r2+n2),a             ; get second sample
    sub     y0,a    y0,b            ; get sample slope in a
                                    ; get first sample again in b
    move    a,x1    y:(r0)+,y0      ; sample slope to x1 for multiply
                                    ; fractional part of address in y0
    macr    x1,y0,b y:(r0),x0       ; interpolated Temp5 in b
                                    ; get decay in x0
; 10. Temp7 <- high-order 12 bits of exponent
    clr b   x:(r0),a        b,y0    ; Temp5 in y0 for later multiply
                                    ; exponent in a
                                    ; b <- 0 for tlt below

; 11. add exponent + right-adjusted decay
    add     x0,a
;   overflow taken care of automatically by chip, but we do <0 by hand:
    tlt     b,a             ; a <- a MAX 0
```

Figure 11 (continued)

```
65      move    a,y:(r0)+       ; store updated decay, with automatic limiting
66
67  ; 13.  Temp8 <- asymptote + Temp7
68      move    x:(r0),x0               ; get asymptote
69      add     x0,a    y:(r0)+,r5      ; Temp8 in a
70                                      ; get sum memory address in r1
71      move    a,x0                    ; Temp8 in x0, again with limiting
72
73  ; 14.  Temp9 <- Temp8 * Temp5
74  ;   right-shift by 1 bit with sign extension
75      mpy     x0,y0,b         y:(r5),a ; get old value from sum memory
76      add     y1,b    x:(r0)+,r6      ; round (the 1 from y1 will become
77                                      ;     0.5 after shift)
78                                      ; get FMadr for next oscillator
79      asr     b       l:(r0)+,x       ; get freq for next oscillator
80  ;   sum right-shifted high-order 19 bits  into sum memory
81      add     a,b     y:(r6),a        ; get mem[FMadr] for next oscillator
82      move    b,y:(r5)                ; write out to sum memory
83      jmp     start
```

```
1   ;; Figure 12. Simple non-interpolating FM sine-wave oscillator.
2   ;; No amplitude envelope, frequency envelope.
3   ;; DRAFT---uncompacted version
4   ;; John Strawn, S Systems
5   ;; 2/9/88
6   ;; Copyright S Systems
7
8   lookup_org  set             $0      ; y
9   summem_org  set             $2000   ; y
10  param_org   set             $3000   ; x and y
11  prog_org    set             $0      ; p
12
13      ; parameters for one oscillator
14      org     x:param_org
15      dc summem+2     ; FM input
16      dc $20          ; freq hi
17      dc 0            ; angle hi
18      dc $7fffff      ; amplitude
19      org     y:param_org
20      dc 0            ; empty
21      dc 0            ; freq lo
22      dc 0            ; angle lo
23      dc summem+1     ; sum memory address
24
25      org     P:prog_org
26      ; setup
27      move    #>1,y1          ; increment of 1 stays here forever
28      move    #>tab_len-1,y0  ; table length stays here forever
29      move    #>sintab,n1
30
31  start       move    #param_org,r0
32      nop
33
34  ; 1.  Temp0 <- mem[FMadr] + freq
35      move    x:(r0)+,r1      ; get FMadr
36      move    l:(r0)+,x       ; get freq
37      move    y:(r1),a        ; get mem[FMadr]
38      add     x,a             ; Temp0 in a
39
40  ; 4.  angle <- angle + Temp0
41      move    l:(r0),x        ; get angle
42      add     x,a             ; angle + Temp0 in a
43
44  ; create modulus for address
45      jge     posFreq         ; if angle now < 0,
46      add     y0,a            ;       jump ahead one table's length
47      add     y1,a
48  posFreq and y0,a            ; y0 has mask for length of lookup table
49      move    a,l:(r0)+       ; store updated angle
50      move    a1,r1           ; integer portion of angle to r1
51
52  ; 8.  Temp5 <- sin(angle)
53      move    y:(r1+n1),x1    ; get sample
54
55  ; 13.  Temp8 <- envelope
56      move    x:(r0),x0       ; get amplitude
57  ;
58  ; 14.  Temp9 <- Temp8 * Temp5
59      mpy     x0,x1,a         ; perform amplitude envelope
60  ;   sum into sum memory
61      move    y:(r0)+,r1      ; get sum memory address
62      nop
63      move    y:(r1),x1       ; get old value from sum memory
64      add     x1,a
65      move    a,y:(r1)        ; write out to sum memory
66
67      jmp     start
```