

Stephen Wozniak  
Apple Computer  
20863 Stevens Creek Blvd, B3C  
Cupertino CA 95014

# SWEET16: The 6502 Dream Machine

While writing Apple BASIC for a 6502 microprocessor I repeatedly encountered a variant of Murphy's Law. Briefly stated, any routine operating on 16 bit data will require at least twice the code that it should. Programs making extensive use of 16 bit pointers (such as compilers, editors and assemblers) are included in this category. In my case, even the addition of a few double byte instructions to the 6502 would have only slightly alleviated the problem. What I really needed was a hybrid of the MOS Technology 6502 and RCA 1800 architectures, a powerful 8 bit data handler complemented by an easy to use processor with an abundance of 16 bit registers and excellent pointer capability. My solution was to implement a nonexistent 16 bit "metaprocessor" in software, interpreter style, which I call SWEET16. This metaprocessor was sketched at the end of my article in May 1977 BYTE, and the purpose of this article is to fill in the details of SWEET16.

SWEET16 is based around sixteen 16 bit

registers called R0 to R15, actually implemented as 32 memory locations. R0 doubles as the SWEET16 accumulator (ACC), R15 as the program counter (PC), and R14 as the status register. R13 holds compare instruction results and R12 is the subroutine return stack pointer if SWEET16 subroutines are used. All other SWEET16 registers are at the user's unrestricted disposal.

SWEET16 instructions fall into register and nonregister categories. The register operations specify one of the 16 registers to be used as either a data element or a pointer to data in memory depending on the specific instruction. For example, the instruction INR R5 uses R5 as data and ST @R7 uses R7 as a pointer to data in memory. Except for the SET instruction, register operations only require one byte. The nonregister operations are primarily 6502 style branches with the second byte specifying a  $\pm 127$  byte displacement relative to the address of the following instruction. If a prior register operation result meets a specified branch condition, the displacement is added to SWEET16's program counter, effecting a branch.

SWEET16 is intended as a 6502 enhancement package, not a stand alone processor. A 6502 program switches to SWEET16 mode with a subroutine call, and subsequent code is interpreted as SWEET16 instructions. The nonregister operation RTN returns the user program to the 6502's direct execution mode after restoring the internal register contents (A, X, Y, P and S). The example of listing 1 illustrates how to use SWEET16 in some program segment.

SWEET16	300 B9 00 02	LDA	IN, Y	Get a char.
	303 C9 CD	CMP	"M"	"M" for move?
	305 D0 09	BNE	NOMOVE	No, skip move.
	307 20 00 08	JSR	SW16	Yes, call SWEET16.
	30A 41	MLOOP	LD	@R1
	30B 52		ST	@R2
	30C F3		DCR	R3
	30D 07 FB		BNZ	MLOOP
	30F 00		RTN	
	310 C9 C5	NOMOVE	CMP	"E"
	312 D0 13		BEQ	EXIT
	314 C8		INY	No, continue.

Note: Registers A, X, Y, P and S are not disturbed by SWEET16.

*Listing 1: Use of SWEET16 within an assembly language program is accomplished by executing a subroutine call to the SWEET16 entry point (address 307 here). This call preserves the processor registers at the time of entry and begins interpretive execution. End of interpretive execution is signaled by a RTN operation code of SWEET16, at which point all the processor registers will be restored.*

## Instruction Descriptions

The SWEET16 op code list is short and uncomplicated. Excepting relative branch displacements, hand assembly is trivial. All register op codes are formed by combining two hexadecimal digits, one for the op code and one to specify a register. For example,

op codes 15 and 45 both specify register R5 while codes 23, 27 and 29 are all ST (store) operations. Most register operations of SWEET16 are assigned to numerically adjacent pairs to facilitate remembering them. Thus LD and ST are op codes 2n and 3n respectively, while LD @ and ST @ are codes 4n and 5n.

Operation codes 00 to 0C (hexadecimal) are assigned to the 13 nonregister operations. Except for RTN (op code 0), BK (0A), and RS (B), the nonregister operations are 6502 style relative branches. The second byte of a branch instruction contains a  $\pm 127$  byte displacement value (in two's complement form) relative to the address of the instruction immediately following the branch. If a specified branch condition is met by the prior register operation result, the displacement is added to the program counter effecting a branch. Except for BR (Branch always) and BS (Branch to Subroutine), the branch operation codes are assigned in complementary pairs, rendering them easily remembered for hand coding. For example, Branch if Plus and Branch if Minus are op codes 04 and 05, while Branch if Zero and Branch if NonZero are op codes 06 and 07.

## Theory of Operation

SWEET16 execution mode begins with a subroutine call to SW16 (see listing 2, an assembly of SWEET16). The user must insure that the 6502 is in hexadecimal mode upon entry. [For those unfamiliar with the 6502, arithmetic is either decimal or hexadecimal (binary) depending on a programmable flag .CH] All 6502 registers are saved at this time, to be restored when a SWEET16 RTN instruction returns control to the 6502. If you can tolerate indefinite 6502 register contents upon exit, approximately 30  $\mu$ s may be saved by entering SWEET16 at location SW16 + 3. Because this might cause an inadvertent switch from hexadecimal to decimal mode, it is advisable to enter at SW16 the first time through.

After saving the 6502 registers, SWEET16 initializes its program counter (R15) with the subroutine return address off the 6502 stack. SWEET16's program counter points to the location preceding the next instruction to be executed. Following the subroutine call are 1 byte, 2 byte, or 3 byte long SWEET16 instructions, stored in ascending

*Listing 2: SWEET16 assembly. The SWEET16 program, assembled to reside at location 800 hexadecimal, is presented by this listing. The primary entry point is at the beginning, location SW16. An alternate entry point if there is no need to save processor registers is at location 803 in this assembly, SW16+3.*

```

SWEET16 INTERPRETER
11:18 A.M., THU, MAY 12, 1977

00001 ****
00002 *
00003 * APPLE-II PSEUDO *
00004 * MACHINE INTERPRETER *
00005 *
00006 * S. WOZNIAK *
00007 * APPLE COMPUTER INC *
00008 *
00009 ****
00010 TITLE 'SWEET16 INTERPRETER'
00011 R0L EPZ $0
00012 R0H EPZ $1
00013 RI4H EPZ $1D
00014 RISL EPZ $1E
00015 RISH EPZ $1F
00016 S16PAG EQU $F7
00017 ORG $800
00018 JSR SAVE PRESERVE 6502 REG CONTENTS
00019 PLA
00020 STA R15L INIT SWEET16 PC
00021 PLA FROM RETURN
00022 STA R15H ADDRESS
00023 JSR SW16C INTERPRET AND EXECUTE
00024 JMP SW16B ONE SWEET16 INSTR.
00025 SW16C INC R15L
00026 BNE S16D INCR SWEET16 PC FOR FETCH
00027 INC RISH
00028 LDA #S16PAG
00029 PHA PUSH ON STACK FOR RTS
00030 LDY #$0
00031 LDA (R15L),Y FETCH INSTR
00032 AND #$F MASK REG SPECIFICATION
00033 ASL A DOUBLE FOR 2-BYTE REG'S
00034 TAX TO X-REG FOR INDEXING
00035 LSR A
00036 EOR (R15L),Y NOW HAVE OPCODE
00037 BEQ TOBR IF ZERO THEN NON-REG OP
00038 STX R14H INDICATE'PRIOR RESULT REG'
00039 LSR A
00040 LSR A OPCODE*2 TO LSB'S
00041 LSR A
00042 TAY TO Y-REG FOR INDEXING
00043 LDA OPTBL-2,Y LOW-ORDER ADR BYTE
00044 PHA ONTO STACK
00045 RTS GOTO REG-OP ROUTINE
00046 E6 1E 00 00046 TOBR
00047 BNE TOBR2 INCR PC
00048 INC RISH
00049 LDA BRTBL,X LOW-ORDER ADR BYTE
00050 PHA ONTO STACK FOR NON-REG OP
00051 LDA R14H 'PRIOR RESULT REG' INDEX
00052 LSR A PREPARE CARRY FOR BC, BNC.
00053 RTS GOTO NON-REG OP ROUTINE
00054 PLA POP RETURN ADDRESS
00055 RTNZ
00056 PLA
00057 JSR RESTORE RESTORE 6502 REG CONTENTS
00058 JMP (R15L) RETURN TO 6502 CODE VIA PC
00059 SETZ LDA (R15L),Y HIGH-ORDER BYTE OF CONST
00060 STA R0H,X
00061 DEY
00062 LDA (R15L),Y LOW-ORDER BYTE OF CONSTANT
00063 STA R0L,X
00064 TYA Y-REG CONTAINS 1
00065 SEC
00066 ADC R15L ADD 2 TO PC
00067 STA R15L
00068 BCC SET2
00069 INC R15H
00070 RTS
00071 DFB SET-1 (IX)
00072 DFB RTN-1 (0)
00073 DFB LD-1 (2X)
00074 DFB BR-1 (1)
00075 DFB ST-1 (3X)
00076 DFB BNC-1 (2)
00077 DFB LDAT-1 (4X)
00078 DFB BC-1 (3)
00079 DFB STAT-1 (5X)
00080 DFB BP-1 (4)
00081 DFB LDDAT-1 (6X)
00082 DFB BM-1 (5)
00083 DFB STDAT-1 (7X)
00084 DFB EC-1 (6)
00085 DFB POP-1 (8X)
00086 DFB BN2-1 (7)
00087 DFB STPAT-1 (9X)
00088 DFB BU1-1 (8)
00089 DFB ADD-1 (AX)
00090 DFB BN11-1 (9)
00091 DFB SUB-1 (BX)
00092 DFB BK-1 (A)
00093 DFB POPD-1 (CX)
00094 DFB RS-1 (B)
00095 DFB CPR-1 (DX)
00096 DFB BS-1 (C)
00097 DFB INR-1 (EX)
00098 DFB NUL-1 (D)

```

*Listing 2, continued:*

0876: DC	00098	DFB DCR-1	(FX)
0877: SE	00099	DFB NUL-1	(E)
0878: SE	00100	DFB NUL-1	(UNUSED)
0879: SE	00101	DFB NUL-1	(F)
087A: 10 CA	00102 SET	BPL SETZ	ALWAYS TAKEN
087C: BS 00	00103 LD	LDA R0L,X	
	00104 BK	EQU *-1	
087E: 85 00	00105	STA R0L	
0880: B5 01	00106	LDA R0H,X	MOVE RX TO R0
0882: 85 01	00107	STA R0H	
0884: 60	00108	RTS	
0885: A5 00	00109 ST	LDA R0L	
0887: 95 00	00110	STA R0L,X	MOVE R0 TO RX
0889: A5 01	00111	LDA R0H	
088B: 95 01	00112	STA R0H,X	
088D: 60	00113	RTS	
088E: A5 00	00114 STAT	LDA R0L	
0890: B1 00	00115 STAT2	STA (R0L,X)	STORE BYTE INDIRECT
0892: A8 00	00116	LDY #\$0	
0894: B4 1D	00117 STAT3	STY R14H	INDICATE R0 IS RESULT REG
0896: F6 00	00118 INR	INC R0L,X	
0898: D6 02	00119	BNE INR2	INCR RX
089A: F6 01	00120	INC R0H,X	
089C: 60	00121 INR2	RTS	
089D: A1 00	00122 LDAT	LDA (R0L,X)	LOAD INDIRECT (RX)
089F: 85 00	00123	STA R0L	TO R0
08A1: A8 00	00124	LDY #\$0	
08A3: B4 01	00125	STY R0H	ZERO HIGH-ORDER R0 BYTE
08A5: F0 ED	00126	BEO STAT3	ALWAYS TAKEN
08A7: A8 00	00127 POP	LDY #\$0	HIGH ORDER BYTE = 0
08A9: F6 06	00128	BEO POP2	ALWAYS TAKEN
08AB: 20 DD 08	00129 POPD	JSR DCR	DEC RX
08AE: A1 00	00130	LDA (R0L,X)	POP HIGH-ORDER BYTE @RX
08B0: A8	00131	TAY	SAVE IN Y-REG
08B1: 20 DD 08	00132 POP2	JSR DCP	DEC RX
08B4: A1 00	00133	LDA (R0L,X)	LOW-ORDER BYTE
08B6: B5 00	00134	STA R0L	TO R0
08B8: B4 01	00135	STY R0H	
08BA: A8 00	00136 POP3	LDY #\$0	INDICATE R0 AS LAST
08BC: B4 1D	00137	STY R14H	RESULT REG
08BE: 60	00138	RTS	
08BF: 20 90 08	00139 LDDAT	JSDAT	LOW BYTE TO R0, INCR RX
08C2: A1 00	00140	LDA (R0L,X)	HIGH-ORDER BYTE TO R0
08C4: B5 01	00141	STA R0H	
08C6: 4C 96 08	00142	JMP INR	INCR RX
08C9: 20 8E 08	00143 STDAT	JSDAT	STORE INDIRECT LOW-ORDER
08CC: A5 01	00144	LDA R0H	BYTE AND INCR RX. THEN
08CE: B1 00	00145	STA (R0L,X)	STORE HIGH-ORDER BYTE.
08D0: 4C 96 08	00146	JMP INR	INCR RX AND RETURN
08D3: 20 DD 08	00147 STPAT	JSDAT	DEC RX
08D6: A5 00	00148	LDA R0L	
08D8: B1 00	00149	STA (R0L,X)	STORE R0 LOW BYTE @RX
08DA: 4C BA 08	00150	JMP POP3	INDICATE R0 AS LAST
08DD: B5 00	00151 DCR	LDA R0L,X	RESULT REG
08DF: D6 02	00152	BNE DCR2	
08E1: D6 01	00153	DEC R0H,X	DEC RX
08E3: D6 00	00154 DCR2	DEC R0L,X	
08E5: 60	00155	RTS	
08E6: A0 00	00156 SUB	LDY #\$0	RESULT TO R0
08EB: 38	00157 CPR	SEC	NOTE Y-REG = 13*2 FOR CPR
08E9: A5 00	00158	LDA R0L	
08EB: F5 00	00159	SBC R0L,X	
08ED: 99 00 00	00160	STA R0L,Y	R0-RX TO RY
08F0: A5 01	00161	LDA R0H	
08F2: F5 01	00162	SBC R0H,X	
08F4: 99 01 00	00163 SUB2	STA R0H,Y	
08F7: 98	00164	TYA	LAST RESULT REG*2
08FB: 69 00	00165	ADC #\$0	CARRY TO LSB
08FA: B5 1D	00166	STA R14H	
08FC: 60	00167	RTS	
08FD: A5 00	00168 ADD	LDA R0L	
08FF: 75 00	00169	ADC R0L,X	
0901: B5 00	00170	STA R0L	R0+PX TO R0
0903: A5 01	00171	LDA R0H	
0905: 75 01	00172	ADC R0H,X	
0907: A0 00	00173	LDY #\$0	R0 FOR RESULT
0909: F0 E9	00174	BEO SUB2	FINISH ADD
090B: A5 1E	00175 BS	LDA R1SL	NOTE X-REG IS 12*2!
090D: 20 90 08	00176	JSDAT	PUSH LOW PC BYTE VIA R12
0910: A5 1F	00177	LDA P1SH	
0912: 20 90 08	00178	JSDAT	PUSH HIGH-ORDER PC BYTE
0915: 18	00179 BP	CLC	
0916: B0 0E	00180 BNC	BCS BN2	NO CARRY TEST
0918: B1 1E	00181 BRI	LDA (R15L),Y	DISPLACEMENT BYTE
091A: 10 01	00182	BPL BR2	
091C: 88	00183	DEY	
091D: 65 1E	00184 BR2	ADC R1SL	ADD TO PC
091F: B5 1E	00185	STA R1SL	
0921: 98	00186	TYA	
0922: 65 1F	00187	ADC R1SH	
0924: 85 1F	00188	STA P1SH	
0926: 60	00189 BNC2	RTS	
0927: B6 EC	00190 BC	BCS BR	
0929: 60	00191	RTS	
092A: 0A	00192 BP	A	DOUBLE RESULT-REG INDEX
092B: AA	00193	TAX	TO X-REG FOR INDEXING
092C: B5 01	00194	LDA R0H,X	TEST FOR PLUS
092E: 10 E8	00195	BPL BRI	BRANCH IF SO
0930: 60	00196	RTS	
0931: 0A	00197 BM	ASL A	DOUBLE RESULT-REG INDEX
0932: AA	00198	TAX	TEST FOR MINUS
0933: B5 01	00199	LDA R0H,X	
0935: 30 E1	00200	BMI BRI	
0937: 60	00201	RTS	
0938: 0A	00202 BZ	ASL A	DOUBLE RESULT-REG INDEX
0939: AA	00203	TAX	TEST FOR ZERO
093A: B5 00	00204	LDA R0L,X	(BOTH BYTES)
093C: 15 01	00205	ORA R0H,X	
093E: F0 DB	00206	BNE BRI	BRANCH IF SO
0940: 60	00207	RTS	
0941: 0A	00208 BN2	ASL A	DOUBLE RESULT-REG INDEX
0942: AA	00209	TAX	TEST FOR NONZERO
0943: B5 00	00210	LDA R0L,X	(BOTH BYTES)
0945: 15 01	00211	ORA R0H,X	
0947: D0 CF	00212	BNE BRI	BRANCH IF SO
0949: 60	00213	RTS	
094A: 0A	00214 BMI	ASL A	DOUBLE RESULT-REG INDEX
094B: AA	00215	TAX	CHECK BOTH BYTES
094C: B5 00	00216	LDA R0L,X	FOR \$FF (MINUS 1)
094E: 35 01	00217	AND R0H,X	

memory locations like 6502 instructions. The main loop at SW16B repeatedly calls the "execute instruction" routine at SW16C which examines one op code for type and branches to the appropriate subroutine to execute it.

Subroutine SW16C increments the program counter (R15) and fetches the next op code which is either a register operation of the form OP REG (2 hexadecimal digits) with OP between hexadecimal 1 and F, or a nonregister operation of the form 0 OP with OP between hexadecimal 0 and D. Assuming a register operation, the register specification is doubled to account for the 2 byte SWEET16 registers and placed in the X register for indexing. Then the instruction type is determined. Register operations place the doubled register specification in the high order byte of R14 indicating the "prior result register" to subsequent branch instructions. Nonregister operations treat the register specification (right-hand half-byte) as their op code, increment the SWEET16 PC to point at the displacement byte of branch instructions, load the A-Reg with the "prior result register" index for branch condition testing, and clear the Y-Reg.

### When Is an RTS Really a JSR?

Each instruction type has a corresponding subroutine. The subroutine entry points are stored in a table which is directly indexed by the op code. By assigning all the entries to a common page, only a single byte of address need be stored per routine. The 6502 indirect jump might have been used as follows to transfer control to the appropriate subroutine:

LDA #ADR	High order address byte
STA IND+1	
LDA OPTBL,X	Low order byte
STA IND	
JMP (IND)	

To save code the subroutine entry address (minus 1) is pushed onto the stack, high order byte first. A 6502 RTS (ReTurn from Subroutine) is used to pop the address off the stack and into the 6502 program counter (after incrementing by 1). The net result is that the desired subroutine is reached by executing a subroutine return instruction! [This ironic situation is an example of what is commonly referred to as "cleverness."]

### Op Code Subroutines

The register operation routines make use of the 6502 "zero page indexed by X" and "indexed by X indirect" addressing modes to access the specified registers and indirect data. The "result" of most register ops is left

# The Art of Computer Programming

Praised by many critics as the best books in their field, **The Art of Computer Programming, Volumes I, II and III**, are part of a projected seven volume omnibus survey of computer science now being completed by Donald E Knuth.

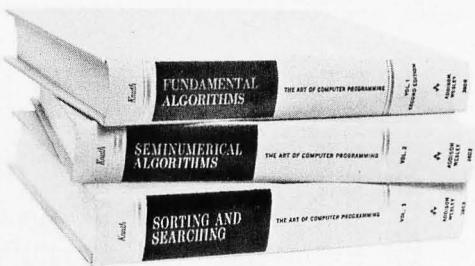
—Volume I, **Fundamental Algorithms**, begins with a thorough discussion of the mathematics used in computer programming, followed by a treatment of information structures, stacks, arrays, linked lists, dynamic storage allocation, and trees. 634 pp; \$20.95.

—Volume II, **Seminumerical Algorithms**, is concerned with random numbers, statistical tests, random sequences, as well as arithmetic (floating point and multiple precision), polynomials, and rational arithmetic. 624 pp; \$20.95.

—Volume III deals with **Searching and Sorting**, and as the name implies, the emphasis is on algorithms for sorting, including combinatorial properties of permutations, internal sorting, optimum sorting, and external sorting. Also included is a section on sequential searching, hashing, digital searching, and more. 722 pp; \$20.95.

A hypothetical assembly language called MIX has been developed by the author to illustrate programming examples throughout the series. MIX is easily convertible to other assembly languages.

Prof Knuth writes with style and wit (among many memorable quotes is one from *McCall's Cookbook!*). This classic work belongs on the reference shelf of everyone seriously interested in computer science.

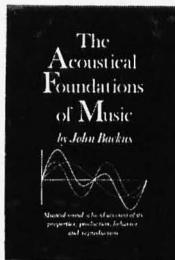
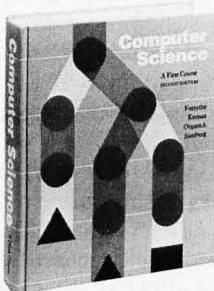


—**The Acoustical Foundations of Music**, subtitled *Musical Sound: a lucid account of its properties, production, behavior, and reproduction*, by John Backus. One of the problems involved in trying to produce music by computer is the scarcity of useful reference material on the basic theory of sound generation and musical acoustics. John Backus has filled this gap with **The Acoustical Foundations of Music**, a readable and informative guide that covers the physiological properties of sound, the ear and its perception of sounds, the effect of acoustic environment, the acoustical behavior of musical instruments, and the various applications of electronics and computers to the production, reproduction and composition of music.

The book assumes some slight knowledge of music on the part of the reader, but covers enough elementary physics and acoustics to orient the reader who has no knowledge of physics.

This clear and lively exposition of music's physical basis will be invaluable to all those experimenters interested in creating music by computer. 312 pages, and only \$9.95 in hardcover.

—**Your Home Computer**, by James White, is a clearly written nontechnical description of personal computers that requires no prior knowledge of computers or electronics. The emphasis is on understanding; over 100 illustrations are included. Topics include: computing and you; communication inside a computer; computer thought processes; fixed memory; inputs and outputs; peripherals; systems components; how to choose a microcomputer; and so on. **Your Personal Computer** is the ideal book for readers who thought they could never understand how computers work. And the best part is that it's easy and fun to read. Yours for \$6.



—**Computer Science — A First Course**. The title of this book belies its thoroughness and rigor. **Computer Science: A First Course** is one of the most complete and well laid out computer science books we've seen in quite a while. This 760+ page book is for the serious experimenter who wants a definitive treatment of every aspect of software design and development. Chapters deal with algorithms, flowcharts, looping structures, stepwise decomposition, trees, storage concepts, interpreting and compiling, data processing, numerical processing, character strings, and a discussion of SAMOS, a special hypothetical computer system used throughout the book to illustrate computer science principles. **Computer Science: A First Course** makes an excellent source book. \$16.95, Hardcover.

—**Understanding Microcomputers and Small Computer Systems** by Nat Wadsworth. What does "hexadecimal" mean? What is Boolean Logic? What's the difference between a PROM and an EROM? These questions and more are answered in Nat Wadsworth's **Understanding Microcomputers and Small Computer Systems**. You'll learn about the basic operation of a microcomputer, the types of instructions that microcomputers are able to carry out, machine language and higher level language programming, input and output devices, etc. The book concludes with a discussion of system considerations. The combined index and glossary is a useful bonus. If you're a beginner in the personal computing field, get a copy today! This 300 page book is only \$9.95.

Send to:  
**BITS, Inc**  
70 Main Street  
Peterborough NH 03458

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip Code \_\_\_\_\_

Signature \_\_\_\_\_

You may photocopy this page if you wish to leave your BYTE intact.

Check Payment method:



My check is enclosed \_\_\_\_\_  
 Bill my MC No. \_\_\_\_\_ Exp. date \_\_\_\_\_  
 Bill my BAC No. \_\_\_\_\_ Exp. date \_\_\_\_\_

Total for all books checked \$ \_\_\_\_\_

Postage, 50 cents per book for \_\_\_\_\_ books \$ \_\_\_\_\_

Grand Total \$ \_\_\_\_\_

Prices shown are subject to change without notice.  
All orders must be prepaid.  
In unusual cases, processing may exceed 30 days.

*Listing 2, continued:*

```

0950: 49 FF    00218      EOR #$FF
0952: F0 C4    00219      BEQ BRI      BRANCH IF SO
0954: 60        00220      RTS
0955: AA        00221 BNMI     ASL A      DOUBLE RESULT+REG INDEX
0956: AA        00222      TAX
0957: B5 00    00223      LDA R0L,X
0959: 35 01    00224      AND R0H,X      CHK BOTH BYTES FOR NO $FF
095B: 49 FF    00225      EOR #$FF
095D: D0 B9    00226      BNE BRI      BRANCH IF NOT MINUS 1
095F: 60        00227 NUL     RTS
0960: A2 18    00228 RS     LDX #$18    12*2 FOR R12 AS STK PTR
0962: 20 DD 08 00229      JSR DCR      DECR STACK POINTER
0965: A1 00    00230      LDA (R0L,X)  POP HIGH RETURN ADR TO PC
0967: 85 1F    00231      STA R1SH
0969: 20 DD 08 00232      JSR DCR      SAME FOR LOW-ORDER BYTE
096C: A1 00    00233      LDA (R0L,X)
096E: 85 1E    00234      STA R1SL
0970: 60        00235      RTS
0971: 4C 3E 08 00236 RTN     JMP RTNZ
0972:          00237 *
0973:          00238 * REG SAVE/RESTORE ROUTINES
0974:          00239 * FOR NON-APPLE-II SYSTEMS
0975:          00240 *
0976:          00241 ASA9     EPZ $45
0977:          00242 XSAV     EPZ $46
0978:          00243 YSAV     EPZ $47
0979:          00244 PSAV     EPZ $48
097A:          00245 SAVE     STA ASA9
097B:          00246 STX XSAV     STA XSAV   SAVE 6502 REG CONTENTS.
097C:          00247 STY YSAV     STA YSAV
097D:          00248 PHP
097E:          00249 PLA
097F:          00250 STA PSAV
0980:          00251 RTS
0981:          00252 RESTORE   LDA PSAV
0982:          00253 PHA
0983:          00254 LDA ASA9
0984:          00255 LDX XSAV     LDX XSAV   RESTORE 6502 REG CONTENTS.
0985:          00256 LDY YSAV     LDY YSAV
0986:          00257 PLP
0987:          00258 RTS
0988:          00259
0989:          00260
0990:          00261
0991:          00262
0992:          00263
0993:          00264
0994:          00265
0995:          00266
0996:          00267
0997:          00268
0998:          00269
0999:          00270
099A:          00271
099B:          00272
099C:          00273
099D:          00274
099E:          00275
099F:          00276

```

Table 1:

### SWEET16 OP CODE SUMMARY

#### Register Ops

1n SET	Rn	Constant (Set)
2n LD	Rn	(Load)
3n ST	Rn	(Store)
4n LD	@Rn	(Load indirect)
5n ST	@Rn	(Store indirect)
6n LDD	@Rn	(Load double indirect)
7n STD	@Rn	(Store double indirect)
8n POP	@Rn	(Pop indirect)
9n STP	@Rn	(Store pop indirect)
An ADD	Rn	(Add)
Bn SUB	Rn	(Sub)
Cn POPD	@Rn	(Pop double indirect)
Dn CPR	Rn	(Compare)
En INR	Rn	(Increment)
Fn DCR	Rn	(Decrement)

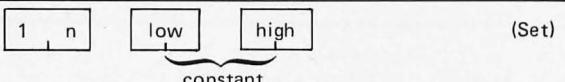
#### Nonregister Ops

00	RTN	(Return to 6502 mode)
01	BR ea	(Branch always)
02	BNC ea	(Branch if No Carry)
03	BC ea	(Branch if Carry)
04	BP ea	(Branch if Plus)
05	BM ea	(Branch if Minus)
06	BZ ea	(Branch if Zero)
07	BNZ ea	(Branch if NonZero)
08	BM1 ea	(Branch if Minus 1)
09	BNM1 ea	(Branch if Not Minus 1)
0A	BK ea	(Break)
0B	RS	(Return from Subroutine)
0C	BS ea	(Branch to Subroutine)
0D		(Unassigned)
0E		(Unassigned)
0F		(Unassigned)

**SWEET16 Operation Code Summary:** Table 1 summarizes the list of SWEET16 operation codes, which are explained in further detail one by one in the descriptions which follow the table. The program of listing 2 implements the execution of these interpretive codes after a call to the entry point SW16. Return to the calling program and normal noninterpretive operation is accomplished with the RTN mnemonic of SWEET16.

### SWEET16 – REGISTER OPERATIONS

SET Rn, Constant



The 2 byte constant is loaded into Rn (n = 0 to F, hexadecimal) and branch conditions set accordingly. The carry is cleared.

Example:

15 34 A0      SET    R5, A034      R5 now contains A034

LD Rn



The ACC (R0) is loaded from Rn and branch conditions set according to the data transferred. The carry is cleared and the contents of Rn are not disturbed.

Example:

15 34 A0      SET    R5, A034  
25                LD     R5              ACC now contains A034

in the specified register and can be sensed by subsequent branch instructions since the register specification is saved in the high order byte of R14. This specification is changed to indicate R0 (ACC) for ADD and SUB instructions and R13 for the CPR (compare) instruction.

Normally the high order R14 byte holds the "prior result register" index times 2 to account for the 2 byte SWEET16 registers, and thus the least significant bit is zero. If ADD, SUB or CPR instructions generate carries, then this index is incremented, setting the least significant bit, which becomes a carry flag.

The SET instruction increments the program counter twice, picking up data bytes for the specified register. In accordance with 6502 convention, the low order data byte precedes the high order byte.

Most SWEET16 nonregister operations are relative branches. The corresponding subroutines determine whether or not the "prior result" meets the specified branch condition and if so update the SWEET16 program counter by adding the displacement value (-128 to +127 bytes).

The RTN operation restores the 6502 register contents, pops the subroutine return stack and jumps indirect through the SWEET16 program counter register. This transfers control to the 6502 at the instruction immediately following the RTN instruction.

The BK operation actually executes a 6502 break instruction (BRK), transferring control to the interrupt handler.

Any number of subroutine levels may be implemented within SWEET16 code via the BS (Branch to Subroutine) and RS (Return from Subroutine) instructions. The user must initialize and otherwise not disturb R12 if the SWEET16 subroutine capability is used since it is utilized as the automatic subroutine return stack pointer.

### Memory Allocation and User Modifications

The only storage that must be allocated for SWEET16 variables are 32 consecutive locations in page zero for the SWEET16 registers, four locations to save the 6502 register contents, and a few levels of the 6502 subroutine return address stack. If you don't need to preserve the 6502 register contents, delete the SAVE and RESTORE subroutines and the corresponding subroutine calls. This will free the four page zero locations ASA9, XSAV, YSAV and PSAV.

You may wish to add some of your own

Text continued on page 159

# FINALLY.

## A State-of-the-Art Tool For Learning Software Design.

And at an affordable price. The Modu-Learn™ home study course from Logical Services.

Now you can learn microcomputer programming in ten comprehensible lessons. At home. In your own time. At your own pace.

You learn to solve complex problems by breaking them down into easily programmed modules. Prepared by professional design engineers, the Modu-Learn™ course presents systematic software design techniques, structured program design, and practical examples from real 8080A micro-computer applications. All in a modular sequence of 10 lessons . . . more than 500 pages, bound into one practical notebook for easy reference.

You get diverse examples, problems, and solutions. With thorough background material on micro-computer architecture, hardware/software trade-offs, and useful reference tables. All for only \$49.95.

For \$49.95 you learn design techniques that make software work for you. Modu-Learn™ starts with the basics. Our problem-solution approach enables you to "graduate" as a programmer.

See Modu-Learn™ at your local computer store or order now using the coupon below.

Please send the Modu-Learn™ course for me to examine. Enclosed is \$49.95 (plus \$2.00 postage and handling) or my Mastercharge/BankAmericard authorization.

Name: \_\_\_\_\_

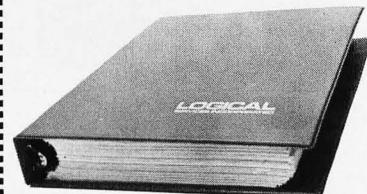
Address: \_\_\_\_\_

City: \_\_\_\_\_ State: \_\_\_\_\_

Card #: \_\_\_\_\_

Expiration date: \_\_\_\_\_

Signature: \_\_\_\_\_



711 Stierlin Road  
Mountain View, CA 94043  
(415) 965-8365

**LOGICAL**  
SERVICES INCORPORATED

ST Rn      

3	n
---	---

      (Store)

The ACC (R0) is stored into Rn and branch conditions set according to the data transferred. The carry is cleared and the ACC contents are not disturbed.

Example:

25	LD	R5	Copy the contents
36	ST	R6	of R5 to R6.

LD @Rn      

4	n
---	---

      (Load indirect)

The low order ACC byte is loaded from the memory location whose address resides in Rn, and the high order ACC byte is cleared. Branch conditions reflect the final ACC contents which will always be positive and never minus 1. The carry is cleared. After the transfer, Rn is incremented by 1.

Example:

15 34 A0	SET	R5, A034	ACC is loaded from
45	LD	@R5	memory location A034
			and R5 is incremented
			to A035.

ST @Rn      

5	n
---	---

      (Store indirect)

The low order ACC byte is stored into the memory location whose address resides in Rn. Branch conditions reflect the 2 byte ACC contents. The carry is cleared. After the transfer, Rn is incremented by 1.

Example:

15 34 A0	SET	R5, A034	Load pointers R5 and R6
16 22 90	SET	R6, 9022	with A034 and 9022.
45	LD	@R5	Move a byte from location
56	ST	@R6	A034 to location 9022. Both
			pointers are incremented.

LDD @Rn      

6	n
---	---

      (Load double byte indirect)

The low order ACC byte is loaded from the memory location whose address resides in Rn, and Rn is then incremented by 1. The high order ACC byte is loaded from the memory location whose address resides in the (incremented) Rn and Rn is again incremented by 1. Branch conditions reflect the final ACC contents. The carry is cleared.

Example:

15 34 A0	SET	R5, A034	The low order ACC byte is
65	LDD	@R5	loaded from location A034,
			the high order byte from
			location A035. R5 is incre-
			mented to A036.

STD @Rn      

7	n
---	---

      (Store double byte indirect)

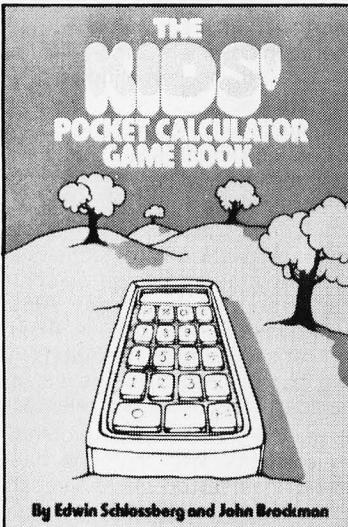
The low order ACC byte is stored into the memory location whose address resides in Rn, and Rn is then incremented by 1. The high order ACC byte is stored into the memory location whose address resides in the (incremented) Rn and Rn is again incremented by 1. Branch conditions reflect the ACC contents which are not disturbed. The carry is cleared.

Example:

15 34 A0	SET	R5, A034	Load pointers R5 and R6
16 22 90	SET	R6, 9022	with A034 and 9022. Move
65	LDD	@R5	double byte from locations
76	STD	@R6	A034 and A035 to locations
			9022 and 9023. Both pointers
			are incremented by 2.

# IT ALL ADDS UP TO EDUCATIONAL **FUN**

The creators of the original Pocket Calculator Game Book now present two fun-filled new game books for use with that incredible machine that has found a place in almost every home



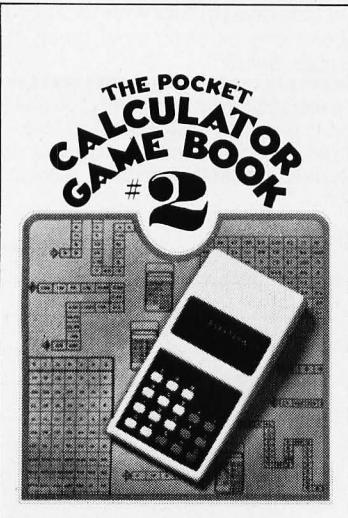
By Edwin Schlossberg and John Brockman

## THE KIDS' POCKET CALCULATOR GAME BOOK

by Edwin Schlossberg and

John Brockman

A quick trip through elementary mathematics—fun and games with real purpose. The first book of its kind for kids from kindergarten through college. Illustrated with line drawings and cartoons. \$6.95 hardcover    \$3.95 paperbound



## THE POCKET CALCULATOR GAME BOOK #2

by Edwin Schlossberg and  
John Brockman

Even more popular in approach than its famous predecessor, this book is simpler, more accessible, and its games are more mathematically basic. Illustrated with line drawings and cartoons.

\$6.95 hardcover    \$3.95 paperbound

**WILLIAM MORROW**

POP @Rn

8 n

(Pop indirect)

The low order ACC byte is loaded from the memory location whose address resides in Rn after Rn is decremented by 1 and the high order ACC byte is cleared. Branch conditions reflect the final 2 byte ACC contents which will always be positive and never minus 1. The carry is cleared. Because Rn is decremented prior to loading the ACC, single byte stacks may be implemented with the ST @Rn and POP @Rn operations (Rn is the stack pointer).

Example:

15 34 A0	SET R5, A034	Init stack pointer.
10 04 00	SET R0, 4	Load 4 into ACC.
35	ST @R5	Push 4 onto stack.
10 05 00	SET R0, 5	Load 5 into ACC.
35	ST @R5	Push 5 onto stack.
10 06 00	SET R0, 6	Load 6 into ACC.
35	ST @R5	Push 6 onto stack.
85	POP @R5	Pop 6 off stack into ACC.
85	POP @R5	Pop 5 off stack.
85	POP @R5	Pop 4 off stack.

STP @Rn

9 n

(Store pop indirect)

The low order ACC byte is stored into the memory location whose address resides in Rn after Rn is decremented by 1. Then the high order ACC byte is stored into the memory location whose address resides in Rn after Rn is again decremented by 1. Branch conditions will reflect the 2 byte ACC contents which are not modified. STP @Rn and PLA @Rn are used together to move data blocks beginning at the greatest address and working down. Additionally, single byte stacks may be implemented with the STP @Rn and LDA @Rn ops.

Example:

14 34 A0	SET R4, A034	Init pointers.
15 22 90	SET R5, 9022	
84	POP @R4	Move byte from A033 to 9021.
95	STP @R5	Move byte from A032 to 9020.
84	POP @R4	
95	STP @R5	

ADD Rn

A n

(Add)

The contents of Rn are added to the contents of the ACC (R0) and the low order 16 bits of the sum restored in ACC. The 17th sum bit becomes the carry and other branch conditions reflect the final ACC contents.

Example:

10 34 76	SET R0, 7634	Init R0 (ACC)
11 27 42	SET R1, 4227	and R1.
A1	ADD R1	Add R1 (sum = B85B, carry clear)
A0	ADD R0	Double ACC (R0) to 70B6 with carry set.

SUB Rn

B n

(Subtract)

The contents of Rn are subtracted from the ACC contents by performing a two's complement addition:

$$\text{ACC } \text{ACC} + \overline{\text{Rn}} + 1$$

The low order 16 bits of the subtraction are restored in the ACC. The 17th sum bit becomes the carry and other branch conditions reflect the final ACC contents. If the 16 bit unsigned ACC contents are greater than or equal to the 16 bit unsigned Rn contents then the carry is set, otherwise it is cleared. Rn is not disturbed.

Example:

10 34 76	SET R0, 7634	Init R0 (ACC)
11 27 42	SET R1, 4227	and R1.
A1	SUB R1	Subtract R1 (diff = 340D with carry set)
A0	SUB R0	Clears ACC (R0)



## POPD @Rn

C n

(POP Double byte indirect)

Rn is decremented by 1 and the high order ACC byte is loaded from the memory location whose address now resides in Rn. Then Rn is again decremented by 1 and the low order ACC byte is loaded from the corresponding memory location. Branch conditions reflect the final ACC contents. The carry is cleared. Because Rn is decremented *prior* to loading each of the ACC halves, double byte stacks may be implemented with the STD @ Rn and POPD @ Rn operations. (Rn is the stack pointer).

**Example:**

15 34 A0	SET R5, A034	Init stack pointer.
10 12 AA	SET R0, AA12	Load AA12 into ACC.
75	STD @R5	Push AA12 onto stack.
10 34 BB	SET R0, BB34	Load BB34 into ACC.
75	STD @R5	Push BB34 onto stack.
10 56 CC	SET R0, CC56	Load CC56 into ACC.
75	STD @R5	Pop CC56 off stack.
C5	POPD @R5	Pop BB34 off stack.
C5	POPD @R5	Pop AA12 off stack.

## CPR Rn

D n

(Compare)

The ACC (R0) contents are compared to Rn by performing the 16 bit binary subtraction ACC-Rn and storing the low order 16 difference bits in R13 for subsequent branch tests. If the 16 bit unsigned ACC contents are greater than or equal to the 16 bit unsigned Rn contents then the carry is set, otherwise it is cleared. No other registers, including ACC and Rn, are disturbed.

**Example:**

15 34 A0	SET R5, A034	Pointer to memory.
16 BF A0	SET R6, A0BF	Limit address.
10 00 00	LOOP SET R0, 0	Zero data.
75	STD @R5	Clear 2 locs, incr R5 by 2.
25	LD R5	Compare pointer R5 to limit R6.
D6	CPR R6	Loop if carry clear.
02 F8	BNC LOOP	

## INR Rn

E n

(Increment)

The contents of Rn are incremented by 1. The carry is cleared and other branch conditions reflect the incremented value.

**Example:**

15 34 A0	SET R5, A034	Init R5 (pointer)
10 00 00	SET R0, 0	Zero to R0.
55	ST @R5	Clears loc A034 and incrs R5 to A035.
E5	INR R5	Incr R5 to A036
55	ST @R5	Clears loc A036 (not A035)

## DCR Rn

F n

(Decrement)

The contents of Rn are decremented by 1. The carry is cleared and other branch conditions reflect the decremented value.

**Example:** (Clear nine bytes beginning at loc A034)

15 34 A0	SET R5, A034	Init pointer.
14 09 00	SET R4, 9	Init count.
10 00 00	SET R0, 0	Zero ACC.
55	LOOP ST @R5	Clear a mem byte.
F4	DCR R4	Decr count.
07 FC	BNZ LOOP	Loop until zero.

## SWEET16 Nonregister Instructions

## RTN

O O

(Return to 6502 mode)

Control is returned to the 6502 and program execution continues at the location immediately following the RTN instruction. The 6502 registers and status conditions are restored to their original contents (prior entering SWEET16 mode).

**BYTE SHOP**  
the affordable computer store™

## WARBLE ALARM CAR-VAN CLOCK WITH HEADLIGHT ALARM



**COMPLETE KIT \$35.95  
ASSEMBLED \$45.95**

- ELAPSED TIMER
- SECONDS DISPLAY SWITCH
- 9 MINUTE SNOOZE ALARM
- SIMPLE 4 WIRE HOOK-UP
- JUMBO 1/2" LED DISPLAY
- 1 TO 59 MINUTE COUNTDOWN TIMER RUNS SIMULTANEOUSLY WITH CLOCK!
- RUGGED ABS CASE
- QUARTZ CRYSTAL ACCURACY

### DIGITAL AUTO INSTRUMENTS #1 TACHOMETER SEVEN MODELS!



**#2 WATER TEMP.  
#3 FUEL LEVEL  
#4 SPEEDOMETER\*  
#5 OIL PRESSURE  
#6 OIL TEMP.  
#7 BATTERY MONITOR**

\*ADD \$10 FOR REQUIRED SPEED SENDER. \$15 FOR SPEED SENDER ALONE

**KIT: \$49.95. . . . . ASSEMBLED: \$59.95**

### ELECTRONIC 'PENDULUM' CLOCK



- SWING PENDULUM
- 7" HOURS AND MINUTES DISPLAY
- TIME SET PUSH BUTTONS
- ALARM FEATURE

**KIT-UNFINISHED CASE \$59.95  
ASSEMBLED-STAINED CASE \$69.95**

### QUARTZ DIGITAL AUTO CLOCK OR ELAPSED TIMER!

ELAPSED TIMER: HRS, MINS & SEC'S  
SIMPLE PUSHBUTTON RESET &  
HOLD TOGGLE SWITCH

KIT INCLUDES EVERYTHING,  
NOTHING ELSE TO BUY! 4" LED'S!  
INTERNAL BATTERY BACKUP!  
NON POLAR INPUT!

12 OR 24 HR MODE  
DIMENSIONS: 4 1/2" x 4" x 2"

**KIT: \$27.95. . . . . ASSEMBLED: \$37.95**

**NOW WITH  
ELAPSED  
TIME!**

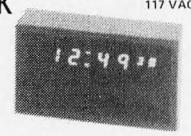


### 3 1/2 DIGITAL CLOCK

- 4 DIGIT KIT \$49.95 • 4 DIGIT ASSEMBLED \$59.95
- 6 DIGIT KIT \$69.95 • 6 DIGIT ASSEMBLED \$79.95

117 VAC - 12 OR 24 HR MODE . . . . . KIT COMES COMPLETE!  
6 DIGIT VERSION: 27" x 5" x 1 1/2" . . . . . 4 DIGIT VERSION: 18" x 1 1/2"

### TV-WALL CLOCK



117 VAC

- 25' VIEWING DISTANCE
- .6" HOURS & MINUTES
- .3" SECONDS
- COMPLETE WITH WOOD CASE

**KIT: \$34.95. . . . . ASSEMBLED: \$39.95**

### ECONOMY CAR CLOCK



1/2" LED MODULE!

- COMPLETE WITH CASE,  
BRACKET & TIME SET  
PUSHBUTTONS

• ALARM OPTION

**KIT: \$19.95. . . . . ASSEMBLED: \$26.95**

### PENDULUM

GIVE YOUR DIGITAL CLOCK A PENDULUM SWING  
9 TO 12V DC. 60 HZ INPUT  
SIMPLE HOOK UP TO ANY CLOCK



**\$14.95**

CASE WITH BRACKET \$3.75

### MARK FOSKETS'

SOLID STATE TIME

P.O. BOX 2159

DUBLIN, CALIF. 94866

ORDERS (415) 828-1923



24 HR  
PHONE



CALIFORNIA RESIDENTS - ADD 6% SALES TAX

BR ea

0 | 1      d | d

(Branch Always)

An effective address (ea) is calculated by adding the signed displacement byte (dd) to the program counter. The program counter contains the address of the instruction immediately following the BR, or the address of the BR operation plus 2. The displacement is a signed two's complement value from -128 to +127. Branch conditions are not changed. Note that effective address calculation is identical to that for 6502 relative branches.

#### Some examples:

dd = \$80	ea = PC + 2 - 128
dd = \$81	ea = PC + 2 - 127
dd = \$FF	ea = PC + 2 - 1
dd = \$00	ea = PC + 2 + 0
dd = \$01	ea = PC + 2 + 1
dd = \$7E	ea = PC + 2 + 126
dd = \$7F	ea = PC + 2 + 127

#### Example:

\$300: 01 50      BR \$352

BNC ea

0 | 2      d | d

(Branch if No Carry)

A branch to the effective address is taken only if the carry is clear, otherwise execution resumes as normal with the next instruction. Branch conditions are not changed.

BC ea

0 | 3      d | d

(Branch if Carry set)

A branch is effected only if the carry is set. Branch conditions are not changed.

BP ea

0 | 4      d | d

(Branch if Plus)

A branch is effected only if the prior "result" (or most recently transferred data) was positive. Branch conditions are not changed.

#### Example: (Clear mem from loc A034 to A03F)

15 34 A0	SET R5, A034	Init pointer.
14 3F A0	SET R4, A03F	Init limit.
10 00 00	LOOP SET R0, 0	
55	ST @R5	Clear mem byte, incr R5.
24	LD R4	Compare limit to
D5	CPR R5	pointer.
04 F8	BP LOOP	Loop until done.

BM ea

0 | 5      d | d

(Branch if Minus)

A branch is effected only if the prior "result" was minus (negative, MSB = 1). Branch conditions are not changed.

BZ ea

0 | 6      d | d

(Branch if Zero)

A branch is effected only if the prior "result" was zero. Branch conditions are not changed.

BNZ ea

0 | 7      d | d

(Branch if NonZero)

A branch is effected only if the prior "result" was nonzero. Branch conditions are not changed.

BM1 ea

0 | 8      d | d

(Branch if Minus 1)

A branch is effected only if the prior "result" was minus 1 (\$FFFF hexadecimal). Branch conditions are not changed.

BNM1 ea

0 | 9      d | d

(Branch if Not Minus 1)

A branch is effected only if the prior "result" was not minus 1 (\$FFFF hexadecim). Branch conditions are not changed.

Text continued from page 154

instructions to this implementation of SWEET16. If you use the unassigned op codes \$0E and \$0F, remember that SWEET16 treats these as 2 byte instructions. You may wish to handle the break instruction as a SWEET16 call, saving two bytes of code each time you transfer into SWEET16 mode. Or you may wish to use the SWEET16 BK (Break) operation as a "CHAROUT" call in the interrupt handler. You can perform absolute jumps within SWEET16 by loading the ACC (R0) with the address you wish to jump to (minus 1) and executing a ST R15 instruction.

And as a final thought, the ultimate modification for those who do not use the 6502 processor would be to implement a version of SWEET16 for some other microprocessor design. The idea of a low level interpretive processor can be fruitfully implemented for a number of purposes, and achieves a limited sort of machine independence for the interpretive execution strings. I found this technique most useful for the implementation of much of the software of the Apple II computer; I leave it to readers to explore further possibilities for SWEET16. ■

BRK

0 A

(Break)

A 6502 BRK (break) instruction is executed. SWEET16 may be reentered non-destructively at SW16D after correcting the stack pointer to its value prior to executing the BRK.

RS

0 B

(Return from SWEET16 Subroutine)

RS terminates execution of a SWEET16 subroutine and returns to the SWEET16 calling program which resumes execution (in SWEET16 mode). R12, which is the SWEET16 subroutine return stack pointer, is decremented twice. Branch conditions are not changed.

BS ea

0 C d d

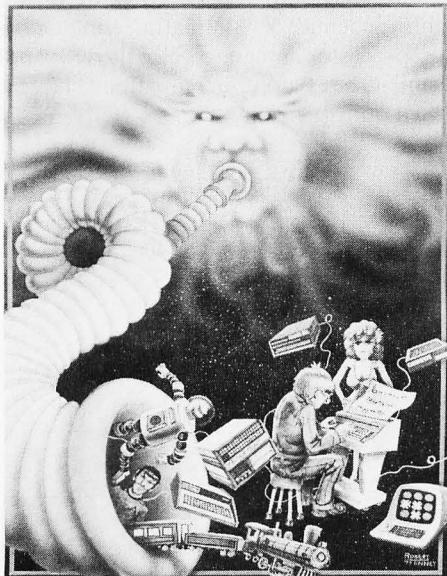
Branch to SWEET16 Subroutine

A branch to the effective address (PC + 2 + d) is taken and execution is resumed in SWEET16 mode. The current PC is pushed onto a "SWEET16 subroutine return address" stack whose pointer is R12, and R12 is incremented by 2. The carry is cleared and branch conditions set to indicate the current ACC contents.

Example: (Calling a "memory move" subroutine to move A034-A03B to 3000-3007)

300: 15 34 A0	SET	R5, A034	Init pointer 1.
303: 14 3B A0	SET	R4, A03B	Init limit 1.
306: 16 00 30	SET	R6, 3000	Init pointer 2.
309: 0C 15	BS	MOVE	Call move subroutine.
.			
320: 45	MOVE	LD @R5	Move one
321: 56		ST @R6	byte.
322: 24		LD R4	
323: D5		CPR R5	Test if done.
324: 04 FA		BP MOVE	
326: 0B		RS	Return.

## The Best of BYTE, Volume 1



Send now to:

BYTE Interface Technical Services, Inc.  
70 Main St  
Peterborough NH 03458

The volume we have all been waiting for! The answer to those unavailable early issues of BYTE. Best of BYTE, edited by Carl Helmers Jr and David Ahl. This 384 page book is packed with a majority of material from the first 12 issues. Included are 146 pages devoted to "Hardware" and how-to articles ranging from TV displays to joysticks to cassette interfaces, along with a section devoted to kit building which describes seven major kits. "Software and Applications" is the other side of the coin: on-line debuggers to games to a complete small business accounting system is included in this 125 page section. A section on "Theory" examines the how and why behind the circuits and programs. "Opinion" closes the book with a look ahead, as to where this new hobby is heading. It is now available through BITS Inc for only \$11.95 and 50 cents postage.

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_

State \_\_\_\_\_

Zip \_\_\_\_\_

Price of Book \$ \_\_\_\_\_

Postage, 50 cents \$ \_\_\_\_\_

Total \$ \_\_\_\_\_

Check enclosed

  Bill MC # \_\_\_\_\_ Exp. Date \_\_\_\_\_

  Bill BA # \_\_\_\_\_ Exp. Date \_\_\_\_\_

Signature \_\_\_\_\_

In unusual cases, processing may exceed 30 days.

All orders must be prepaid.

You may photocopy this page if you wish to leave your BYTE intact.