SESSION VIII SOFTWARE DESIGN STRATEGIES IN OPERANT CONDITIONING SYSTEMS

Howard Kaplan, *Presider*

Parallel processing strategies in the application of microcomputers to the behavioral laboratory

DAVID G. SPENCER, JR.
Troponwerke, Cologne, West Germany

and

MICHAEL W. EMMETT-OGLESBY
Texas College of Osteopathic Medicine, Fort Worth, Texas

The Operant Package for the Neurosciences (OPN) is a software system that runs on the TRS-80 Model I and Model III and the Apple II+ and Apple IIe microcomputers. It is designed to allow a single microcomputer to control behavioral experiments in up to eight different stations, record experimental data, and provide detailed data analysis. Although segments of the program are written in BASIC, allowing users to interact with OPN in a simple English question-and-answer format, the part that provides real-time experimental control is written in Z-80 assembly language. The present report describes in detail assembly language algorithms that we found to facilitate speed of processing and flexibility of control. Particular problems dealt with how to update and change flag variables controlling the reinforcement contingencies and output conditions in each station, as well as how to decide when each station should shift contingencies. The solutions to these problems utilize byte-oriented parallel processing strategies, converting serially organized reference information for each station into a group of working flag bytes that control contingency and output variables for all stations in parallel.

The potential flexibility and data-analysis capabilities of computers have made them extremely popular alternatives to digital logic modules in the operant behavioral laboratory. Although software designed to conduct behavioral experiments has long been available for minicomputers such as the PDP-11 (Snapper & Kadden, 1973), software for microcomputers has only recently been developed. Minicomputers can use software written in high-level languages like FORTRAN to run complex behavioral experiments in several stations with an acceptable temporal resolution. However, microcomputers have important constraints on memory storage capacity and processing speed. Thus, microcomputer software systems

for the behavioral laboratory that primarily use high-level languages (BASIC, FORTRAN, or Pascal; see Thompson, 1979, Carroll, Santi, & Rudell, 1981, Gordon, Foree, & Eckerman, 1983, and Hale, Myerson, & Miezin, 1982) have severe limitations, even though some assembly language subroutines are also used. First, only one station is usually controlled. Second, there is a trade-off between real-time temporal resolution (frequency of sampling inputs and sending outputs) and complexity of the behavioral schedule. Third, the preceding fact has the consequence that software cannot be made complex enough to shift reinforcement contingencies depending on the behavior of the subject. Although some advantage may be gained by executing a compiled version of a BASIC program (Rayfield, 1982), the gain in execution speed is not enough to overcome the problems listed above. In fact, it is only through the exclusive use of assembly language programs during real-time session execution that instruction processing speed is high enough to allow substantial program complexity (Emmett-Oglesby, Spencer, & Arnoult, 1982).

Our goals in developing a microcomputer software sys-

The assistance of Douglas E. Arnoult in developing OPN into a functioning software system is gratefully acknowledged, as is the patience and continual support of Dr. Harbans Lal. Supported in part by the William and Catherine Bryce Memorial Fund. Reprints may be obtained from either author. David G. Spencer's mailing address is: Neurobiology Department, Troponwerke, Neurather Ring 1, 5000 Cologne 80, West Germany. Michael W. Emmett-Oglesby's mailing address is: Department of Pharmacology, Texas College of Osteopathic Medicine, Fort Worth, TX 76107.

tem for behavioral laboratories were flexibility and ease of use. Flexibility was to be attained through the use of parallel processing algorithms in assembly language. Ease of use was to be produced through an interactive question-answer format with the user, which would take in control parameters without the user being aware of a computer language as such. These programs are written in BASIC, but this is transparent to the user because the programs interact with the user in simple English sentences, obtaining the schedule parameters and reporting the experimental data, respectively. Ease of use was considered important for reasons of efficiency. Behavioral scientists should not be forced to spend considerable amounts of time learning to program computers, regardless of how "easy" the programming language is. In addition, turnover of students and other personnel in most laboratories today is high enough that it is important to have a system that can be quickly mastered, encouraging users to design their own experiments. Simplicity of use is not achieved through simple programming strategies, however. The purpose of the present paper is to describe some assembly language algorithms used in developing such a microcomputer laboratory software system that are essential in producing the necessary speed of execution and flexibility.

DESCRIPTION OF OPN

Hardware

OPN is a Z-80-based program. It executes on Radio Shack TRS-80 Model III and IV microcomputers, as well as Z-80-adapted Apple II+ and IIe systems. All versions require two disk drives and full RAM. An LVB interface (Med Associates, Inc.; Lafayette Instruments, Inc.) is necessary for interfacing operant chambers with the microcomputer.

Input Configuration

OPN allows up to three independent manipulanda or other input devices for each of up to eight operant stations. The only restriction is that they must supply binary, rather than analog, data.

Output Configuration

On the TRS-80, OPN can control up to 16 independent outputs for each of up to eight operant stations. Eight outputs are designed for continuous operation (e.g., a red panel light illuminated for 5 sec) during a certain segment of the session. Six outputs are designed for either cycled (e.g., a flashing houselight) or continuous operation during a certain segment of the session. Two outputs are designed as special-purpose reinforcement outputs. The outputs configuration is identical for the Apple, with the exception that only three outputs are available for continuous operation.

Programmable Schedule Contingencies

OPN is a modular program with two distinctly different portions. One part of the program gathers user input

with regard to stimulus-response-reinforcement contingencies and data-collection parameters. A second part of the program is an assembly language routine that executes the functions set up by the user. The user input portion of the program is handled through a greatly simplified version of state notation to describe operant contingencies. State notation has been widely advocated as a means for precisely specifying operant contingencies (Mechner, 1959; Snapper, Knapp, & Kusher, 1970); however, as employed in a programming language such as SUPERSKED (Snapper, Kadden, & Inglis, 1982), state notation does not resemble the ordinary language of operant description (e.g., FR10, VI30-sec, etc.). OPN's version of state notation uses a menu-driven system to help the user specify fundamental operant contingencies, called components. Each component is roughly equivalent in capability to the previous OPS program, and consists of specification of the controlling schedule (fixed or variable ratio, fixed or variable interval, fixed or variable time, extinction, DRL, DRH, or combinations of these to form tandem and concurrent schedules), which of the three inputs are to be considered correct toward fulfilling schedule requirements, and what the output configuration is to be. For any experimental procedure, up to 15 different components can be defined. By entering the sequence in which these components are to be executed, a wide range of procedures can be created.

Any combination of three specific conditions can be requested for switching from one component to the next: (1) meeting the component's reinforcement contingency, (2) time-out of the component "clock," and (3) the occurrence of an incorrect response. Furthermore, the user can specify for each of these exit conditions whether the next component is to be selected from the previously defined component sequence, or whether a specific component is to be selected. This feature gives OPN the capacity to modify the programmed sequence of events in the operant chamber, depending on the behavior of the organism within that chamber. An example of this is provided in the spatial alternation procedure, in which the subject must alternate responses over trials between two levers. If an incorrect response occurs, then a correction trial is delivered next and the subject must switch levers to go to the one that was correct on the previous trial.

Data Analysis

OPN is designed to give two main types of data analysis. First, every time a component transfer occurs, OPN stores data on responses on each of the three manipulanda, number of reinforcements, and amount of time spent in the component. At the end of the session, the cumulated data on these measures are automatically printed out for each of the different component types in use. Since data for each component transfer are stored in a separate location for each chamber, procedures that use a large number of component transfers allow for a rather fine-grained analysis of the session. Analysis programs exist within OPN to allow the rapid sorting through of the component data to provide information ranging from the total data

output, to data from specific occurrences of certain components, to data from components only when they occurred in certain sequences. Second, OPN allows the user to perform temporal response analyses in up to two different component types. There are three types of temporal analysis: interresponse time (IRT), postreinforcement pause (PRP), and postreinforcement response frequency (PRRF). Each analysis provides 56 temporal "bins" of response information, including an underflow and an overflow bin. The IRT analysis plots frequency of responses by time since the previous response. The PRP analysis plots frequency of the responses first made following a reinforcement by time since the last reinforcement. The PRRF analysis plots frequency of all responses by time since the last reinforcement.

ASSEMBLY LANGUAGE EXECUTION PROGRAMS IN OPN

The decision to write the execution portion of OPN entirely in assembly language is based on two factors: first, higher level languages execute more slowly than assembly language routines, and second, higher level languages require more space for both their programs and their data than do assembly programs. Many behavioral software systems use higher level languages, such as BASIC, mixed with assembly routines to control real-time experiments (e.g., Carroll et al., 1981). In most of these programs, the assembly routine handles input/output (I/O) tasks, while logical decisions are made in the high-level language. However, in even a simple operation such as decrementing a counter from a specified value to 0, BASIC requires approximately 400 times as long to perform the operation as does assembly code (Barden, 1979). In tests with the MBASIC compiler BASCOM (Microsoft, Inc.), we determined that compiling the BASIC program increased execution speed approximately threefold, which is still very slow relative to assembly execution. Moreover, compiled programs occupy severalfold more RAM than do assembly programs, and when storage is a limiting constraint, their use results in reduced capacity for data acquisition.

Program speed is a crucial variable for real-time programs. If few stations are being controlled, and if input and output events happen infrequently, then any language will be suitable for programming. However, as control complexity increases, the time of processing is lengthened. Higher level languages use a compiler to select algorithms to be executed in assembly language, and, particularly for complicated procedures, the exact set of assembly code generated by the compiler cannot be specified in advance. Thus, although the time to execute each assembly instruction is known, there may be thousands of assembly instructions in a compiled program, and their order of execution under all conditions is difficult to ascertain. As a result, no high-level language can precisely specify how long it takes to execute code under all conditions. On the other hand, when assembly instructions are coded by the

user, the time to execute all subroutines under worst-case conditions can be specified precisely. In OPN, using eight stations with 3 possible inputs per station and 16 outputs per station, a worst-case condition can be envisioned in which all stations generate input simultaneously, all stations employ maximum data-collection options, and all stations change control contingencies after this input. OPN accomplishes these tasks in under 25 msec; therefore, 25 msec was used as the timing interval governing each initiation of program execution. This rate of execution is compatible with operant responses from essentially all laboratory species, including pigeons, which can respond on illuminated keys with interresponse times of less than 0.1 sec (Ferster & Skinner, 1957).

Unfortunately, the time and space advantages of assembly language are offset by many disadvantages. Good assembly language routines require understanding of the hardware and software characteristics of a specific CPU as well as the ability to work in hexadecimal arithmetic and a working knowledge of either Boolean algebra or symbolic logic (Zaks, 1980); therefore, assembly language programs are difficult to write and debug, particularly if the overall program is long. The real-time execution program, OPN, in its uncommented form is a 49K assembly language file that compiles to a 4K program in machine code. Approximately 2,000 to 3,000 h were required to develop the program to this state. In addition, assembly language routines are difficult to modify. High-level languages can rely on compilers to reallocate memory and modify subroutines as the number of stations being run increases or decreases. Assembly language programming requires that the user oversee all memory allocations and subroutine alterations. Thus, major modifications to a program of the size of OPN would require substantial hours of programming.

Fundamental Algorithms in OPN

The preceding discussion briefly described the capabilities of OPN based on state-type user input and emphasized why many real-time application programs are written partly in assembly language. OPN differs from these programs in that the entire execution program is written in assembly language. Of the various programming problems that this approach entails, the ones that prove most challenging are the development of algorithms to generate flags governing contingencies between a subject's responses and consequent events. These algorithms can logically be separated into those dealing with two separate problems: how to make the decision for each chamber of whether the controlling contingency should be changed, and how to update the appropriate schedule flags.

Contingency-Shift Decisions

Contingency shifts present three problems: First, some method is necessary for determining the logical conditions that, on each pass through the program, may result in a shift of the prevailing contingencies for each station. Sec-

ond, upon the occurrence of one of these candidate conditions, there must be procedures for deciding whether a shift of components is necessary. Third, an algorithm for the selection of the new schedule and output conditions must be generated.

In OPN, three conditions are defined that may logically result in a termination of the present component: (1) fulfillment of the schedule requirements a specified number of times, (2) the occurrence of an incorrect response, and (3) component duration. The process of evaluating the status of these conditions is schematized in Figure 1 and is described verbally below. Prior to the operation of the algorithm in Figure 1, three flags are generated, with set bits for stations meeting one (or more) of the three conditions listed above. The decision routine discussed below uses these three flags to assess all stations on each of the three exit conditions, in order.

If one or more stations have completed the present reinforcement schedule, two further decisions must be made to determine whether a new component should be begun. First, bits representing the relevant stations must be examined in the flag byte that holds the information of whether or not to exit at all on a schedule fulfillment. Second, if the appropriate bits are set in this flag, then the

2-byte counters holding the number of times the schedules are to be repeated for these stations are inspected. If the counters have been decremented to zero, then the present components can be terminated. If nonzero values exist in these counters, however, these values are decremented and the current components are continued. Similarly, if incorrect responses occur (responses on a manipulandum other than that defined as appropriate for reinforcement), the appropriate bit positions are examined for set bits in the flag byte representing stations to be terminated upon occurrence of an incorrect response. Finally, if set bits exist in a flag byte that represents stations governed by time, a 2-byte timer is decremented for these stations; if a zero value occurs, the current components are terminated.

When a termination of the current component has been dictated by the occurrence of one of the three conditions described above, the remaining task is to decide which of the up to 15 components (set of schedule contingencies and output patterns) for the chamber in question is to be activated next. As described previously, users of OPN are allowed to define a sequence of components, which are to be executed in order. Thus, one possibility is simply to select the next component by inspecting the next position in the component sequence. However, users are also allowed to define whether a specific component (e.g., number 3) is to be exited following any of the three exit conditions. Upon component termination, then, the appropriate bit position in the flag byte representing stations to exit to a specific component (given this specific exit condition) is inspected (see Figure 1). If a set bit is encountered, then a 1-byte variable is inspected to determine the component number to activate next. Since the variable that points to the current location in the component sequence is not affected by this action, the programmed sequence of events can be rejoined at any time.

Updating Flags

As described in Preliminary Considerations, computers are most powerful when executing loop-oriented subroutines. The execution portion of OPN takes advantage of this property by generating 8-bit bytes in which each bit position represents an experimental station. Thus, during program execution, a byte is generated in which set bits represent stations that should decrement reinforcement-governing counters as a consequence of response input. A subroutine that decrements counters corresponding to set bits is then called. We refer to this strategy as parallel processing of information for two reasons: First, logic decisions can be made for all stations in parallel. For example, if a flag exists in which set bits govern stations on ratio-based schedules, then it can be ANDed against the input bytes to determine simultaneously which station, of all stations that were governed by ratio contingencies, emitted a response. Second, all events of a type are processed for all stations before the next type of event is considered. This strategy can be contrasted

Figure 1. Schematic of the algorithm used to evaluate component exit conditions and to select new components. Using the flag-rotation strategy described in the text, stations are sequentially evaluated at each decision step before proceeding to the next. Thus, although evaluation proceeds station-wise at each decision, the entire algorithm proceeds decision-wise, essentially processing stations in parallel. The three main decisions shown on the top row of the figure are shown in order of priority in case more than one of the three conditions occurs simultaneously.

with interrupt-driven or round-robin type of approaches in which all functions for each station are processed station by station. The advantage of the parallel strategy is that counters of the same type (e.g., all stations' response counters) can be stored in sequence in memory, and counters for the different stations can be located very rapidly using a pointer and an offset routine.

OPN allows up to 15 different components for each station, and the components for each station need not be the same. To reduce this enormous amount of information into a usable form during session execution, the results from user input routines are compressed into binary variables (Zaks, 1980). Thus, the execution routine in OPN stores each component as 32 bytes, in which successive bits in a byte control different variables. At this point, the information for control flags can be conceptualized as being stored sequentially in the bytes associated with each station and each component. To minimize the time spent searching for the control flags for a particular station, a component buffer area is established in memory where the 32 bytes of the component currently scheduled to execute are transferred on encountering the logical conditions described under Contingency-Shift Decisions. Having transferred the next executing component to this area, the remaining problem is to transform the sequentially stored information for each station into the parallel flags used during OPN execution. This process turns out to be surprisingly time consuming, and a novel method to solve this problem in real time is presented below.

The algorithm can be understood most easily by beginning with two 8-bit flags. Set bits in one flag might represent stations under response ratio-based schedules of reinforcement, and set bits in the other flag might represent stations under a time-based schedule of reinforcement. What happens during the course of an experimental session when one (or more) of the stations change from one type of reinforcement control to the other? Suppose, for example, that the station represented by bit position 3 was initially governed by a ratio-based schedule (e.g., reinforcement might be delivered every 10th leverpress). The control flag for this type of schedule would have a 1 set in bit position 3, and the byte resulting from response input could be logically ANDed against this flag to determine whether a counter governing responses remaining before reinforcement should be decremented. If this decrement results in a counter value of zero, the station in question might then be given reinforcement and be due for a shift to a time-based schedule of reinforcement. Prior to the change of contingencies, bit position 3 in the ratio flag was set to 1 and the third bit position in the time-based flag was 0. Following the change of contingencies, the reverse will be true. There are numerous approaches for accomplishing these discrete changes of bit patterns, but few readily lend themselves to rapid real-time processing.

One reasonable approach might be termed the "event-oriented" approach. Here, each component for each experimental station is represented by individual bits in con-

trol bytes. Each byte refers to the status of one particular contingency (e.g., a ratio schedule), and each bit within a byte represents the status of a particular station with respect to that contingency. A value of 0 indicates that the corresponding contingency does not occur, and a value of 1 indicates that it is in effect. At the time of a contingency shift (the current schedule requirements having been satisfied) for the third station, for instance, the processor could check the bytes governing ratio-based and time-based contingencies for the next component of this station (represented by the 3-bit position in the control flags) and, according to their content, set or reset the 3-bit positions of these control flags. That is, the computer checks and modifies the control bits for a particular station when that station must change components. This is the most intuitively logical solution, but it turns out to be excessively wasteful of both computer memory and execution time: Roughly 45 μ sec are required on the TRS-80 Model III to update each bit position. Unfortunately, a program with the flexibility of OPN requires 46 controlling flags. Thus, each time a station changes contingencies (components), 2.07 msec would be required just to establish the new control flags. If flags for all eight stations must be updated simultaneously, 16.56 msec would be required. In such a case, completion of other tasks (e.g., sorting incoming response data, making decision about outputs, updating

Bit configuration Pre-switch Flag Function		Bit configuration Post-switch	
0 0 0 0 1 1 1 1	RATIO	0 0 0 0 0 1 1 1	
0 0 1 1 0 0 0 0	INTERVAL	0 0 1 1 1 0 0 0	
0 0 0 0 0 0 0 0	TIME	0 0 0 0 0 0 0 0	
0 0 1 1 0 0 0 0	VARIABLE	0 0 1 1 0 0 0 0	
0 0 0 0 0 0 0 0	DRH	0 0 0 0 0 0 0 0	
0 0 0 0 0 0 0 0	DRL	0 0 0 0 0 0 0 0	
0 0 1 1 1 1 1 1	GIVESR	0 0 1 1 1 1 1 1	
0 0 1 1 1 1 1 1	LIGHTS	0 0 1 1 1 1 1 1	

Figure 2. Examples of bit configurations in flags used to make logical decisions in OPN. Bits are referenced right to left, from 0 to 7, respectively. In this example, the experimental stations governed by bit positions 6 and 7 are not being run (all schedule-governing bits are 0). Prior to the shift of contingencies, stations governed by bits 0-3 are ratio-based (RATIO); stations governed by bits 4 and 5 are interval-based (INTERVAL), and, in addition, the interval is variable (VARIABLE); stations 0-5 are candidates to receive reinforcement if contingencies of reinforcement have been fulfilled (GIVESR); the houselights (LIGHTS) are on in stations 0-5.

clocks and screen displays) are unlikely to be accomplished in 25 msec, the maximum period of time allowed by OPN between monitoring of inputs. In addition, the number of bytes required to store flag information for each station becomes excessive.

OPN utilizes an alternative technique to change controlling flag bits, which might be termed a "worst-case" approach. The program executes within specified time limits under a worst-case scenario, and thus all situations less demanding than the worst case are readily accommodated at least as rapidly as the worst case is. The fundamental problem can be conceptualized as the transformation of information stored sequentially by stations into information stored in parallel for all stations. That is, because the stations are controlled independently of each other, each station must have its information stored separately from that of others. However, when the session is being executed, this information must be integrated with congruent information for the seven other experimental stations. As described previously, program execution is maximally efficient when eight stations are represented per byte, and each byte represents a different control flag.

Surprisingly, a worst-case orientation to the problem provides a solution that executes quite rapidly under all circumstances. In the worst case, all eight stations must have their flags updated before another response can occur. A critical point for this approach is that no logical decisions pertaining to changing flags are necessary because all flags are routinely updated every pass. Thus, in assembly language, whether one or all stations change controlling variables, it turns out to be faster to update all flags rather than to make the logical decisions necessary for determining whether flags for each individual station should be set or reset. On the basis of this principle, we developed an algorithm that updates all flags for all stations whenever it is executed. Stations whose controlling information should remain intact simply have their flag bits refreshed; stations whose controlling flags must change are modified at the same rate. The time required for this routine to update 46 flag variables for all eight stations is 3.2 msec using a Z-80 CPU with a 2-MHz clock.

Figures 2-4 illustrate this subroutine. Eight control flags used for making logical decisions during program execution are shown in Figure 2. Each experimental station occupies a specified bit position in each flag, and in accordance with bit-numbering conventions within bytes, bits are assigned to stations from right to left. The left side of Figure 2 describes the control status for all eight stations for these eight flags prior to changing contingencies; the right side depicts the control status after the change. Figure 3 shows the sequential arrangement of information for each station that must be transferred to the proper bit position for each station in the control flags after the contingency change. Note that for any station in Figure 3, the right to left arrangement of bits is identical to the top to bottom arrangement in Figure 2 for that station. As shown in Figure 4, each new control flag is generated by shifting 1 bit from each of eight registers

1 1 0 0 0 0 0 1	Station 0
1 1 0 0 0 0 0 1	Station 1
1 1 0 0 0 0 0 1	Station 2
1 1 0 0 0 0 1 0	Station 3
1 1 0 0 1 0 1 0	Station 4
1 1 0 0 1 0 1 0	Station 5
0 0 0 0 0 0 0 0	Station 6
0 0 0 0 0 0 0 0	Station 7

Figure 3. Serial storage of control flag information for experimental stations 0-7. Bits are numbered right to left, 0-7. For each experimental station, the same bit position contains information for the same control variable: bit 0 = RATIO; bit 1 = INTERVAL; bit 2 = TIME; bit 3 = VARIABLE; bit 4 = DRH; bit 5 = DRL; bit 6 = GIVESR; and bit 7 = LIGHTS. See text and Figure 2 for details.

Prior to Flag Generation		After Flag Generation	
Register A	CARRY	Register A	CARRY
x x x x x x x x	x	0 0 0 0 0 1 1 1	x
Register D		Register D	
1 1 0 0 0 0 0 1		x 1 1 0 0 0 0 0	
Register E		Register E	
1 1 0 0 0 0 0 1		x 1 1 0 0 0 0 0	

Figure 4. Representation of the mechanism by which serial information is transformed to parallel information. 1 = bit position is set; 0 = bit position is reset; x = bit position is either set or reset. This example generates the flag labeled RATIO postswitch shown in Figure 2. The flag will be generated in register A, and the initial contents of the register are immaterial. At the same time the flag is to be constructed, the bytes from Figure 3 describing the eight experimental stations are loaded in the Z-80 general utility registers. In this example, stations 0 and 1 are shown in the Z-80 D and E registers, respectively. To obtain the flag constructed in the A register, the D register is shifted right, producing the bit pattern shown in the D register on the right side of the figure. The shift deposits the least significant bit of the D register in the CARRY, and this 1 is subsequently rotated into the most significant bit position of the A register. Next, this process is repeated for the E register, generating the bit configuration in E shown on the right side of the figure, resulting in 1s in bit positions 6 and 7 of the A register. The process is repeated for the other six experimental stations, resulting in the construction of the RATIO flag in the A register. This flag is stored to memory, and the entire process is repeated to construct flags, INTERVAL, TIME, VARIABLE, DRH, DRL, GIVESR, and LIGHTS, respectively. The entire process requires less than 0.7 msec.

(each register holding one station's bit status on eight different schedule flags) one at a time into the A register. The end result of eight such bit shifts is the production of the top flag labeled "after" in Figure 2.

CONCLUSION

Assembly language algorithms with rather general utility were developed in order to generate a program for a microprocessor that meets the requirements of flexibility and speed necessary for real-time control of operant chambers in a behavioral laboratory. The algorithm for execution of one of several possible reinforcement schedules at each of a large number of "choice points" during the experimental session, depending on the behavior of the subject, provides the flexibility necessary for OPN. The algorithms that access and transfer reference material to working variables in parallel for eight behavioral stations provide the speed and multiple-processing capability necessary to execute OPN in minimum time on relatively inexpensive microcomputers.

REFERENCES

- BARDEN, W., JR. (1979). *TRS-80 assembly-language programming*. Fort Worth, TX: Radio Shack.
- CARROLL, M. E., SANTI, P. A., & RUDELL, R. L. (1981). A microcomputer system for the control of behavioral experiments. *Pharmacology, Biochemistry and Behavior*, 14, 415-417.
- EMMETT-OGLESBY, M. W., SPENCER, D. G., JR., & ARNOULT, D. E. (1982). A TRS-80-based system for the control of behavioral experiments. *Pharmacology, Biochemistry and Behavior*, 17, 583-587.
- FERSTER, C. B., & SKINNER, B. F. (1957). *Schedules of reinforcement*. New York: Appleton-Century-Crofts.
- GORDON, W. A., FREE, D., & ECKERMAN, D. A. (1983). Using an Apple II microcomputer for real-time control in a behavioral laboratory. *Behavior Research Methods & Instrumentation*, 15, 158-166.
- HALE, S. S., MYERSON, J., & MIEZIN, F. M. (1982). Top-down design for a system to control operant choice experiments. *Behavior Research Methods & Instrumentation*, 14, 272-275.
- MECHNER, F. (1959). A notation system for description of behavioral processes. *Journal of the Experimental Analysis of Behavior*, 2, 133-150.
- RAYFIELD, F. (1982). Experimental control and data acquisition with BASIC in the Apple computer. *Behavior Research Methods & Instrumentation*, 14, 409-411.
- SNAPPER, A. G., & KADDEN, R. M. (1973). Time-sharing in a small computer based on a behavioral notation system. In B. Weiss (Ed.), *Digital computers in the behavioral laboratory*. New York: Appleton-Century-Crofts.
- SNAPPER, A. G., KADDEN, R. M., & INGLIS, G. B. (1982). State notation of behavioral procedures. *Behavior Research Methods & Instrumentation*, 14, 329-342.
- SNAPPER, A. G., KNAPP, J. Z., & KUSHNER, H. K. (1970). Mathematical description of schedules of reinforcement. In W. N. Schoenfeld (Ed.), *The theory of reinforcement schedules*. New York: Appleton-Century-Crofts.
- THOMPSON, G. C. (1979). Behavioral programming with the Apple II microcomputer. *Behavior Research Methods & Instrumentation*, 11, 585-588.
- ZAKS, R. (1980). *How to program the Z80*. Berkeley, CA: Sybex.