ALGORITHMIC MUSIC LANGUAGE
Raymond F. Jurgen
Electronic Arts Research
Box 746
Monterey, CA
93940

Introduction

Algorithmic Music Language (AML) has been developed over several years to provide an advanced microcomputer language for the control of analog synthesizers. The language is especially well suited to the realization of "process music" or any music for which the organization and structure can be analyzed and reduced to simpler structures. The language system operates on an 8080 or 280 microprocessor system and employs the Digital Research CP/M operating system. The software system consists of a machine emulator and an assembler/compiler to generate object code for the emulator. Presently there exists two versions of the emulator: the first version uses fixed length five-byte instructions, and the second version uses variable length instructions, many of which require only one byte. The second version also contains a small operating system designed to provide interactive control of the emulator. Presently, the assembler/compiler for version 2.0 is incomplete, and little experience has been gained using version 1.0. Therefore, this paper will be restricted to discussion of the features and operation of version 1.0, which is designated as AML-1.

Microcomputer systems have more severe limitations than the larger, more expensive machines. In particular, in low cost computers, the amount of memory available is fairly restricted and the speed of execution of certain arithmetic instructions may be slower than desired. Because of these limitations, it is very difficult to obtain professional sound electronic music synthesis using the presently available microcomputers, thus, hybrid systems are still favored. Many fine analog synthesizers are available on the market, and many studios are equipped with these. A low cost computer control system can provide a great extension in the types of music that can be executed on small systems as will be seen in a few taped examples. Although the data control rate is
minimized by hybrid implementations, the memory limitations may still pose a severe constraint. The use of algorithmic structures in music in place of truly expandable command instructions can often reduce the memory requirements significantly; factors of ten are common. The simplest example of such a structure is the repeat loop. AML provides the composer with a full retinue of control structures as well as the ability to arrange stochastic events in real time.

AML Emulator

The emulator portion of the AML system is a software realization of a machine designed to command one or more analog synthesizers via digital to analog interfaces. Most synthesizers require the standard one-volt/octave control of frequency, a gate signal to play the note for a given duration, and a signal for the control of loudness. AML provides each voice with these signals; however, these signals do not have to be used for their intended purposes. For example, the gate control could advance a slide projector and the loudness could control the brightness.

In AML each "voice" is assigned a pseudo-machine. The number of pseudo-machines is presently limited to eight, and the first hardware realization supports only four. Machines not supported by hardware can be used to cause certain modifications of those that are, if desired. Thus complicated computations can be distributed among the pseudo-machines to simplify the calculation of several simultaneously changing events.

The AML emulator operates in two modes which can be intermixed at will. Mode A was designed for the standard 12 tone system with each note specified by a five-byte code. Mode B permits computation of notes of any required pitch and duration during run time.

The operation of Mode A is the simplest to understand and can be hand assembled and loaded into memory by the most primitive computer operating systems. The entire emulator for version 1.0 fits into less than two thousand bytes of memory.

The byte format for each note specified in mode A is shown in Figure 1. The first byte specifies the note duration in clock counts, and any number from 1 to 144 is legal. The second byte specifies the gate duration, and any number from 0 to the full value of the duration counts is legal. The gate count provides some special features; if its value is zero, a rest is specified.
<table>
<thead>
<tr>
<th>BYTE 1</th>
<th>BYTE 2</th>
<th>BYTE 3</th>
<th>BYTE 4</th>
<th>BYTE 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>DURATION</td>
<td>GATE</td>
<td>PITCH</td>
<td>OCTAVE</td>
<td>LOUDNESS</td>
</tr>
<tr>
<td>0-144</td>
<td>0-144</td>
<td>0-144</td>
<td>0-11</td>
<td>0-9</td>
</tr>
</tbody>
</table>

Figure 1
Memory Layout for notes specified directly in the Mode B procedure.

and if its value equals the duration count, a tie to the next note is set (i.e., the gate signal is not reset). Notes of any desired length can be set using the tie. The third and fourth bytes specify pitch in a note-octave system.
Pitches are specified by numbers 0-11 in the third byte where 0 corresponds to C natural and 11 to B natural. The octave is specified by numbers 0-9 where 5 indicates the middle octave. The proper 16-bit digital number used to specify the frequency is computed by the table look-up by the emulator. It is this feature that makes
assembly of this note relatively easy. The fifth field is used to specify loudness, but is often used for other purposes such as the control of the filter cutoff frequency of a tremolo or vibrato oscillator. Field five permits values from 0 to 255 and therefore requires only an 8-bit DAC for the hardware implementation. The pitches, on the other hand, are specified by a 16-bit (two-byte) field and may be set to either a 12-bit or 16-bit DAC. Clearly the most straightforward way to specify a range of music is to code each note directly in the above notation. Several 3-and 4-part pieces by J. S. Bach and A. Corelli have been encoded in just this manner without the aid of the assembler/compiler programs.

Mode B provides the power to efficiently generate vast quantities of notes with relatively few instructions.
In this system the notes may be computed or stochastically generated according to the user's algorithms. Pitches may be set to any frequency within the resolution of 12 or 16-bit DAC's. Writing efficient code in this system is somewhat more difficult in that it involves the analysis of the composition to determine repeated material, simple transpositions, or other computable alterations to the basic
lines. Also efficient use of Mode B requires some understanding of the operation of the emulator. Thus we will begin there.

Direct notes are specified when the first byte of the five byte instruction field contains a value from 1 to 144 inclusive. The maximum count of 144 was chosen because it is 127 and is rich in divisions by 2 and 3. Values outside of this range are interpreted as instructions in pseudo-machine code and are executed without counting time. Several instructions of this type can follow in sequence until a note is computed and ready to be played. Notes are set up by passing the results of computations to the note buffers. The note buffers contain the various parameters describing how a note is to be played as well as other data. For example, two bytes in the note buffer are used as counters for the duration and gate length of the note. One special instruction, PLAY, is used to exit the emulation mode and either pass control to the next voice or to the timer.

While in the computational mode, the pseudo-machine acts as a single stack oriented machine. The stack is used to handle program control as well as providing a place to carry out computation. Control instructions that use the stack are REPEAT, END, CALL, and RETURN. These instructions are always used in pairs. Because of the stack implementation, repeats can be nested to any reasonable depth, and subroutines can call subroutines.

Each pseudo-machine has instructions to carry out arithmetical and logical operations on data contained in its stack. The arithmetical operations are full 16 bit two's complement operations. The stack of each pseudo-machine is implemented as a push-down stack, i.e., new elements pushed on to the stack go to consecutively lower memory locations, and the stack pointer is decreased by one word count each time. Pop operations remove words from the stack. Dyadic operations combine the top two elements on the stack and leave the result in place of the next element down as shown in Figure 2. The stack pointed points to the result. Monadic operations modify the top element of the stack and the stack pointer remains unchanged. Since the stack is used by the control instructions, the user is responsible for clearing off any calculations that have been set up, i.e., the stack pointer that is returned to the same location it was prior to the computation.

An important feature of the pseudo-machines is that they employ relative addressing for all control instructions, therefore object modules can be executed anywhere in memory without modifications. And because of the stack-oriented architecture, subroutines can be made re-entrant, i.e., a subroutine may call itself, or may be used by several pseudo-.
machines at one time. Some of these features are best illustrated by examples, and the examples are most easily discussed in the last section which describes the assembler-compiler. A more detailed understanding of the operation of the emulator can be acquired from a sequence of block diagrams.

The basic operation of the AML emulator is shown in Figure 3 where the major blocks called Initializer and Play are shown. The initialization establishes the starting location for the object code for each voice, clears the triggers, initialises the digital to analog converters (DAC's), and clears the note count registers so that they immediately demand new information upon entering the PLAY block.

In the PLAY block, a voice counter is used to scan through each pseudo-machine. The pseudo-machine can either count down the duration and gate count or demand to read new information. If an RNSW (end of voice) code is read by any pseudo-machine, control is transferred out of the play block and a test is made to determine if the infinite repeat flag is set. If not, the program returns to the monitor.

Figure 4 shows a detailed diagram of the operation of the PLAY block. The voice scan operation begins at the largest assigned voice and works down to the last. Voices may be skipped as indicated by the skip control flags. The next test determines whether the last operation has been completed by checking the duration counter. If it is zero, a transfer to the fetch-new-instruction routine is executed, otherwise the duration count and gate count are decreased, and the gate control signals are set appropriately. Following this, the next lower pseudo-machine is processed. When the last machine is processed, the timing routine is entered. This routine is a simple software loop that simply burns time. Hardware timing can be accomplished in several other ways if desired.
Figure 2
Operation of the pseudo-machine stack before and after the execution of the add instruction. Note that SP = SP+1 after the operation.

Figure 3
primitive block diagram of major subsystems of the AML Simulator.
Figure 4

Basic NPL Machine Architecture showing important blocks and process flow from one machine to the next until the timer is encountered.
Figure 5
Detail of the Instruction Fetching and processing operations

344
Figure 6 shows a block diagram of the fetch-new-instruction processor. Here, each machine has its own program counter, stack pointer, and stack. Each pseudo-machine can communicate with each other in common memory. Common memory must be used to communicate information between the pseudo-machine, since no other method has been provided. Once a new instruction is read, the first byte is tested to see if it is of the Mode A (direct mode specification) or Mode B (calculation mode) type. In Mode A the next buffer is set directly, and the 4 and 12-bit DAC's are set just prior to the exit from the routine. If a Mode B instruction is encountered, the instruction is processed unless it is the PLAY instruction, which directs an exit through the set-DAC's block. Otherwise, the program loops back to the beginning of the fetch-new-instruction routine. This looping can continue until either the PLAY instruction is encountered or until a Mode A instruction is encountered.

The power of the pseudo-machine depends upon its instructions. Appendix A shows that the instructions fall into nine classes. Program control instructions are SEND (end of voice marker), REPEAT (repeat following instructions), ENDR (end of repeated sequence marker), and unconditional branches. Conditional branches test the top of the stack, but flags are not saved as a result of arithmetical or logical operations. Therefore, overflow conditions cannot be tested.

The arithmetical instructions include ADD, SUB, INC, and DEC. Each of these operates on elements at the top of the stack. Logical instructions include WIP, XR, AND, and ORP. Logical shift and left shift are provided where the second byte sets the number of bits to shift. Obviously if operations are to be performed in the stack, instructions must be provided to move data from memory to the stack and back. These instructions are called PUSH and POP. On version 1.0 only three forms of these instructions exist, these are: (1) PUSH (push immediate) pushes the two bytes following the instruction code onto the stack. (2) PUSHA and PFFA push or pop data given by the absolute address specified by the two bytes following the instruction code on to or from the stack respectively; (3) PUSHM and PFFM are the same as (2) except that the relative address of the location of the data is specified (relative to the address of the PUSHM or PFFM instruction). DATA on the stack may be saved to the note buffer by a STNV (stack to duration) STNS (stack to gate), STNP (stack to pitch), and STNL (stack to loudness).

In order to operate effectively with a 16-bit stack oriented machine several special instructions are needed. The instruction, CMR, makes a copy of the word on the top of the stack. CMR does an interchange of the top two elements of the stack. SWI exchanges the high and low bytes of the top element of the stack.
Data input and output are handled by two special instructions: INPUT and OUTPUT, that either place a word on the top of the stack from a device specified by the second byte of the instruction or remove a word from the stack and send it to the device specified by the second byte of the instruction. Since input-output instructions on the 8600 computer transfer 8-bit data, only the low byte is put on the stack or sent to the device.

Several important instructions have been designed for special musical purposes. The simplest of these is the NOTE or note immediate instruction. Here, the duration, gate, and pitch are set into the note buffer from data set in the instruction field. The loudness is not set due to the 5-byte limit of the instruction field, and the note is not played until a PLAY is executed. In this instruction, the duration can be any number of counts from 1 to 255, and the pitch is set directly by the 16-bit value set in the instruction field. The special instruction STRAN can be used to transpose a voice by any interval. It provides an additional pitch bias for each voice which is set from the top element of the stack. Thus the entire line of music can be being read by each voice but played in a different key. CRTNA is used to reset the pitch bias to zero. Execution of the RANDOM instruction puts a 16-bit random number on the top of the stack. These can be used to create stochastic processes. For example, the simple random sample-and-hold sequence of pitches can be formed by passing the random number to the pitch buffer. Changes in tempo can be set up by the TEMPO instruction which moves the value on the top of the stack to the software tempo counter. In this way, the tempo can be changed at any time or even continuously during the composition.

The AMI-1 Assembler-Compiler

The assembler-compiler operates in two modes depending upon the type of mode desired. Mode A instructions are generally set up by the compiler while Mode 3 instructions are set up by the assembler. Because the assembler-compiler has been implemented using the CM-5 MAC assembler, all of the standard pseudo-operators are available for use as well as the full capability of given by MAC instructions. The macros enable the user to expand the language to handle special operations or complex instructions. The use of the MAC assembler does force certain syntax rules to be used that otherwise might be specified more simply if a special purpose assembler-compiler were written.

Mode A System (Direct Mode Notation)

Mode A, as explained before, is a direct 12 tone system designed to communicate standard music notation to the computer. The present version does not implement the rules of key signature, i.e., sharps and flats must be specified for each note.

34
Notes are set up using the $\#$ instruction which requires five arguments to specify the values of parameters required for each note. In general, the fields of the arguments can contain any of the special pre-assigned designators, numbers within the proper range, and expressions that evaluate to the proper range. Each argument has special designators excepting the octave argument. The five arguments are: duration count, gate count, pitch, octave, loudness. Although a maximum duration count of 144 is permitted, the compiler specifies a whole note as 128 clock counts. The duration field permits the following special characters: W (whole), H (half), Q (quartet), S (sevenths), T (thirty second), A (sixty fourth). Any of these can be dotted if followed by a period(s). Multiple dots are implemented down to a 1/128 note duration. Thus C., is legal, while C.~ is not. Dotted extensions to W are not permitted, since they could exceed the maximum count of 144.

The gate count controls the duration of the gate signal. Special designators for this field are: R (rest), S (staccato), L (legato), T (tie), and P (portamento-something between legato and staccato). Presently the special characters set up gate count of 254 of the duration for staccato, 50% for portamento, and 75% for legato. These equations have been found to be unsatisfactory in many cases depending upon the settings of the envelope generators, and modifications of these are planned for the near future. The pitch field permits special characters which are the letter names of the notes: C, D, E, F, G, A, B. These may also be set by numbers 0 through 11 or any expression having a value within that range. Sharps and flats are handled by preceding the letter with + or -, respectively. Both and b are permitted and automatically modify the octave to insure the proper pitch. The octave is specified simply by numbers from 0 to 9, where 0 is the middle octave. A special pitch definition is also available in the case where the pitch is specified by a number or an expression and the octave is left unspecified. In this case, the pitches are based on the number of half tones measured from middle C. Positive and negative values are valid. The proper octave is set automatically if it is not overridden by a specification in the octave field. This procedure allows the semitone values to be computed for compile time. All special designators are determined by testing for the specific characters, and values are not assigned to the character names, thus letters like A,B,C, ..., R, V, ..., K can also be used as variables names within the music program, and values assigned to them will not affect the operation of the compiler. This is not true of the fifth field which controls the loudness. Here specific values are attached to the variables names PPP, PPP, PP, P, MP, MF, MF, F, F, and FPP. These are preassigned variables, and any attempt to change their values will lead to a program error at compile time. An example of a valid set of notes using Mode A might be:

\[ \text{Example:} \]
V1M21: @ Q, L, A, 5, PF ; play A
@ ; the note is played again
@,C, ; play C below A
@,T,E, ; play E with Tie
:next measure to next measure
@ N,; ; whole note
etc.

The V1M21 is an optional label that may be referenced by subroutine calls or branches in the program. In this case it designates the beginning Voice 1 measure 21. The single note specification, above, is useful when changes in several fields are desired for each note, but this notation becomes cumbersome when only pitches are to be changed. A special MACB has been assigned for this case. It is called Pitches.

As an example of its use consider the following:

XX: <Pitches <5, A, 6, +C, E>

Here A in the middle octave, C sharp and E the sixth octave are set up using previously defined other values. If the other fields have not been set up, they may be set using the following special MACB's:

@D Q ; set up a dotted quarter.
@G L ; play it legato.
@L FFFF ; play it very loudly.

Pitches can be preset in the same way if desired, for example:

@F +C ; set up a C sharp
@G 4 ; in the fourth octave.

Note that these presets do not cause a note to be compiled until the single @ is encountered. Comment fields can always be used following the semicolon as shown above to form a well documented score.

Note B

In note B, the AMI assembler-compiler acts as an assembler for the instruction codes. It is important to reemphasize that notes set up by notes A are set directly into memory and will always be played identically as the AMI emulator reads them. Notes set up by the assembler may be computed by the emulator, and the variables used to form them could change. It is important to understand the differences between computed notes and notes set directly in memory. A simple example should make this clear. The assembler contains a repeat operator called REPT which must be terminated by an enQ-macro instruction ENDM. The following instructions set up 20 notes directly in memory.

REPT 20
E 8
S, L, C, 5, F
ENDM
The assembler expands this into 100 bytes of memory with 10 repetitions of the same code. The assembler is also not very smart, so it will decode the field designators each pass through the loop. The following code generates exactly the same sequence of code but compiles much faster:

```
0 S.L.C,5,P
REPT 19
0
ENDR
```

The above examples used only mode A notes. The same repeated sequence can be set up with only 15 bytes of memory using the repeat instruction of the emulator. This code is as follows:

```
REPEAT 20
0 S.L.C,5,P
ENDR
```

This form has two advantages, first it takes only 15 bytes of memory, and it compiles rapidly. This example also shows how mode A and B instructions can be intermixed. As an example of a fully computed note sequence, consider the generation of a chromatic scale over two octaves beginning on middle C. The following program demonstrates the use of many of the instructions available in Mode B.

```
; SET UP THE DURATION VALUES
W EQU 128 ; SET THE VALUE OF A WAVE NOTE
H EQU W/2 ; SET HALF
C EQU H/2 ; SET QUARTER
; SET UP THE STANDARD PITCH VALUES FOR
; OCTAVE PER VEL SYSTEMS
SCV EQU 32768/5 ; SET OCTAVE
; INIT EQU SCV/14 ; SET SEMITONE
; SET A LOUDNESS
PNTS EQU 240 ; 255 IS MAX
; GET VALUE OF MIDDLE C
C5 EQU 5*SCV ; STARTING PITCH
V1 EQU 10498 ; VRC1 CODE IS STARTED AT
1060 Hz IN MEMORY
; RCVI FROM PNTS ; PRESET LOUDNESS
; RCVI CS-SEMIT
; RCVI PITCH ; PRESET PITCH VARIABLE
; BEGIN MAIN HERE
REPEAT 24 ; REPEAT 24 TIMES
PUSHI 0 ; SET DURATION
PUSHI 1 ; SET LEGATO
PUSHI 0-1 ; SET LEGATO
PUSHI PITCH ; GET PITCH VARIABLE
PUSHI SEMIT ; GET VALUE OF SEMITONE
ADD ; INCREASE PITCH BY SEMITONE
; MAKE A COPY
PUSHI PITCH ; RESET PITCH VARIABLE
STOR 340 ; SET PITCH IN BUFFER
```
Of particular interest in the above example is the use of the \texttt{CQNT} instruction to duplicate the pitch value on the stack. This duplication is required since both the \texttt{PWM} and the \texttt{SWP} instructions remove one word from the top of the stack. The \texttt{ENVR} instruction terminates the composition, and the \texttt{END} instructs the assembler that all instructions have been processed.

An example of the use of \texttt{RANDOM} to specify a sequence of pitches is instructive. In this case, the pitches are to be notes from middle C to E flat of the next octave up and the sequence is to run indefinitely.

\begin{verbatim}
; SET UP FIXED NOTE PARAMETERS
O EQU #2 ; SET CQNT FOR A QUARTER NOTE
C7 EQU 32768/5 ; SET OCTAVE
SEM EQU 256/12 ; SEMITONE VALUE
C5 EQU 5*SEM ; MIDDLE C
PWM EQU #205 ; SET PWM
GRO #100H
V1: PUSHI PWM ; SET LIVENESS
STVL RANDOM
L00P: PUSHI 15 ; GET A RANDOM NUMBER
AND ; MASK OFF HIGH BITS
;MULTIPLICATION BY 546 = SEMI = 122H
PWM RN ; SAVE RANDOM NUMBER
PUSHI 0 ; CLEAR TOP SP STACK
PUSHM RN ; PICK UP RANDOM NUMBER
SHFL 1 ; MUL BY 2H
ADD ;
ADD RN ; MUL BY 20H
SHFL 5
ADD ;
PUSHM RN ; MUL BY 200H
SHFL 9
ADD C5
ADD ; ADD MIDDLE C
STPP ; SET PITCH
PUSHI Q ; SET DURATION
STPD PUSHI O-1 ; SET GATE FOR LEGATO
STOG PLAYB ; PLAY NOTE IN BUFFER
BR L00P ; REPEAT FOREVER
ENVR
DN 2 ; random number storage
END
\end{verbatim}
The generation of random semitone sequences is difficult on most analog synthesizers that employ saw and hold gates, but the programmed example above demonstrates how easily this may be accomplished using the emulator to create random numbers. Beginning at LAPP, a 16-bit random number is drawn. Since only 8 values are desired between 0 and 15 inclusive, all but the last 4 bits are masked off by the AND instruction. Ritches are saw generated by the formula

\[ \text{RITCH} = \text{RN} \times \text{SIMT} + \text{C5} \]

where RN is the random number. Since \( 2^{16} = 65536 \) does not have a multiply instruction, the multiplication by a semitone is accomplished by noting that the semitone value is 32768/64 = 512. The number 546 may be represented by 222 hexadecimal, thus the multiplication can be accomplished by three shifts and three adds. The additions remain on the top of the stack where they is added to C5 to give the final value. The final value is transferred to the pitch buffer by the STKP instruction. The duration of each note is set at a quarter note with full legato. The final RN (branch) instruction sends the program to the LAPP location where the process is repeated indefinitely.

An example of a simple two voice sound can be used to show how two voices can be read the same code as well as how two voices are set up. In this case the notes to be played are written as a subroutine called EVENT.

```
V1: 89C 1000H CALL EVMT1 ; PLAY EVENT1
     8 W.R.E.S,F,P ; PLAY A MEASURE BF REST
     END
V2: 89C 1400H CALL EVMT1 ; PLAY EVENT2
     8 W.R.E.S,F,P ; PLAY MEASURE BF REST
     CALL EVENT1
     CALL EVMT1 ; PLAY IT AGAIN
     END
     EVENT1 8D @ ; EIGHTH HEPTE
     8G L ; LEGATO
     86 P ; PERSE
     8PITCHES <S,E,C,G,C,A,C,G,A>
     ; SECOND MEASURE
     8PITCHES <G,A,E,B,D,C,S,B>
     8 RETURN
     END
```

In this example, all notes are specified in node A notation while the actual voices use node B instructions to call the event. Both voices play the same subroutine but separated by a full measure, but note that both voices play for the same total duration as fixed by rest.
Although several of the important features of this language have been shown in the above examples, several more will be demonstrated by the composition PHASE ONE. In this short piece, the rhythmic material is generated by phase drifts between the notes. Precise control of the gates is used to generate the rapidly fading cascades. Tempo changes are written throughout the piece. The third voice is used only to control the loudness while voices one and two play the notes.

Conclusions

The AML-1 emulator and assembler-compiler offers a powerful system for controlling analog synthesizers. Notes may be fully specified at compile-time or calculated during run time. The algorithmic nature of the language can often provide large reductions in the number of notes that need to be specified and likewise, the amount of memory space required to store the composition. Since the assembler-compiler is realized by the standard Digital Research Macro assembler, all of the features of the Macro assembler are available to the user. Also extensions to the compiler are easily accomplished through the use of Macro definitions. Presently the only disadvantages to the system are that the compile-time for large compositions can be fairly long, and the use of the stack-oriented computer with the PDP-8 instruction set requires some understanding of programming techniques.

Acknowledgements

1) I would like to express my appreciation to Dr. Ralph Wickard whose studio has been used for much of the past year in testing the usefulness of AML-system and whose help was invaluable in preparing the taped examples for this talk. He is also responsible for the fine arrangement of the J.S. Bach Sonata #6 in E major for solo violin.

2) I wish to express special thanks to Dr. Philip Springer for the years of discussions, consultations, and inspirations related to electronic music and the interaction of art, science, and technology. His studio was also used to test concepts in the early phases of the AML development.

3) I express my appreciation to composers Furry Riley, Steve Welch, Phil Glass, and Georgi Ligeti whose music provided much incentive to continue the long hours of hardware and software development required to realize this initial study and development. A good process is worth a thousand notes.
Summary of Instruction for the ANL-1 Music Emulator

I. Program Control Instructions

ENOV  End of Voice or Composition
REPEAT n  repeat following material n times
END*  end of repeated material
CALL  call a subroutine or named sequence of notes
RETURN  return from a subroutine
BR  unconditional branch
BRZ  branch on zero
BRE  branch on not zero
BPF  branch on positive
BN  branch on negative

II. Arithmetical Instructions

ADD  add top two elements of the stack
SUB  subtract the top two elements of the stack
INC  increment the top of stack
DEC  decrement the top of stack

III. Logical Instructions

NXT  logical NXT of top of stack
SR  logical SR of top two elements of the stack
AND  logical AND of top two elements of the stack
EOR  exclusive OR of top two elements of the stack
SHIFTL n  shift n bits left
SHIFTR n  shift n bits right

IV. Data Transfer Instructions

PUSH  push immediate data on to stack
PUSHA  push data specified by the absolute address on to the stack
PUSHM  push data specified by the relative address on to the stack.
V. Special Stack Operations

POP Top the top element off the stack to the absolute address location.
POPm Pop the top element off the stack to the relative address location.

VI. Input-Output Instructions

INPUT Input data from specified I/O port to top of stack
OUTPUT Output data from the top of the stack to the specified I/O port.

VII. Special Music Instructions

CTRAN Clear transpose setting.
STRN Set transpose from data on top of stack.
RANDOM Place a 16-bit random number on the top of the stack.
TEMPO Set the tempo counter to the specified rate.
NOFR Set the note buffers according to the data in the instruction.

VIII. Stack to Command buffer Transfer Instructions

STBR The low byte on the top of the stack is transferred to the duration counter.
STBG The low byte on the top of the stack is transferred to the gate counter.
STBB The 16-bit data on the top of the stack is transferred to the pitch buffer.
STBL The low byte on the top of the stack is transferred to the loudness buffer.

IX. Debugging Instructions

STBC The data on the top of the stack is transferred to the user console. The stack is not modified.