

[\[Return to Main Page\]](#)

Porting Sweet 16 by Carsten Strotmann (carsten@strotmann.de)

[\[Up to Source Code Repository\]](#)

Sweet 16 is a metaprocessor or "pseudo microprocessor" implemented in 6502 assembly language. Originally written by Steve Wozniak and used in the Apple II, Sweet 16 can also be ported to other 6502-based systems to provide useful 16-bit functionality. This article includes the source code for Sweet 16, along with a brief history, programming instructions, and notes to help port it.

This material was provided by Carsten Strotmann, who has built a working Atari Port of Sweet 16, [for BIBO](#) and [for Mac/65](#) assemblers. See also [the appendix](#) for material added later, after [a discussion on the 6502 forum](#).

- [Porting Sweet 16](#)
- [The Story of Sweet 16 \(as told by Woz\)](#)
- [Source code \(Sweet 16\)](#)
- [Sweet 16 - Introduction](#)
- [Sweet 16: A Pseudo 16-bit Microprocessor](#)
 - [Description](#)
 - [Instruction Descriptions](#)
 - [Sweet 16 Opcode Summary](#)
 - [Register Instructions](#)
 - [Theory of Operation](#)
 - [When is an RTS really a JSR?](#)
 - [Opcode Subroutines](#)
 - [Memory Allocation](#)
 - [User Modifications](#)
 - [Notes from Apple II System Description](#)
- [Sweet 16 S-C Macro Assembler Text](#)
 - [Programming Model](#)
 - [Sweet 16 Opcodes](#)
- [Appendix: Additional Material](#)

Porting Sweet 16

Sweet 16 is not a teen-magazine, nor is it a brand name for candy. It is a really tricky und useful extension to a 6502 computer. It was originally written by Apple's Steve "Woz" Wozniak and can be found in the ROM of some APPLE II Computers. Sweet 16 is a kind of virtual machine that gives the 6502 programmer a 16 bit extension to the CPU. Sweet 16 creates sixteen 16-bit registers/pointers (in zero page) and new opcodes to use this registers. Although Sweet 16 is not as fast as standard 6502 code, it can reduce the code size of your programmes and ease programming.

This text focuses only on the task of porting Sweet 16 to another 6502 based computer. Information on the useage and the interals of Sweet 16 can be found in the documentation by Steve Wozniak and the article by Dick Sedgewick.

Porting Sweet 16 is easy, if you know what to take into account. There are three main issues:

1. the location of the zero page registers
2. the base address of the code in memory
3. the save and restore routines

Let's start with the zero page registers

Sweet 16 needs 32 zero page addresses for the 16 registers (R0-R15). In the original Apple Version this registers are from \$0 to \$1F, in the ATARI Version the locations \$E0-\$FF are used. You can place this registers anywhere in the zeropage, but it have to be a contiguous range.

The base address of the code

Sweet 16 uses a tricky indirect 8-Bit jump to access the routines for each opcode. This saves some space in the code. But for this hack to work, some precaution have to be taken for the code. All code from the label "SET" to the label "RTN" have to be assembled in one 6502 Page (\$100/256 Bytes). If you mind this, everything should work fine.

The "save" and "restore" routines

The Apple II Version of Sweet 16 uses the ROM-internal "save" and "restore" routines to save and restore all processor registers and flags. If your system has such routines in ROM, fine, just use them. If not, you have to add these routines to your programm. Here is a template:

```
;-----  
SAVE  
    STA ACC  
    STX XREG  
    STY YREG  
    PHP  
    PLA  
    STA STATUS  
    CLD  
    RTS  
;-----  
RESTORE  
    LDA STATUS  
    PHA  
    LDA ACC  
    LDX XREG
```

```
LDY YREG
PLP
RTS
;-----
ACC .BYTE 0
XREG .BYTE 0
YREG .BYTE 0
STATUS .BYTE 0
```

Thats all. Porting Sweet 16 is easy and straightforward. I hope to see some Sweet 16 enhanced 6502 source in the future. If someone is searching for a nice project, extending an open-source 6502 Assembler (like CA65 from CC65) with the Sweet 16 opcodes would be one :)

The Story of Sweet 16 (as told by Woz)

While writing Apple BASIC, I ran into the problem of manipulating the 16 bit pointer data and its arithmetic in an 8 bit machine.

My solution to this problem of handling 16 bit data, notably pointers, with an 8 bit microprocessor was to implement a non-existent 16 bit processor in software, interpreter fashion, which I refer to as SWEET16. SWEET16 contains sixteen internal 16 bit registers, actually the first 32 bytes in main memory, labelled R0 through R15. R0 is defined as the accumulator, R15 as the program counter, and R14 as a status register. R13 stores the result of all COMPARE operations for branch testing. The user acceses SWEET16 with a subroutine call to hexadecimal address F689. Bytes stored after the subroutine call are thereafter interpreted and executed by SWEET16. One of SWEET16's commands returns the user back to 6502 mode, even restoring the original register contents.

Implemented in only 300 bytes of code, SWEET16 has a very simple instruction set tailored to operations such as memory moves and stack manipulation. Most opcodes are only one byte long, but since she runs approximately ten times slower than equivalent 6502 code, SWEET16 should be employed only when code is at a premium or execution is not. As an example of her usefulness, I have estimated that about 1K byte could be weeded out of my 5K byte Apple-II BASIC interpreter with no observable performance degradation by selectively applying SWEET16. [Full article in BYTE magazine [here](#).]

Source code (Sweet 16)

The following is from a 1994 post to Usenet group comp.sys.apple2.programmer by Sheldon Simms.

```
Article 1684 of comp.sys.apple2.programmer:
Newsgroups: comp.sys.apple2.programmer
Path: amdahl.uts.amdahl.com!amdahl!amd!decwrl!sdd.hp.com!elroy.jpl.nasa.gov!swrinde!ihnp4.ucsd.edu!library.ucla.edu!csulb.edu!csus.edu!netcom.com!sheldon
From: sheldon@netcom.com (Sheldon Simms)
Subject: Source code (Sweet 16)
Message-ID: <sheldonCLz3qM.85r@netcom.com>
Organization: NETCOM On-line Communication Services (408 241-9760 guest)
Date: Tue, 1 Mar 1994 06:37:33 GMT
Lines: 264
```

Well maybe this should go to comp.sources.apple2 but it's not too long and in my mind it's part of the thread on learning assembly language programming. It's an opportunity to learn from Woz himself.

-Sheldon

```
*****
*
* APPLE-II PSEUDO MACHINE
* INTERPRETER
*
* COPYRIGHT (C) 1977
* APPLE COMPUTER, INC
*
* ALL RIGHTS RESERVED
*
* S. WOZNIAK
*
*****
*
* TITLE: SWEET 16 INTERPRETER
*
*****
```

```
R0L EQU $0
R0H EQU $1
R14H EQU $1D
R15L EQU $1E
R15H EQU $1F
SAVE EQU $FF4A
RESTORE EQU $FF3F

ORG $F689

AST 32

JSR SAVE ;PRESERVE 6502 REG CONTENTS
PLA
STA R15L ;INIT SWEET16 PC
PLA ;FROM RETURN
STA R15H ;ADDRESS
SW16B JSR SW16C ;INTERPRET AND EXECUTE
JMP SW16B ;ONE SWEET16 INSTR.
SW16C INC R15L
BNE SW16D ;INCR SWEET16 PC FOR FETCH
INC R15H
SW16D LDA >SET ;COMMON HIGH BYTE FOR ALL ROUTINES
```

```

        PHA                ;PUSH ON STACK FOR RTS
        LDY $0
        LDA (R15L),Y       ;FETCH INSTR
        AND $F             ;MASK REG SPECIFICATION
        ASL                ;DOUBLE FOR TWO BYTE REGISTERS
        TAX                ;TO X REG FOR INDEXING
        LSR
        EOR (R15L),Y       ;NOW HAVE OPCODE
        BEQ TOBR           ;IF ZERO THEN NON-REG OP
        STX R14H           ;INDICATE "PRIOR RESULT REG"
        LSR
        LSR                ;OPCODE*2 TO LSB'S
        LSR
        TAY                ;TO Y REG FOR INDEXING
        LDA OPTBL-2,Y      ;LOW ORDER ADR BYTE
        PHA                ;ONTO STACK
        RTS                ;GOTO REG-OP ROUTINE
TOBR    INC R15L
        BNE TOBR2          ;INCR PC
        INC R15H
TOBR2   LDA BRTBL,X        ;LOW ORDER ADR BYTE
        PHA                ;ONTO STACK FOR NON-REG OP
        LDA R14H           ;"PRIOR RESULT REG" INDEX
        LSR                ;PREPARE CARRY FOR BC, BNC.
        RTS                ;GOTO NON-REG OP ROUTINE
RTNZ    PLA                ;POP RETURN ADDRESS
        PLA
        JSR RESTORE        ;RESTORE 6502 REG CONTENTS
        JMP (R15L)         ;RETURN TO 6502 CODE VIA PC
SETZ    LDA (R15L),Y       ;HIGH ORDER BYTE OF CONSTANT
        STA R0H,X
        DEY
        LDA (R15L),Y       ;LOW ORDER BYTE OF CONSTANT
        STA R0L,X
        TYA                ;Y REG CONTAINS 1
        SEC
        ADC R15L           ;ADD 2 TO PC
        STA R15L
        BCC SET2
        INC R15H
SET2    RTS
OPTBL   DFB SET-1          ;1X
BRTBL   DFB RTN-1          ;0
        DFB LD-1           ;2X
        DFB BR-1           ;1
        DFB ST-1           ;3X
        DFB BNC-1          ;2
        DFB LDAT-1         ;4X
        DFB BC-1           ;3
        DFB STAT-1         ;5X
        DFB BP-1           ;4
        DFB LDDAT-1        ;6X
        DFB BM-1           ;5
        DFB STDAT-1        ;7X
        DFB BZ-1           ;6
        DFB POP-1          ;8X
        DFB BNZ-1          ;7
        DFB STPAT-1        ;9X
        DFB BM1-1          ;8
        DFB ADD-1          ;AX
        DFB BNM1-1         ;9
        DFB SUB-1          ;BX
        DFB BK-1           ;A
        DFB POPD-1         ;CX
        DFB RS-1           ;B
        DFB CPR-1          ;DX
        DFB BS-1           ;C
        DFB INR-1          ;EX
        DFB NUL-1          ;D
        DFB DCR-1          ;FX
        DFB NUL-1          ;E
        DFB NUL-1          ;UNUSED
        DFB NUL-1          ;F
```

* FOLLOWING CODE MUST BE
* CONTAINED ON A SINGLE PAGE!

```

SET     BPL SETZ           ;ALWAYS TAKEN
LD      LDA R0L,X
BK      EQU *-1
        STA R0L
        LDA R0H,X          ;MOVE RX TO R0
        STA R0H
        RTS
ST      LDA R0L
        STA R0L,X          ;MOVE R0 TO RX
        LDA R0H
        STA R0H,X
        RTS
STAT    LDA R0L
STAT2   STA (R0L,X)        ;STORE BYTE INDIRECT
        LDY $0
STAT3   STY R14H           ;INDICATE R0 IS RESULT NEG
INR     INC R0L,X
        BNE INR2           ;INCR RX
        INC R0H,X
INR2    RTS
LDAT    LDA (R0L,X)        ;LOAD INDIRECT (RX)
```

	STA	R0L	;TO R0
	LDY	\$0	
	STY	R0H	;ZERO HIGH ORDER R0 BYTE
	BEQ	STAT3	;ALWAYS TAKEN
POP	LDY	\$0	;HIGH ORDER BYTE = 0
	BEQ	POP2	;ALWAYS TAKEN
POPD	JSR	DCR	;DECR RX
	LDA	(R0L,X)	;POP HIGH ORDER BYTE @RX
	TAY		;SAVE IN Y REG
POP2	JSR	DCR	;DECR RX
	LDA	(R0L,X)	;LOW ORDER BYTE
	STA	R0L	;TO R0
	STY	R0H	
POP3	LDY	\$0	;INDICATE R0 AS LAST RESULT REG
	STY	R14H	
	RTS		
LDDAT	JSR	LDAT	;LOW ORDER BYTE TO R0, INCR RX
	LDA	(R0L,X)	;HIGH ORDER BYTE TO R0
	STA	R0H	
	JMP	INR	;INCR RX
STDAT	JSR	STAT	;STORE INDIRECT LOW ORDER
	LDA	R0H	;BYTE AND INCR RX. THEN
	STA	(R0L,X)	;STORE HIGH ORDER BYTE.
	JMP	INR	;INCR RX AND RETURN
STPAT	JSR	DCR	;DECR RX
	LDA	R0L	
	STA	(R0L,X)	;STORE R0 LOW BYTE @RX
	JMP	POP3	;INDICATE R0 AS LAST RESULT REG
DCR	LDA	R0L,X	
	BNE	DCR2	;DECR RX
	DEC	R0H,X	
DCR2	DEC	R0L,X	
	RTS		
SUB	LDY	\$0	;RESULT TO R0
	CPR	SEC	;NOTE Y REG = 13*2 FOR CPR
	LDA	R0L	
	SBC	R0L,X	
	STA	R0L,Y	;R0-RX TO RY
	LDA	R0H	
	SBC	R0H,X	
SUB2	STA	R0H,Y	
	TYA		;LAST RESULT REG*2
	ADC	\$0	;CARRY TO LSB
	STA	R14H	
	RTS		
ADD	LDA	R0L	
	ADC	R0L,X	
	STA	R0L	;R0+RX TO R0
	LDA	R0H	
	ADC	R0H,X	
	LDY	\$0	;R0 FOR RESULT
	BEQ	SUB2	;FINISH ADD
BS	LDA	R15L	;NOTE X REG IS 12*2!
	JSR	STAT2	;PUSH LOW PC BYTE VIA R12
	LDA	R15H	
	JSR	STAT2	;PUSH HIGH ORDER PC BYTE
BR	CLC		
BNC	BCS	BNC2	;NO CARRY TEST
BR1	LDA	(R15L),Y	;DISPLACEMENT BYTE
	BPL	BR2	
	DEY		
BR2	ADC	R15L	;ADD TO PC
	STA	R15L	
	TYA		
	ADC	R15H	
	STA	R15H	
BNC2	RTS		
BC	BCS	BR	
	RTS		
BP	ASL		;DOUBLE RESULT-REG INDEX
	TAX		;TO X REG FOR INDEXING
	LDA	R0H,X	;TEST FOR PLUS
	BPL	BR1	;BRANCH IF SO
	RTS		
BM	ASL		;DOUBLE RESULT-REG INDEX
	TAX		
	LDA	R0H,X	;TEST FOR MINUS
	BMI	BR1	
	RTS		
BZ	ASL		;DOUBLE RESULT-REG INDEX
	TAX		
	LDA	R0L,X	;TEST FOR ZERO
	ORA	R0H,X	; (BOTH BYTES)
	BEQ	BR1	;BRANCH IF SO
	RTS		
BNZ	ASL		;DOUBLE RESULT-REG INDEX
	TAX		
	LDA	R0L,X	;TEST FOR NON-ZERO
	ORA	R0H,X	; (BOTH BYTES)
	BNE	BR1	;BRANCH IF SO
	RTS		
BM1	ASL		;DOUBLE RESULT-REG INDEX
	TAX		
	LDA	R0L,X	;CHECK BOTH BYTES
	AND	R0H,X	;FOR \$FF (MINUS 1)
	EOR	\$FF	
	BEQ	BR1	;BRANCH IF SO
	RTS		

```
BNM1    ASL                ;DOUBLE RESULT-REG INDEX
        TAX
        LDA    R0L,X
        AND    R0H,X      ;CHECK BOTH BYTES FOR NO $FF
        EOR    $FF
        BNE    BR1        ;BRANCH IF NOT MINUS 1
NUL      RTS
RS       LDX    $18        ;12*2 FOR R12 AS STACK POINTER
        JSR    DCR        ;DECR STACK POINTER
        LDA    (R0L,X)    ;POP HIGH RETURN ADDRESS TO PC
        STA    R15H
        JSR    DCR        ;SAME FOR LOW ORDER BYTE
        LDA    (R0L,X)
        STA    R15L
        RTS
RTN      JMP    RTNZ
--

W. Sheldon Simms      | Newt's Friend / Jack Kemp for President
sheldon@netcom.com    | Freedom implies responsibility
-----+-----
```

```
Article 1685 of comp.sys.apple2.programmer:
Newsgroups: comp.sys.apple2.programmer
Path: amdahl.uts.amdahl.com!amdahl!amd!decwrl!sdd.hp.com!elroy.jpl.nasa.gov!swrinde!ihnp4.ucsd.edu!library.ucla.edu!csulb.edu!csus.edu!netcom.com!sheldon
From: sheldon@netcom.com (Sheldon Simms)
Subject: Sweet 16 description
Message-ID: <sheldonCLz3tB.8C4@netcom.com>
Organization: NETCOM On-line Communication Services (408 241-9760 guest)
Date: Tue, 1 Mar 1994 06:39:11 GMT
Lines: 992

Well since I posted the source, here's what it does for anyone who
might not know...

-Sheldon
```

Sweet 16 - Introduction

by Dick Sedgewick (from Merlin Users Manual)

Sweet 16 is probably the least used and least understood seed in the Apple][.

In exactly the same sense that Integer and Applesoft Basics are languages, SWEET 16 is a language. Compared to the Basics, however, it would be classed as low level with a strong likeness to conventional 6502 Assembly language.

To use SWEET 16, you must learn the language - and to quote "WOZ", "The opcode list is short and uncomplicated". "WOZ" (Steve Wozniak), of course is Mr. Apple, and the creator of SWEET 16.

SWEET 16 is ROM based in every Apple][from \$F689 to \$F7FC. It has it's own set of opcodes and instruction sets, and uses the SAVE and RESTORE routines from the Apple Monitor to preserve the 6502 registers when in use, allowing SWEET 16 to be used as a subroutine.

It uses the first 32 locations on zero page to set up its 16 double byte registers, and is therefore not compatible with Applesoft Basic without some additional efforts.

The original article, "SWEET 16: The 6502 Dream Machine", first appeared in Byte Magazine, November 1977 and later in the original "WOZ PAK". The article is included here and again as test material to help understand the use and implementation of SWEET 16.

Examples of the use of SWEET 16 are found in the Programmer's Aid #1, in the Renumber, Append, and Relocate programs. The Programmer's Aid Operating Manual contains complete source assembly listings, indexed on page 65.

The demonstration program is written to be introductory and simple, consisting of three parts:

- 1. Integer Basic Program
- 2. Machine Language Subroutine
- 3. SWEET 16 Subroutine

The task of the program will be to move data. Parameters of the move will be entered in the Integer Basic Program.

The "CALL 768" (\$300) at line 120, enters a 6502 machine language subroutine having the single purpose of entering SWEET 16 and subsequently returning to BASIC (addresses \$300, \$301, \$302, and \$312 respectively). The SWEET 16 subroutine of course performs the move, and is entered at Hex locations \$303 to \$311 (see listing Number 3).

After the move, the screen will display three lines of data, each 8 bytes long, and await entry of a new set of parameters. The three lines of data displayed on the screen are as follows:

- Line 1: The first 8 bytes of data starting at \$800, which is the fixed source data to be moved (in this case, the string A\$).
- Line 2: The first 8 bytes of data starting at the hex address entered as the destination of the move (high order byte only).
- Line 3: The first 8 bytes of data starting at \$0000 (the first four SWEET 16 registers).

The display of 8 bytes of data was chosen to simplify the illustration of what goes on.

Integer Basic has its own way of recording the string A\$. Because the name chosen for the string "A\$" is stored in 2 bytes, a total of five housekeeping bytes precede the data entered as A\$, leaving only three additional bytes available for display. Integer Basic also adds a

housekeeping byte at the end of a string, known as the "string terminator".

Consequently, for convenience purposes of the display, and to see the string terminator as the 8th byte, the string data entered via the keyboard should be limited to two characters, and will appear as the 6th and 7th bytes. Additionally, parameters to be entered include the number of bytes to be moved. A useful range for this demonstration would be 1-8 inclusive, but of course 1-255 will work.

Finally, the starting address of the destination of the move must be entered. Again, for simplicity, only the high-order byte is entered, and the program allows a choice between Decimal 9 and high-order byte of program pointer 1, to avoid unnecessary problems (in this demonstration enter a decimal number between 9 and 144 for a 48K APPLE).

The 8 bytes of data displayed starting at \$00 will enable one to observe the condition of the SWEET 16 registers after a move has been accomplished, and thereby understand how the SWEET 16 program works.

From the article "SWEET 16: A 6502 Dream Machine", remember that SWEET 16 can establish 16 double byte registers starting at \$00. This means that SWEET 16 can use the first 32 addresses on zero page.

The "events" occurring in this demonstration program can be studied in the first four SWEET 16 registers. Therefore, the 8 byte display starting at \$0000 is large enough for this purpose.

These four registers are established as R0, R1, R2, R3:

R0	\$0000	&	0001	-SWEET 16 accumulator
R1	\$0002	&	0003	-Source address
R2	\$0004	&	0005	-Destination address
R3	\$0006	&	0007	-Number of bytes to move
.				
.				
R14	\$001C	&	001D	-Prior result register
R15	\$001E	&	001F	-SWEET 16 Program counter

Additionally, an examination of registers R14 and R15 will extend our understanding of SWEET 16, as fully explained in the "WOZ" text. Notice that the high order byte of R14, (located at \$1D) contains \$06, and is the doubled register specification (3X2=\$06). R15, the SWEET 16 program counter contains the address of the next operation as it did for each step during execution of the program, which was \$0312 when execution ended and the 6502 code resumed.

To try a sample run, enter the Integer Basic program as shown in Listing #1. Of course, REM statements can be omitted, and line 10 is only helpful if the machine code is to be stored on disk. Listing #2 must also be entered starting at \$300.

NOTE: A 6502 disassembly does not look like listing #3, but the SOURCEROR disassembler would create a correct disassembly.

```
Enter "RUN" and hit RETURN
Enter "12" and hit RETURN (A$ - A$ string data)
Enter "18" and hit RETURN (high-order byte of destination)
```

The display should appear as follows:

```
$0800-C1 40 00 10 08 B1 B2 1E (SOURCE)
$0A00-C1 40 00 10 08 B1 B2 1E (Dest.)
$0000-1E 00 08 08 08 0A 00 00 (SWEET 16)
```

NOTE: The 8 bytes stored at \$0A00 are identical to the 8 bytes starting at \$0800, indicating that an accurate move of 8 bytes length has been made. They are moved one byte at a time starting with token C1 and ending with token 1E. If moving less than 8 bytes, the data following the moved data would be whatever existed at those locations before the move.

The bytes have the following significance:

A Token\$

C1	40	00	10	08	B1	B2	1E
-----	-----	-----	-----	-----	-----	-----	---
						String	
VN	DSP	NVA	DATA	DATA	Terminator		

The SWEET 16 registers are as shown:

low	high	low	high	low	high	low	high
\$0000	1E 00	08 08	08 08	08 0A	0A 00	00 00	00 00
-----	-----	-----	-----	-----	-----	-----	-----
register	register	register	register	register	register	register	register
R0	R1	R2	R3				
(acc)	(source)	(dest)	(#bytes)				

The low order byte of R0, the SWEET 16 accumulator, has \$1E in it, the last byte moved (the 8th).

The low order byte of the source register R1 started as \$00 and was incremented eight times, once for each byte of moved data.

The high order byte of the destination register R2 contains \$0A, which was entered at 10 (the variable) and poked into the SWEET 16 code. The low-order byte of R2 was incremented exactly like R1.

Finally, register R3, the register that stores the number of bytes to be moved, has been poked to 8 (the variable B) and decremented eight times as each byte got moved, ending up \$0000.

By entering character strings and varying the number of bytes to be moved, the SWEET 16 registers can be observed and the contents predicted.

Working with this demonstration program, and study of the text material will enable you to write SWEET 16 programs that perform additional 16 bit manipulations. The unassigned opcodes mentioned in the "WOZ Dream Machine" article should present a most interesting opportunity to "play".

SWEET 16 as a language - or tool - opens a new direction to Apple][owners without spending a dime, and it's been there all the time.

"Apple-ites" who desire to learn machine language programming, can use SWEET 16 as a starting point. With this text material to use, and less opcodes to learn, a user can quickly be effective.

Listing #1

```
>List
10 PRINT "[D]BLOAD SWEET": REM CTRL D
20 CALL - 936: DIM A $ (10)
30 INPUT "ENTER STRING A $ " , A $
40 INPUT "ENTER # BYTES " , B
50 IF NOT B THEN 40 : REM AT LEAST 1
60 POKE 778 , B : REM POKE LENGTH
70 INPUT "ENTER DESTINATION " , A
80 IF A > PEEK (203) - 1 THEN 70
90 IF A < PEEK (205) + 1 THEN 70
100 POKE 776 , A : REM POKE DESTINATION
110 M = 8 : GOSUB 160 : REM DISPLAY
120 CALL 768 : REM GOTO $0300
130 M = A : GOSUB 160 : REM DISPLAY
140 M = 0 : GOSUB 160 : REM DISPLAY
150 PRINT : PRINT : GOTO 30
160 POKE 60 , 0 : POKE 61 , M
170 CALL -605 : RETURN : REM XAM8 IN MONITOR
```

Listing #2

```
300:20 89 F6 11 00 08 12 00 00 13 00 00 41 52
F3 07 FB 00 60
```

Listing #3

SWEET 16

\$300	20	89	F6	JSR	\$F689
\$303	11	00	08	SET	R1 source address
\$306	12	00	00	SET	R2 destination address
				A	
\$309	13	00	00	SET	R3 length
				B	
\$30C	41			LD	@R1
\$30D	52			ST	@R2
\$30E	F3			DCR	R3
\$30F	07			BNZ	\$30C
\$311	00			RTN	
\$312	60			RTS	

Data will be poked from the Integer Basic program:

"A"	from Line 100
"B"	from Line 60

SWEET 16: A Pseudo 16 Bit Microprocessor

by Steve Wozniak

Description:

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 6502/RCA 1800 hybrid - an abundance of 16-bit registers and excellent pointer capability.

My solution was to implement a non-existent (meta) 16-bit processor in software, interpreter style, which I call SWEET 16.

SWEET 16 is based on sixteen 16-bit registers (R0-15), which are actually 32 memory locations. R0 doubles as the SWEET 16 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 SWEET 16 subroutines are used. All other SWEET 16 registers are at the user's unrestricted disposal.

SWEET 16 instructions fall into register and non-register categories. The register ops specify one of the sixteen registers to be used as either a data element or a pointer to data in memory, depending on the specific instruction. For example INR R5 uses R5 as data and ST @R7 uses R7 as a pointer to data in memory. Except for the SET instruction, register ops take one byte of code each. The non-register ops are primarily 6502 style branches with the second byte specifying a +/-127 byte displacement relative to the address of the following instruction. Providing that the prior register op result meets a specified branch condition, the displacement is added to the SWEET 16 PC, effecting a branch.

SWEET 16 is intended as a 6502 enhancement package, not a stand alone processor. A 6502 program switches to SWEET 16 mode with a subroutine call and subsequent code is interpreted as SWEET 16 instructions. The nonregister op RTN returns the user program to 6502 mode after restoring the internal register contents (A, X, Y, P, and S). The following example illustrates how to use SWEET 16.

```
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 89 F6       JSR  SW16       ;Yes, call SWEET 16
30A 41            MLOOP LD  @R1     ;R1 holds source
30B 52            ST   @R2         ;R2 holds dest. addr.
30C F3            DCR  R3          ;Decr. length
30D 07 FB          BNZ  MLOOP      ;Loop until done
30F 00            RTN             ;Return to 6502 mode.
310 C9 C5          NOMOVE CMP  #"E" ;"E" char?
312 D0 13          BEQ  EXIT       ;Yes, exit
314 C8            INY             ;No, cont.
```

NOTE: Registers A, X, Y, P, and S are not disturbed by SWEET 16.

Instruction Descriptions:

The SWEET 16 opcode listing is short and uncomplicated. Excepting relative branch displacements, hand assembly is trivial. All register opcodes are formed by combining two Hex digits, one for the opcode and one to specify a register. For example, opcodes 15 and 45 both specify register R5 while codes 23, 27, and 29 are all ST ops. Most register ops are assigned in complementary pairs to facilitate remembering them. Therefore, LD ans ST are opcodes 2N and 3N respectively, while LD @ and ST @ are codes 4N and 5N.

Opcodes 0 to C (Hex) are assigned to the thirteen non-register ops. Except for RTN (opcode 0), BK (0A), and RS (0B), the non register ops are 6502 style branches. The second byte of a branch instruction contains a +/-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 op result, the displacement is added to the PC effecting a branch. Except for the BR (Branch always) and BS (Branch to a Subroutine), the branch opcodes are assigned in complementary pairs, rendering them easily remembered for hand coding. For example, Branch if Plus and Branch if Minus are opcodes 4 and 5 while Branch if Zero and Branch if [NonZero?](#) are opcodes 6 and 7.

SWEET 16 Opcode 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)

Non-register 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		(Break)
0B	RS		(Return from Subroutine)
0C	BS	ea	(Branch to Subroutine)
0D			(Unassigned)
0E			(Unassigned)
0F			(Unassigned)

Register Instructions:

SET:

SET Rn,Constant [1n Low High]

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

EXAMPLE:

```
15 34 A0  SET  R5  $A034      ;R5 now contains $A034
```

LOAD:

LD Rn [2n]

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

EXAMPLE:

```
15 34 A0 SET R5 $A034
25      LD R5      ;ACC now contains $A034
```

STORE:

ST Rn [3n]

The ACC 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
```

LOAD INDIRECT:

LD @Rn [4n]

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
45      LD @R5     ;ACC is loaded from memory
                     ;location $A034
                     ;R5 is incr to $A035
```

STORE INDIRECT:

ST @Rn [5n]

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, R6 with
16 22 90 SET R6 $9022 ;$A034 and $9022
45      LD @R5     ;Move byte from $A034 to $9022
56      ST @R6     ;Both ptrs are incremented
```

LOAD DOUBLE-BYTE INDIRECT:

LDD @Rn [6n]

The low order ACC byte is loaded from 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 loaded
65      LDD @R6     ;from $A034, high-order from
                     ;$A035, R5 is incr to $A036
```

STORE DOUBLE-BYTE INDIRECT:

STD @Rn [7n]

The low-order ACC byte is stored into memory location whose address resides in Rn, and Rn is the 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, R6
16 22 90 SET R6 $9022 ;with $A034 and $9022
65      LDD @R5     ;Move double byte from
76      STD @R6     ;$A034-35 to $9022-23.
                     ;Both pointers incremented by 2.
```

POP INDIRECT:

POP @Rn [8n]

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 one. 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 ops (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
55 ST @R5 ;Push 4 onto stack
10 05 00 SET R0 5 ;Load 5 into ACC
55 ST @R5 ;Push 5 onto stack
10 06 00 SET R0 6 ;Load 6 into ACC
55 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

```

STORE POP INDIRECT:

STP @Rn [9n]

The low-order ACC byte is stored into the memory location whose address resides in Rn after Rn is decremented by 1. Branch conditions will reflect the 2-byte ACC contents which are not modified. STP @Rn and POP @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 ops.

EXAMPLE:

```

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

```

ADD:

ADD Rn [An]

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

EXAMPLE:

```

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

```

SUBTRACT:

SUB Rn [Bn]

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

$ACC = ACC + Rn + 1$

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
B1 SUB R1 ;subtract R1
;(diff=$340D with c set)
B0 SUB R0 ;clears ACC. (R0)

```

POP DOUBLE-BYTE INDIRECT:

POPD @Rn [Cn]

Rn is decremented by 1 and the high-order ACC byte is loaded from the memory location whose address now resides in Rn. 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 ops (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
C5 POPD @R5 ;Pop $BB34 off stack
C5 POPD @R5 ;Pop $AA12 off stack

```

COMPARE:

CPR Rn [Dn]

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
B0           LOOP1 SUB  R0          ;Zero data
75           STD   @R5            ;clear 2 locations
                        ;increment R5 by 2
25           LD    R5            ;Compare pointer R5
D6           CPR   R6            ;to limit R6
02 FA        BNC   LOOP1         ;loop if C clear
```

INCREMENT:**INR Rn [En]**

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      ;(Pointer)
B0           SUB  R0          ;Zero to R0
55           ST   @R5          ;Clr Location $A034
E5           INR   R5          ;Incr R5 to $A036
55           ST   @R5          ;Clrs location $A036
                        ;(not $A035)
```

DECREMENT:**DCR Rn [Fn]**

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

EXAMPLE: (Clear 9 bytes beginning at location A034)

```
15 34 A0      SET  R5  $A034      ;Init pointer
14 09 00      SET  R4   9         ;Init counter
B0           SUB  R0          ;Zero ACC
55           LOOP2 ST   @R5          ;Clear a mem byte
F4           DCR  R4          ;Decrement count
07 FC        BNZ  LOOP2         ;Loop until Zero
```

Non-Register Instructions:

RETURN TO 6502 MODE:**RTN 00**

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 to entering SWEET 16 mode).

BRANCH ALWAYS:**BR ea [01 d]**

An effective address (ea) is calculated by adding the signed displacement byte (d) to the PC. The PC contains the address of the instruction immediately following the BR, or the address of the BR op plus 2. The displacement is a signed two's complement value from -128 to +127. Branch conditions are not changed.

NOTE: The effective address calculation is identical to that for 6502 relative branches. The Hex add & Subtract features of the APPLE][monitor may be used to calculate displacements.

```
d = $80  ea = PC + 2 - 128
d = $81  ea = PC + 2 - 127
```

```
d = $FF  ea = PC + 2 - 1
d = $00  ea = PC + 2 + 0
d = $01  ea = PC + 2 + 1
```

```
d = $7E  ea = PC + 2 + 126
d = $7F  ea = PC + 2 + 127
```

EXAMPLE:

```
$300: 01 50 BR $352
```

BRANCH IF NO CARRY:**BNC ea [02 d]**

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.

BRANCH IF CARRY SET:**BC ea [03 d]**

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

BRANCH IF PLUS:

BP ea [04 d]

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

EXAMPLE: (Clear mem from A034 to A03F)

15 34 A0	SET	R5	\$A034	;Init pointer
14 3F A0	SET	R4	\$A03F	;Init limit
B0		SUB	R0	
55	LOOP3	ST	@R5	;Clear mem byte
				;Increment R5
24		LD	R4	;Compare limit
D5		CPR	R5	;to pointer
04 FA		BP	LOOP3	;Loop until done

BRANCH IF MINUS:

BM ea [05 d]

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

BRANCH IF ZERO:

BZ ea [06 d]

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

BRANCH IF NONZERO

BNZ ea [07 d]

A branch is effected only if the priot 'result' was non-zero Branch conditions are not changed.

BRANCH IF MINUS ONE

BM1 ea [08 d]

A branch is effected only if the prior 'result' was minus one (\$FFFF Hex). Branch conditions are not changed.

BRANCH IF NOT MINUS ONE

BNM1 ea [09 d]

A branch effected only if the prior 'result' was not minus 1. Branch conditions are not changed.

BREAK:

BK [0A]

A 6502 BRK (break) instruction is executed. SWEET 16 may be re-entered non destructively at SW16d after correcting the stack pointer to its value prior to executing the BRK.

RETURN FROM SWEET 16 SUBROUTINE:

RS [0B]

RS terminates execution of a SWEET 16 subroutine and returns to the SWEET 16 calling program which resumes execution (in SWEET 16 mode). R12, which is the SWEET 16 subroutine return stack pointer, is decremented twice. Branch conditions are not changed.

BRANCH TO SWEET 16 SUBROUTINE:

BS ea [0c d]

A branch to the effective address (PC + 2 + d) is taken and execution is resumed in SWEET 16 mode. The current PC is pushed onto a SWEET 16 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)

15 34 A0	SET	R5	\$A034	;Init pointer 1
14 3B A0	SET	R4	\$A03B	;Init limit 1
16 00 30	SET	R6	\$3000	;Init pointer 2
0C 15		BS	MOVE	;Call move subroutine
45		LD	@R5	;Move one
56	MOVE	ST	@R6	;byte
24		LD	R4	
D5		CPR	R5	;Test if done
04 FA		BP	MOVE	
0B		RS		;Return

Theory of Operation:

SWEET 16 execution mode begins with a subroutine call to SW16. All 6502 registers are saved at this time, to be restored when a SWEET 16 RTN instruction returns control to the 6502. If you can tolerate indefinite 6502 register contents upon exit, approximately 30 usec may be saved by entering at SW16 + 3. Because this might cause an inadvertant switch from Hex to Decimal mode, it is advisable to enter at SW16 the first time through.

After saving the 6502 registers, SWEET 16 initializes its PC (R15) with the subroutine return address off the 6502 stack. SWEET 16's PC points to the location preceding the next instruction to be executed. Following the subroutine call are 1-,2-, and 3-byte SWEET 16 instructions, stored in ascending memory like 6502 instructions. the main loop at SW16B repeatedly calls the 'execute instruction' routine to execute it.

Subroutine SW16C increments the PC (R15) and fetches the next opcode, which is either a register op of the form OP REG with OP between 1 and 15 or a non-register op of the form 0 OP with OP between 0 and 13. Assuming a register op, the register specification is doubled to account for the 3 byte SWEET 16 registers and placed in the X-reg for indexing. Then the instruction type is determined. Register ops place the doubled register specification in the high order byte of R14 indicating the 'prior result register' to subsequent branch instructions. Non-register ops treat the register specifcation (right-hand half-byte) as their opcode, increment the SWEET 16 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 into by the opcode. By assigning all the entries to a common page, only a single byte to address need be stored per routine. The 6502 indirect jump might have been used as follows to transfer control to the appropriate subroutine.

```
LDA    #ADRH      ;High-order 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 PC (after incrementing by 1). The net result is that the desired subroutine is reached by executing a subroutine return instruction!

Opcode Subroutines:

The register op routines make use of the 6502 'zero page indexed by X' and 'indexed by X direct' addressing modes to access the specified registers and indirect data. The 'result' of most register ops is left 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 SWEET 16 registers and the LSB is zero. If ADD, SUB, or CPR instructions generate carries, then this index is incremented, setting the LSB.

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

Most SWEET 16 non-register ops are relative branches. The corresponding subroutines determine whether or not the 'prior result' meets the specified branch condition and if so, update the SWEET 16 PC by adding the displacement value (-128 to +127 bytes).

The RTN op restores the 6502 register contents, pops the subroutine return stack and jumps indirect through the SWEET 16 PC. This transfers control to the 6502 at the instruction immediately following the RTN instruction.

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

Any number of subroutine levels may be implemented within SWEET 16 code via the BS (Branch to Subroutine) and RS (Return from Subroutine) instructions. The user must initialize and otherwise not disturb R12 if the SWEET 16 subroutine capability is used since it is utilized as the automatic return stack pointer.

Memory Allocation:

The only storage that must be allocated for SWEET 16 variables are 32 consecutive locations in page zero for the SWEET 16 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 ASAV, XSAV, YSAV, and PSAV.

User Modifications:

You may wish to add some of your own instructions to this implementation of SWEET 16. If you use the unassigned opcodes \$0E and \$0F, remember that SWEET 16 treats these as 2-byte instructions. You may wish to handle the break instruction as a SWEET 16 call, saving two bytes of code each time you transfer into SWEET 16 mode. Or you may wish to use the SWEET 16 BK (break) op as a 'CHAROUT' call in the interrupt handler. You can perform absolute jumps within SWEET 16 by loading the ACC (R0) with the address you wish to jump to (minus 1) and executing a ST R15 instruction.

Notes from Apple II System Description

This is kind of old stuff, but I ran across the issue of Byte that had the Apple II system description by Steve Wozniak:

```
00  1  Return to 6502 mode
01  2  Branch Always
02  2  Branch no Carry
03  2  Branch on Carry
```

```
04 2 Branch on Positive
05 2 Branch on Negative
06 2 Branch if equal
07 2 Branch not equal
08 2 Branch on negative 1
09 2 Branch not negative 1
0A 1 Break to Monitor
0B-0F 1 No operation
1R 3 R<-2 byte constant (load register immediate)
2R 1 ACC<-R
3R 1 ACC->R
4R 1 ACC<-@R, R<-R+1
5R 1 ACC->@R, R<-R+1
6R 1 ACC<-@R double
7R 1 ACC->@R double
8R 1 R<-R-1, ACC<-@R (pop)
9R 1 R<-R-1, ACC->@R
AR 1 ACC<-@R(pop) double
BR 1 compare ACC to R
CR 1 ACC<-ACC+R
DR 1 ACC<-ACC-R
ER 1 R<-R+1
FR 1 R<-R-1
```

notes

- 1. All braches are followed by a 1 byte relative displacement. Works identically to 6502 branches.
- 2. Only ADD,SUB, and COMPARE can set carry
- 3. Notation:
 - o R = a 16 bit "Register" operand designation, one of 16 labelled 0 to 15 (decimal), 0 to F (hexidecimal)
 - o ACC = register operand R0
 - o @R = indicrect reference, using the register R as the pointer
 - o <-, -> = assignment of values
- 4. Length of instructions: Branches are always two bytes: opcodes followed by relative displacement. Load register immediate (1R) is three bytes: the hexadecimal opcode 10 to 1F followed by the 2 byte literal value of a 16 bit number. All other instructions are one byte in length.

And from that issue of Byte (Apr 1977, I think)...some words from the Woz himself. Retyped without permission.

Sweet 16 S-C Macro Assembler Text

SWEET-16 is a powerful programming tool developed by Steve Wozniak in the early days of Apple. Chances are that you do have this tool, whether you know it or not. The standard version is hidden away inside the Integer BASIC system. If you have Integer BASIC on your mother board, or in a firmware card, or in a 16K RAM card, then you have SWEET-16. I have included a commented source file of SWEET-16 on your S-C MACRO ASSEMBLER II disk, so you can assemble your own copy if you wish.

SWEET-16 is really a language, just like 6502 machine language, BASIC, Pascal, FORTRAN. It looks a lot like a machine language for a computer that does not really exist, so "Woz" has called it his "dream machine". You can read all about it in an old issue of BYTE Magazine (November, 1977, pages 150-159): "SWEET-16 – The 6502 Dream Machine". Another article you may want to find is "SWEET-16 Revisited", by Charles F. Taylor. in MICRO–The 6502/6809 Journal. January, 1982, pages 25-42.

The beauty of SWEET-16 is in its ability to perform 16-bit arithmetic and data moves using automatically updated address pointers. And to add icing to the cake, most of the instructions are only one byte long! You can write extremely compact code, if you are willing to pay the price of slower execution. (A typical program will take half as many bytes, but ten times longer to execute.)

Does anyone really use SWEET-16? Yes, in a big way. I used it in several places inside the early versions of S-C Assembler II. The TED/ASM assembler, and all its descendants (including DOS Tool Kit, TED 11+, Big Mac, Merlin, and others) use SWEET-16 heavily. Several of the programs in the Apple Programmer's Aid ROM use SWEET-16. including the Integer BASIC Renumber/Append program.

The standard version of SWEET-16 is invoked by the 6502 instruction "JSR \$F689"; the bytes immediately following contain opcodes for SWEET-16 to process. SWEET-16 opcodes will be executed until the "RTN" opcode, which returns to 6502 mode.

Programming Model

The SWEET-16 "machine" has sixteen 16-bit registers (R0-R15). R0 is actually the two memory bytes at \$0000 and \$0001. The next two bytes are called R1; R15 is stored in \$001E and \$001F. Several of the registers have special functions: R0 is used as an accumulator (like the 6502's A-register); R12 is the subroutine return stack pointer; R13 receives the results of comparisons; R14 is a status register; R15 is the program address counter.

SWEET-16 REGISTERS		
Register	6502 Address	Purpose
0	\$00,01	Accumulator
1	\$02,03	General
2	\$04,05	General
.	.	.
.	.	.
.	.	.
11	\$16,17	General
12	\$18,19	Subroutine Stack Pointer
13	\$1A,1B	Difference of comparands
14	\$1C,1D	Status
15	\$1E,1F	Program address

There are two general types of opcodes recognized by SWEET-16: register and non-register opcodes. The non-register opcodes all have the form "0x", where x is a hexadecimal digit from 0 through C. (Opcodes 0D, 0E, and 0F are not used.) These opcodes are used for relative branches, subroutine call and return, and to leave SWEET-16. The register opcodes have the format "xR", where x is a hexadecimal digit from 0 through F, and R is a register number (0-F).

SWEET-16 OPCODES

Non-Register Opcodes: RTN, BK, and RS are one byte opcodes. The rest have a second byte which is a relative address, similar to the relative branch addresses used in 6502 opcodes. The conditional branches use status bits found in R14.

00		RTN		Return to 6502 code.
01	ea	BR	addr	Unconditional Branch.
02	ea	BNC	addr	Branch if Carry=0.
03	ea	BC	addr	Branch if Carry=1.
04	ea	BP	addr	Branch if last result positive.
05	ea	BM	addr	Branch if last result negative.
06	ea	BZ	addr	Branch if last result zero.
07	ea	BNZ	addr	Branch if last result non-zero.
08	ea	BM1	addr	Branch if last result = -1.
09	ea	BNM1	addr	Branch if last result not -1.
0A		BK		Execute 6502 BRK instruction.
0B		RS		Return from SWEET-16 subroutine.
0C	ea	BS	addr	Call SWEET-16 subroutine.

Register Opcodes: The SET opcode uses three bytes, to load a 16-bit immediate value into a register. All the rest of the register opcodes only use one byte. ("MA" = memory address)

1n	lo	hi	SET	n,value	Rn <-- value.
2n			LD	n	R0 <-- (Rn).
3n			ST	n	Rn <-- (R0).
4n			LD	@n	MA = (Rn), R0L <-- (MA), Rn <-- MA+1, R0H <-- 0.
5n			ST	@n	MA = (Rn), MA <-- (R0L), Rn <-- MA+1.
6n			LDD	@n	MA = (Rn), R0 <-- (MA, MA+1), Rn <-- MA+2.
7n			STD	@n	MA = (Rn), MA,MA+1 <-- (R0), Rn <-- MA+2.
8n			POP	@n	MA = (Rn)-1, R0L <-- (MA), R0H <-- 0, Rn <-- MA.
9n			STP	@n	MA <-- (Rn)-1, (MA) <-- R0L, Rn <-- MA.
An			ADD	n	R0 <-- (R0) + (Rn).
Bn			SUB	n	R0 <-- (R0) - (Rn).
Cn			POPD	@n	MA = (Rn)-2, MA,MA+1 <-- R0, Rn <-- MA.
Dn			CPR	n	R13 <-- (R0) - (Rn), R14 <-- status flags.
En			INR	n	Rn <-- (Rn) + 1.
Fn			DCR	n	Rn <-- (Rn) - 1.

The S-C Assembler II includes all of the SWEET-16 opcodes, in the formats shown above. You can write programs which mix both 6502 code and SWEET-16 together in any combination.

Here are a few examples which illustrate programming in SWEET-16.

```

                                1000 *-----
                                1010 *      CLEAR A BLOCK OF MEMORY
                                1020 *-----
F689-      1030 SWEET.16      .EQ $F689
0A00-      1040 BLOCK      .EQ $A00
0234-      1050 N          .EQ $234
                                1060 *-----
0800- 20 89 F6 1070 CLEAR      JSR SWEET.16
0803- 10 00 00 1080          SET 0,0      0 FOR CLEARING WITH
0806- 11 00 0A 1090          SET 1,BLOCK  ADDRESS OF BLOCK
0809- 12 34 02 1100          SET 2,N      # BYTES TO CLEAR
080C- 51      1110 .1        ST @1        STORE IN BLOCK
080D- F2      1120          DCR 2
080E- 07 FC   1130          BNZ .1        NOT FINISHED YET
0810- 00      1140          RTN
```

SYMBOL TABLE

```
0A00- BLOCK
0800- CLEAR
.1=080C
0234- N
F689- SWEET.16
```

```
0000 ERRORS IN ASSEMBLY
```

```

                                1000 *-----
                                1010 *      MOVE A BLOCK OF MEMORY
                                1020 *-----
F689-      1030 SWEET.16      .EQ $F689
0A00-      1040 SOURCE      .EQ $A00
0A80-      1050 DESTIN      .EQ $A80
0023-      1060 N          .EQ $23
                                1070 *-----
```

```
0800- 20 89 F6 1080 MOVE      JSR SWEET.16
0803- 11 00 0A 1090      SET 1,SOURCE  ADDRESS OF SOURCE BLOCK
0806- 12 80 0A 1100      SET 2,DESTIN  ADDRESS OF DESTINATION BLOCK
0809- 13 23 00 1110      SET 3,N      # BYTES TO MOVE
080C- 41      1120 .1      LD @1      GET BYTE FROM SOURCE
080D- 52      1130      ST @2      STORE IN DESTINATION
080E- F3      1140      DCR 3
080F- 07 FB      1150      BNZ .1      NOT FINISHED YET
0811- 00      1160      RTN
```

SYMBOL TABLE

```
0A80- DESTIN
0800- MOVE
.01=08CC
0023- N
0A00- SOURCE
F689- SWEET.16
```

0000 ERRORS IN ASSEMBLY

```
1000 *------
1010 *      RENUMBER S-C ASSEMBLER SOURCE CODE
1020 *------
F689- 1030 SWEET.16 .EQ $F689
004C- 1040 HIMEM   .EQ $4C,4D
00CA- 1050 PP      .EQ $CA,CB
1060 *------
1070 RENUMBER
0800- 20 89 F6 1080      JSR SWEET.16
0803- 11 CA 00 1090      SET 1,PP      PP HAS ADDRESS OF SOURCE CODE
0806- 61      1100      LDD @1      GET ADDRESS OF SOURCE CODE
0807- 31      1110      ST 1        ...IN R1
0808- 12 0A 00 1120      SET 2,10     INCREMENT = 10
080B- 13 4C 00 1130      SET 3,HIMEM  HIMEM HAS ADDR OF END OF SOURCE
080E- 63      1140      LDD @3      GET ADDRESS IN HIMEM
080F- 33      1150      ST 3        ...IN R3
0810- 14 DE 03 1160      SET 4,990    START=990 (1ST LINE WILL BE 1000)
0813- 21      1170 .1      LD 1        TEST IF FINISHED
0814- D3      1180      CPR 3
0815- 03 0E      1190      BC .2      YES
0817- 41      1200      LD @1      GET # BYTES IN THIS SOURCE LINE
0818- 35      1210      ST 5        ... INTO R5
0819- 24      1220      LD 4        GET SEQUENCE NUMBER
081A- A2      1230      ADD 2      ADD INCREMENT
081B- 34      1240      ST 4        ... INTO R4 AGAIN
081C- 71      1250      STD @1     ... AND ALSO INTO SOURCE LINE
081D- F1      1260      DCR 1      BACK UP POINTER
081E- F1      1270      DCR 1
081F- F1      1280      DCR 1
0820- 21      1290      LD 1        ADD LENGTH OF LINE TO POINTER
0821- A5      1300      ADD 5
0822- 31      1310      ST 1        POINT AT NEXT SOURCE LINE
0823- 01 EE      1320      BR .1
0825- 00      1330 .2      RTN
0826- 60      1340      RTS
```

SYMBOL TABLE

```
004C- HIHEM
00CA- PP
0800- RENUMBER
.01-0813, .02-0825
F689- SWEET.16
```

0000 ERRORS IN ASSEMBLY

Appendix: Additional Material

A [discussion in August 2021](#) on the 6502 forum notes that some of this document is derived from [the file sweet16.txt](#), possibly collated by W. Sheldon Simms and posted to Usenet, and in turn drawing from BYTE, and from the Merlin assembler documentation. Another source is [Call-A.P.P.L.E. Woz Pak II](#) from 1979.

Also noted in the discussion, the original listing from the BYTE article was captured by Paul R. Santa-Maria in [SW16RB.TXT](#).

Last page update: August 24, 2021.