8.

The architecture of the 6809 processor has up to this point been described piecemeal. Now I'd like to summarize the 6809 processor's architecture, making the description a bit more formal. Please look once again at Figure 4 on page 5 of the MC6809E data booklet.

The PROGRAM COUNTER keeps the machine language program running in order. The Program Counter register contains the 16-bit address of the next instruction to be performed in the program sequence. The Program Counter can be changed directly by the programmer, by jumps and branches within the program, by subroutines, and by stack operations. The Program Counter is one of the POINTER registers.

The two ACCUMULATORS perform simple arithmetic. The A and B Accumulators are each one byte (8 bits) in size. For some operations, the two Accumulators are concatenated, creating a single, 16-bit Accumulator. When A and B are used together as one 16-bit Accumulator, they are collectively called the D Accumulator.

There are two INDEX registers, each 16 bits in size, which can be used to identify memory locations. Although by themselves they are very limited in capability, the Index Registers X and Y can be used, together with various calculated offsets, to load or store data anywhere in memory. To increase their flexibility, the X and Y registers can also be automatically incremented or decremented during the course of a machine language instruction. The Index Registers are also POINTER registers.

There are also two STACK POINTER registers, each 16 bits in size, and each with a different purpose. The User Stack Pointer, the U register, is only controlled by the programmer by pushing and pulling information. This program control allows information to be transferred easily between portions of a program. The Hardware Stack Pointer, the S register, is also used for pushing and pulling information, but is used automatically by the processor to save Program Counter address information during subroutine calls.

After two lessons of heavy abstract learning, you're back with some familiar concepts and practice. At the end of this lesson, you'll be a third of the way through the course — ready to jump into the programming details of the computer. So give this lesson lots of time, and practice each instruction until it's comfortable ... whether or not you know what it's good for!

* Name the 16-bit registers of the 6809.

X and Y, program counter PC, S and U stacks, and the D accumulator.

* Name the 8-bit registers of the 6809.

A and B accumulators, condition code register CC, and direct page register DP.

* What is the purpose of the program counter, PC?

It keeps the machine language program running in order.

* What value does the program counter hold?

The 16-bit address of the next instruction to be performed.

Clock cycles

* What is the purpose of the A and B registers?

To perform simple arithmetic.

* What is the D register?

The concatenation of the 8-bit A and B registers into a single 16-bit register.

* What are the X and Y registers?

Index registers.

* How are index registers most often used?

To identify memory locations.

* What are the S and U registers?

The S register is the hardware stack pointer, and the U register is the user stack pointer.

* How are the S and U registers different?

The U register is reserved for pushing and pulling program information; the S register is used for pushing and pulling as well as for subroutine calls.

* What is the purpose of the condition code register?

The condition code register provides information about the most recent instruction executed by the processor.

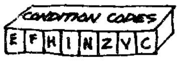
* What is another name for the condition code register?

The flags.

* What does the direct page register store?

The direct page register stores the most-significant half of an address.

The CONDITION CODE register, or flags, is an 8-bit register wherein each bit has a meaning and can be used to make simple judgments (such as greater than, less than, equal to, positive, negative, carry, borrow, etc.) within a program. The Condition Code Register is automatically modified by the results of machine language instructions, or can be changed directly by the programmer.

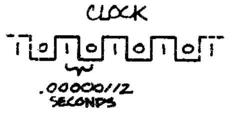


The DIRECT PAGE register, 8 bits wide, is given the most-significant byte of an address. During Direct Addressing, the Direct Page register provides this half of the address, and the program provides the least-significant half of the address. The result is a complete address which can be used to access data in memory.

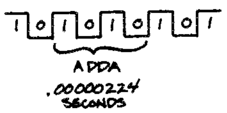
Please read pages 4 and 5, and the first portion of page 6, in the MC6809E data booklet. This section describes the architecture of the 6809 processor. Return to the tape when you have completed the reading.

I wanted you to read that to get a firm idea of the 6809's innards. The next step is getting a handle on some of the 6809's instructions, and for this I'll return to your computer and to a BASIC program. Turn back to your MC6809E data booklet, pages 30 and 31. These pages contain an alphabetical list of the 6809 processor instructions, and are chock full of information.

In the first column is the generalized mnemonic, such as ADD, DECrement, LoaD, etc. The second column shows the specific editor/assembler forms it can take, meaning how to indicate the registers or memory the instruction can use. The next block of information is entitled "Addressing Modes", and provides detailed information on each instruction in that mode, its specific opcode in hexadecimal, the number of bytes the instruction requires for completion, and the number of clock cycles needed for the process.



I haven't mentioned clock cycles before; they are vital to understand when your programming begins to get sophisticated. You've probably heard that the Color Computer runs at .89 MHz. Actually, the precise figure for the computer's speed is .894886 MHz, that is, 894,886 clock pulses per second. Any action taken by the 6809 processor is triggered by one clock pulse; at 894,886 clock pulses per second, that means that the Color Computer's 6809 can't do anything in a shorter time than .00000112 seconds. .00000112 seconds is 1.12 microseconds, slightly longer than a millionth of a second. Knowing this timing is important when writing programs that transfer information properly to the printer port, the RS-232, the cassette, the disk and other devices. Later, when you begin producing audio from your computer, knowing the clock cycles required for each 6809 instruction will be essential.



Back to the booklet, page 30. The description column, toward the right, gives in abbreviated notation the function of each machine language instruction. The symbols and abbreviations are explained at the bottom of the page; glance at the ADD instruction. You will discover that addition using the A Accumulator, mnemonic **ADDA**, is valid in four addressing modes. In the immediate mode, for example, you find that the hexadecimal opcode for this instruction is **8B**, that the complete instruction consists of 2 bytes, and that it takes 2 clock cycles (that is, 2.24 microseconds) to execute. The description column says that the result of A Accumulator plus a value from memory is transferred into the A Accumulator.

The last group of columns provides detailed information about the condition code register — how each flag is affected by the instruction. In the case of the **ADDA** instruction, all five condition code bits are affected (either set or reset) by the results of that command.

These are pretty dense pages. In order to simplify them a little, I've put together a program in BASIC. It's fairly long, so while it's loading, start to get familiar with pages 30 and 31. By the way, there are two program dumps on the tape, just to make certain you've got a good one.

Program #13, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
1 CLS
2 PRINTSTRING$(32,45);
3 PRINTSTRING$(5,191)" INSTRUCTION EXAMPLES "STRING$(5,191);
4 PRINTSTRING$(32,45);
5 PRINT"(1)  ADD                    (ADD)
6 PRINT"(2)  AND            (LOGICAL AND)
7 PRINT"(3)  ASL/ASR (ARITHMETIC SHIFT)
8 PRINT"(4)  COM             (COMPLEMENT)
9 PRINT"(5)  DEC              (DECREMENT)
10 PRINT"(6)  EOR          (EXCLUSIVE OR)
11 PRINT"(7)  INC             (INCREMENT)
12 PRINT"(8)  LSL/LSR     (LOGICAL SHIFT)
13 PRINT"(9)  NEG              (NEGATIVE)
14 PRINT"(A)  OR             (LOGICAL OR)
15 PRINT"(B)  ROL/ROR            (ROTATE)
16 PRINT"(C)  SUB              (SUBTRACT)
17 PRINTCHR$(191)"  TOUCH 1 - C TO DEMONSTRATE  ";:POKE1535,191
18 A$=INKEY$:IFA$=""THEN18
19 A=ASC(A$):A=A-48:IFA<1 OR A>19 THEN18
20 ONA GOSUB23,37,50,76,86,97,112,122,138,18,18,18,18,18,18,18,1
49,163,192
21 RUN
22 GOTO22
23 CLS:NF=0:ZF=0:CF=0
24 PRINT"------> ADD TWO NUMBERS <------"
25 GOSUB225:IFQQ=1THEN23
26 INPUT"VALUE TO ADD FROM MEMORY OR FROMOTHER REGISTER (HEX)";A
2$:A$=A2$
27 Q=0:GOSUB210:IFQ=1THEN23
28 X=A:A2=A:GOSUB212:Q2$=Q$
29 X=A1+A2:A3=X
30 IFX>255THENX=X-256:CF=1:A3=X
31 IFX=0THENZF=1
```

* How is the direct page register used?

In the direct addressing mode, the register is used to create a complete address.

* What is an addressing mode?

How the machine language program gets its information.

* What do you call the verbal description of a processor command?

A mnemonic.

* What is the proper name for a processor command?

An opcode.

* What is the clock speed of the Color Computer?

.89 MHz (.894886 MHz or 894,886 pulses per second).

* How long is one clock pulse on the Color Computer?

Approximately .00000112 seconds or 1.12 microseconds. (More accurately, 1.11746 microseconds).

* How long is a microsecond?

One millionth of a second.

* The MUL (multiply) instruction takes 11 clock cycles. How long is this on the Color Computer?

11 times 1.11746 microseconds, or 12.29206 microseconds.

* LDA immediate and LDB immediate each take 2 clock cycles. How long is each instruction on the Color Computer?

2 times 1.11746 microseconds, or 2.23492 microseconds.

* STD extended takes 6 clock cycles. How long is this?

6 times 1.11746 microseconds, or 6.70476 microseconds.

* MULtiply is A times B, with the result in D. If a multiplication program consists of LDA and LDB immediate (each 2 clock cycles), MULtiply (11 clock cycles), and STD extended (6 clock cycles), how long is this?

(2+2+11+6) times 1.11746 microseconds, or 23.46666 microseconds).

* At 23.46666 microseconds per multiplication program, how many complete multiplication programs can the Color Computer do in one second?

The Color Computer can perform 42,613 multiplication programs per second.

* What is the purpose of the condition code register?

The condition code register provides information about the most recent instruction executed by the processor.

* In the following exercises, give the results of the instruction, where the result is found, and the effect on the three flags N, Z and C (condition codes negative, zero and carry). For example, the problem: A contains $41. Execute ADDA #$CC. The answer: A contains $0D. Carry flag set. Zero and negative flags reset.

* Problem: A contains $04. Execute ADDA #$FB.

Answer: A contains $FF. Negative flag set. Zero and carry flags reset.

```
32 GOSUB212:Q3$=Q$
33 PRINT
34 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)Q2$"    "A2$:PRINTTAB(5)STRI
NG$(20,45):PRINTTAB(5)Q3$"    ";:IFA3<16THENPRINT"Ø"+HEX$(A3) ELS
EPRINTHEX$(A3)
35 GOSUB224
36 GOSUB222:RETURN
37 CLS:NF=Ø:ZF=Ø:CF=Ø
38 PRINT"--> LOGICAL AND  TWO NUMBERS <--";
39 GOSUB225:IFQQ=1THEN37
40 INPUT"VALUE TO AND FROM MEMORY OR FROMOTHER REGISTER (HEX)";A
2$:A$=A2$
41 Q=Ø:GOSUB21Ø:IFQ=1THEN37
42 X=A:A2=A:GOSUB212:Q2$=Q$
43 X= A1 AND A2 : A3=X
44 IFX=ØTHENZF=1
45 GOSUB212:Q3$=Q$
46 PRINT
47 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)Q2$"    "A2$:PRINTTAB(5)STRI
NG$(2Ø,45):PRINTTAB(5)Q3$"    ";:IFA3<16THENPRINT"Ø"+HEX$(A3)ELSE
PRINTHEX$(A3)
48 GOSUB224
49 GOSUB222:RETURN
5Ø CLS:NF=Ø:ZF=Ø:CF=Ø
51 PRINT"ARITHMETIC SHIFT LEFT OR RIGHT":PRINT"TOUCH L OR R"
52 A$=INKEY$:IFA$="L"ORA$="l"THEN53ELSEIFA$="R"ORA$="r"THEN63ELS
E52
53 CLS:PRINT"---> ARITHMETIC SHIFT LEFT <---"
54 GOSUB225:IFQQ=1THEN53
55 X=A*2:A2=X
56 IFX>255THENX=X-256:CF=1:A2=X
57 IFX=ØTHENZF=1
58 GOSUB212:Q2$=Q$
59 PRINT
6Ø PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"<---- SHIFT ----":PRINTTAB
(5)Q2$"    ";:IFA2<16THENPRINT"Ø"+HEX$(A2) ELSEPRINTHEX$(A2)
61 GOSUB224
62 GOSUB222:RETURN
63 CLS:PRINT"---> ARITHMETIC SHIFT RIGHT <---";
64 GOSUB225:IFQQ=1THEN63
65 IFA>127THENNF=1
66 X=FIX(A/2):IFX>63THENX=X OR128:A2=X:ELSEA2=X
67 IF(A/2)<>FIX(A/2)THENCF=1
68 IFX=ØTHENZF=1
69 GOSUB212:Q2$=Q$
7Ø PRINT
71 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"---- SHIFT ---->":PRINTTAB
(5)Q2$"    ";:IFA2<16THENPRINT"Ø"+HEX$(A2) ELSEPRINTHEX$(A2)
72 GOSUB224
73 IFNF=1 THEN PRINT:PRINT"NOTE BIT 7; SEE DATA BOOKLET.":GOTO75
74 IFCF=1 AND NF=Ø THEN PRINT:PRINT"NOTE CARRY FLAG; SEE DATA BO
OK."
75 GOSUB222:RETURN
76 CLS:NF=Ø:ZF=Ø:CF=1
77 PRINT"----> COMPLEMENT A NUMBER <----"
78 GOSUB225:IFQQ=1THEN76
79 X=NOTA AND 255:A2=X:GOSUB212:Q2$=Q$
8Ø IFX=ØTHENZF=1
81 IFX>127THENNF=1
82 PRINT:PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"** COMPLEMENT **":PR
INTTAB(5)Q2$"    ";:IFA2<16THENPRINT"Ø"+HEX$(A2) ELSEPRINTHEX$(A2
)
83 GOSUB224
84 PRINT:PRINT"NOTE CARRY FLAG; SEE DATA BOOK."
85 GOSUB222:RETURN
86 CLS:NF=Ø:ZF=Ø:CF=Ø
87 PRINT"----> DECREMENT A NUMBER <----"
88 GOSUB225:IFQQ=1THEN86
89 X=A-1:A2=X:IFX<ØTHENX=255:A2=X:NF=1
9Ø IFX=ØTHENZF=1
91 IFX>127THENNF=1
92 GOSUB212:Q2$=Q$
93 PRINT
94 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"** DECREMENT **":PRINTTAB(
5)Q2$"    ";:IFA2<16THENPRINT"Ø"+HEX$(A2) ELSEPRINTHEX$(A2)
95 GOSUB224
96 GOSUB222:RETURN
97 CLS:NF=Ø:ZF=Ø:CF=Ø
98 PRINT"LOGICAL EXCLUSIVE-OR TWO NUMBERS";
99 GOSUB225:IFQQ=1THEN97
1ØØ INPUT"VALUE TO EXCLUSIVE-OR, TAKEN     FROM MEMORY OR FROM AN
OTHER      REGISTER";A2$:A$=A2$
1Ø1 QQ=Ø:GOSUB21Ø:IFQQ=1THEN97
```

```
102 X=A:A2=A:GOSUB212:Q2$=Q$
103 X=(A1 AND NOT(A2)) OR (NOT(A1) AND A2):A3=X
104 IFX=0THENZF=1
105 IFX>127THENNF=1
106 GOSUB212:Q3$=Q$
107 PRINT
108 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)Q2$"    "A2$:PRINTTAB(5)STR
ING$(20,45):PRINTTAB(5)Q3$"    ";:IFA3<16THENPRINT"0"+HEX$(A3) EL
SEPRINTHEX$(A3)
109 GOSUB224
110 GOSUB222:RETURN
111 RETURN
112 CLS:NF=0:ZF=0:CF=0
113 PRINT"---->  INCREMENT A NUMBER <----"
114 GOSUB225:IFQQ=1THEN112
115 X=A+1:A2=X:IFX>255THENX=0:A2=X:ZF=1:NF=0
116 IFX>127THENNF=1
117 GOSUB212:Q2$=Q$
118 PRINT
119 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"** INCREMENT **":PRINTTAB
(5)Q2$"    ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
120 GOSUB224
121 GOSUB222:RETURN
122 CLS:NF=0:ZF=0:CF=0
123 PRINT"> LOGICAL SHIFT LEFT OR RIGHT <"
124 PRINT"TOUCH L OR R"
125 A$=INKEY$:IFA$="L"ORA$="l"THEN126ELSEIFA$="R"ORA$="r"THEN129
ELSE125
126 CLS:PRINT"----> LOGICAL SHIFT LEFT <----"
127 GOSUB225:IFQQ=1THEN126
128 GOTO55
129 CLS:PRINT"----> LOGICAL SHIFT RIGHT <----"
130 GOSUB225:IFQQ=1THEN129
131 X=FIX(A/2):A2=X:IFA/2<>FIX(A/2)THENCF=1
132 IFX=0THENZF=1
133 GOSUB212:Q2$=Q$
134 PRINT
135 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"---- SHIFT ---->":PRINTTA
B(5)Q2$"    ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
136 GOSUB224
137 GOSUB222:RETURN
138 CLS:NF=0:ZF=0:CF=0
139 PRINT"------> NEGATE A NUMBER <------"
140 GOSUB225:IFQQ=1THEN138
141 REM
142 REM
143 X=(NOTA AND 255)+1:A2=X:GOSUB212:Q2$=Q$
144 IFX=0THENZF=1:CF=1
145 IFX>127THENNF=1
146 PRINT:PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"** NEGATIVE **":PRI
NTTAB(5)Q2$"    ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
147 GOSUB224
148 GOSUB222:RETURN
149 CLS:NF=0:ZF=0:CF=0
150 PRINT"---> LOGICAL OR TWO NUMBERS <---";
151 GOSUB225:IFQQ=1THEN149
152 INPUT"VALUE TO OR FROM MEMORY OR FROM ANOTHER REGISTER (HEX)
";A2$:A$=A2$
153 QQ=0:GOSUB210:IFQQ=1THEN149
154 X=A:A2=A:GOSUB212:Q2$=Q$
155 X=A1 OR A2 : A3=X
156 IFX=0THENZF=1
157 IFX>127THENNF=1
158 GOSUB212:Q3$=Q$
159 PRINT
160 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)Q2$"    "A2$:PRINTTAB(5)STR
ING$(20,45):PRINTTAB(5)Q3$"    ";:IFA3<16THENPRINT"0"+HEX$(A3) EL
SEPRINTHEX$(A3)
161 GOSUB224
162 GOSUB222:RETURN
163 CLS:NF=0:ZF=0:CF=0
164 PRINT"----> ROTATE LEFT OR RIGHT <----";
165 PRINT"TOUCH L OR R"
166 A$=INKEY$:IFA$="L"ORA$="l"THEN167ELSEIFA$="R"ORA$="r"THEN180
ELSE166
167 CLS:PRINT"STATE OF CARRY FLAG? (0 OR 1) ";
168 A$=INKEY$:IFA$="0" OR A$="1"THENPRINTA$:CF=VAL(A$):ELSE168
169 GOSUB225:IFQQ=1THEN167
170 X=A*2:A2=X
171 IFX<256THENX=X ORCF:A2=X:CF=0:GOTO173:ELSE172
172 X=X-256:X=X ORCF:CF=1:A2=X
173 IFX=0THENZF=1
174 IFX>127THENNF=1
```

* Problem: B contains $AA. Execute ANDB #$55.

Answer: B contains $00. Zero flag set. Negative flag reset. Carry flag unaffected.

* Problem: B contains $AA. Execute ANDB #$5F.

Answer: B contains $0A. Zero and negative flags reset. Carry flag unaffected.

* Problem: A contains $FF. Execute ORA #$A5.

Answer: A contains $FF. Negative flag set. Zero flag reset. Carry flag unaffected.

* Problem: A contains $AA. Execute ORA #$55.

Answer: A contains $FF. Negative flag set. Zero flag reset. Carry flag unaffected.

* Problem: A contains $00. Execute ORA #$00.

Answer: A contains $00. Zero flag set. Negative flag reset. Carry flag unaffected.

* Problem: B contains $F0. Execute ORB #$0F.

Answer: B contains $FF. Negative flag set. Zero flag reset. Carry flag unaffected.

* Problem: B contains $FF. Execute COMB.

Answer: B contains $00. Zero flag set. Negative flag reset. Carry flag always set by COM instruction.

* Problem: A contains $AA. Execute COMA.

Answer: A contains $55. Zero and negative flags reset. Carry flag always set by COM instruction.

* Problem: A contains $04. Execute ADDA #$FC.

Answer: A contains $00. Zero and carry flags set. Negative flag reset.

* Problem: A contains $04. Execute ADDA #$FD.

Answer: A contains $01. Carry flag set. Negative and zero flags reset.

* Problem: B contains $80. Execute SUBB #$01.

Answer: B contains $7F. All flags reset.

* Problem: B contains $81. Execute SUBB #$01.

Answer: B contains $80. Negative flag set. Zero and carry flags reset.

* Problem: B contains $80. Execute SUBB #$80.

Answer: B contains $00. Zero flag set. Negative and carry flags reset.

* Problem: B contains $80. Execute SUBB #$81.

Answer: B contains $FF. Negative and carry flags set. Zero flag reset.

* Problem: A contains $FF. Execute ANDA #$FF.

Answer: A contains $FF. Negative flag set. Zero flag reset. Carry flag unaffected.

* Problem: A contains $FF. Execute ANDA #$A5.

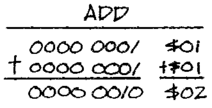
Answer: A contains $A5. Negative flag set. Zero flag reset. Carry flag unaffected.

```
175 GOSUB212:Q2$=Q$
176 PRINT
177 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"<--- ROTATE ---":PRINTTAB
(5)Q2$"    ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
178 GOSUB224
179 GOSUB222:RETURN
180 CLS:PRINT"STATE OF CARRY FLAG? (0 OR 1) ";
181 A$=INKEY$:IFA$="0" OR A$="1"THENPRINTA$:CF=VAL(A$):ELSE181
182 GOSUB225:IFQQ=1THEN180
183 X=(FIX(A/2))OR(CF*128):A2=X
184 IFFIX(A/2)<>A/2THENCF=1ELSECF=0
185 IFX=0THENZF=1
186 IFX>127THENNF=1
187 GOSUB212:Q2$=Q$
188 PRINT
189 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"--- ROTATE --->":PRINTTAB
(5)Q2$"    ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
190 GOSUB224
191 GOSUB222:RETURN
192 CLS:NF=0:ZF=0:CF=0
193 PRINT"----> SUBTRACT TWO NUMBERS <----";
194 GOSUB225:IFQQ=1THEN192
195 REM
196 REM
197 INPUT"VALUE TO SUBTRACT, TAKEN FROM   MEMORY OR OTHER REGIST
ER (HEX)";A2$:A$=A2$
198 QQ=0:GOSUB210:IFQQ=1THEN192
199 X=A:A2=A:GOSUB212:Q2$=Q$
200 X=A1-A2:A3=X
201 IFX<0THENCF=1:X=X+256:A3=X
202 IFX=0THENZF=1
203 IFX>127THENNF=1
204 GOSUB212:Q3$=Q$
205 PRINT
206 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)Q2$"    "A2$:PRINTTAB(5)STR
ING$(20,45):PRINTTAB(5)Q3$"    ";:IFA3<16THENPRINT"0"+HEX$(A3) EL
SEPRINTHEX$(A3)
207 GOSUB224
208 GOSUB222:RETURN
209 FORN=1TO1000:NEXT:RETURN
210 A=VAL("&H"+A$):IFA<0 OR A>255 THEN PRINT"VALUE OUT OF RANGE"
:GOSUB209:QQ=1:RETURN
211 QQ=0:RETURN
212 C=INT(X/128):D=C*128
213 E=INT((X-D)/64):F=E*64
214 G=INT((X-D-F)/32):H=G*32
215 I=INT((X-D-F-H)/16):J=I*16
216 K=INT((X-D-F-H-J)/8):L=K*8
217 M=INT((X-D-F-H-J-L)/4):N=M*4
218 O=INT((X-D-F-H-J-L-N)/2):P=O*2
219 Q=INT(X-D-F-H-J-L-N-P)
220 Q$=STR$(C)+STR$(E)+STR$(G)+STR$(I)+STR$(K)+STR$(M)+STR$(O)+S
TR$(Q)
221 RETURN
222 PRINT"PRESS ENTER TO CONTINUE";
223 A$=INKEY$:IFA$<>CHR$(13)THEN223ELSERETURN
224 PRINT:PRINT"FLAGS:":PRINTTAB(7)"N  Z  C":PRINTTAB(6)C;ZF;CF:
PRINT:RETURN
225 INPUT"VALUE IN ACCUMULATOR (HEX)";A1$:A$=A1$
226 QQ=0:GOSUB210:IFQQ=1THENRETURN
227 X=A:A1=A:GOSUB212:Q1$=Q$
228 RETURN
```

RUN this program. On the screen are 12 common instructions selected from the total of 59 that the 6809 processor can execute. For your amusement, I've numbered them in hexadecimal.

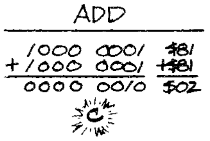
Some of the instructions will already be familiar, but I'd like you to get a detailed idea of how each one works, and what its results are. Here's how it goes. You will input all values in hexadecimal, but the display will be done in both hex and binary, so that the inner workings of each instruction will be evident. Although there are five flags, I've chosen only the most used ones (negative, zero, and carry) to display in these examples.

You can start with a familiar instruction, selection #1, ADD. Touch 1 on the keyboard.

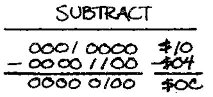


As you can see, for simplicity I'm making the assumption that the initial value will always be in an accumulator, and that all values will be 8 bits. Enter hex number **01** as the accumulator value. The second prompt appears. To the accumulator value will be added a value from memory or from another register in the processor. You'll add 1 to this. Type **$01** and hit <ENTER>.

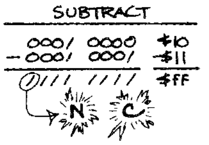
On your display are the two numbers being added, and the result, which is **$02**. All three are displayed in both binary and hexadecimal. The flags reveal three pieces of information: that the resulting number is not negative, it is not zero, and there was no carry generated by the calculation.



Hit <ENTER>, and touch 1 again. This time, enter hex **$81** into the accumulator. Add to this the value hex **$81**. The result is the same as before — **$02**. But the carry flag reveals something very important. It tells you that, although the apparent 8-bit result is **$02**, the addition actually produced a number larger than 8 bits.

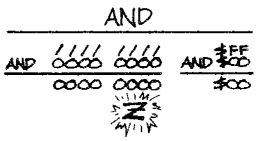


Now some subtraction. Hit <ENTER>, and touch selection C. Enter **$10** into the accumulator. From this, subtract the value, say, **$04**. The result is **$0C**, a non-zero positive number, as the flags indicate. Hit <ENTER> and touch C again. Enter **$10** into the accumulator again, but this time subtract **$11**. The result is **$FF**. The flags tell the story. It is a negative number, and the carry/borrow flag shows that a borrow was required to complete it. That carry/borrow flag is vital to recognize.



Add and subtract are straightforward. Try each of them a few times at the end of this lesson. I'll go through the rest of the instructions in this group. When I'm done, you're on your own for a while. Let me walk over to the kitchen . . .

Hit <ENTER> to get back to the menu. You've tried ADD and SUBtract, so now touch 2 for logical AND. This is the first of the logical instructions (also called Boolean Algebra, but we'll forget *that* term). Logical AND requires both statements of a pair to be true for the result to be true. For example, this statement demonstrates logical AND: "If I break this plate AND Claire sees the broken plate, then she will scream at me". If either statement is not true — that is, if either I didn't break the plate, or if Claire didn't see the broken plate — then I'll get off. Here comes Claire now. <Breaking plate. "Look, you broke that plate! Arrrggh!!" etc.> Likewise, in binary arithmetic, both bits must be ones — that is, both bits must be true — for the result to be true. Enter **$FF** into the accumulator, and **$00** into memory. Each bit of the accumulator is ANDed with each corresponding bit in the operand. The results here are all zeros. The zero flag goes on.



*** Problem: A contains $EC. Execute COMA.**

Answer: A contains $13. Zero and negative flags reset. Carry flag always set by COM instruction.

*** Problem: A contains $47. Execute COMA.**

Answer: A contains $B8. Negative flag set. Zero flag reset. Carry flag always set by COM instruction.

*** Problem: B contains $0F. Execute COMB.**

Answer: B contains $F0. Negative flag set. Zero flag reset. Carry flag always set by COM instruction.

*** Problem: A contains $AA. Execute EORA #$00.**

Answer: A still contains $AA. Negative flag set. Zero flag reset. Carry flag not affected.

*** Problem: A contains $AA. Execute EORA #$AA.**

Answer: A contains $00. Zero flag set. Negative flag reset. Carry flag not affected.

*** Problem: A contains $AA. Execute EORA #$FF.**

Answer: A contains $55. Negative and zero flag reset. Carry flag not affected. Has effect of COMA except does not affect carry flag.

*** Problem: B contains $00. Execute EORB #$C8.**

Answer: B contains #$C8. Negative flag set. Zero flag reset. Carry flag not affected.

* Problem: A contains $0F. Execute ASLA.

Answer: A contains $1E. All flags reset.

* Problem: A contains $0F. Execute ASRA.

Answer: A contains $07. All flags reset.

* Problem: A contains $88. Execute ASLA.

Answer: A contains $10. Carry flag set as bit drops into "bucket". Negative and zero flags reset.

* Problem: A contains $88. Execute ASRA.

Answer: A contains $C4 (bit 7 duplicated at left). Negative flag set. Zero and carry flags reset.

* Problem: B contains $88. Carry flag is set. Execute ROLB.

Answer: B contains $11. Carry flag set. Zero and negative flags reset.

* Problem: B contains $88. Carry flag is set. Execute RORB.

Answer: B contains $C4. Negative flag is set. Carry and zero flags are reset.

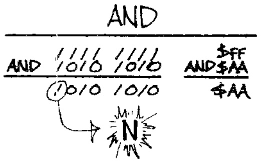
* Problem: A contains $02. Execute DECA.

Answer: A contains $01. Negative and zero flags reset. Carry flag not affected.

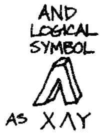
* Problem: A contains $01. Execute DECA.

Answer: A contains $00. Zero flag set. Negative flag reset. Carry flag not affected.

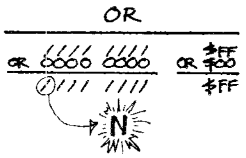
Hit <ENTER>, and touch 2. Again enter **$FF** into the accumulator. This time try **$AA** as the memory contents, and hit <ENTER>. Each bit of the pair is ANDed, and the result is **$AA**. The negative flag flips on.



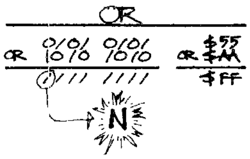
Contrast this with logical OR. Hit <ENTER>, and touch A. Logical OR states that if either or both of two conditions is true, then the result will be true. For example, this statement describes logical OR: "If I eat this pie OR I eat this ice cream, then I will be pleased." Binary numbers can't measure my level of pleasure, but they can report that <mouth full> I will be pleased if I eat either the pie or the ice cream, or if I eat both. Likewise, in binary arithmetic, if either number is a one — that is, if either number is true — the result will be true.



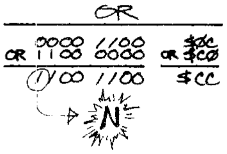
Enter **$FF** into the accumulator. Then enter **$00** as the operand. You can see two things: first, you find that since all bits in the accumulator are one, all bits in the result will be one, regardless of the operand; second, the negative flag flips on because bit 7 is a one. Hit <ENTER>, touch A, and put **$55** in the accumulator this time. As the operand, enter **$AA**. The numbers I chose here have alternating bits, just to demonstrate that neither byte need have bits in common — it is truly an either/or situation. Note the negative flag is on.



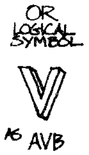
Just one more OR. Hit <ENTER>, and touch A. Put **$0C** in the accumulator, and **$C0** into the operand. In this example, you can see that where neither bit is true, zeros do result from the logical OR process. Again, you'll want to try examples of logical OR at the end of the lesson.



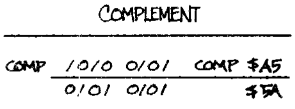
Move on to COMplement, selection 4. Hit <ENTER>, and touch 4. A number's complement is created by reversing all the binary digits in that number; it's the equivalent of a logical NOT. For example, enter **$A5**. All the zeros become ones, all the ones become zeros. The result after the complement is **$5A**. Hit <ENTER>, touch 4, and place **$FF** in the accumulator. The complement of **$FF** is **$00**. The zero flag flips on. Notice that in this instruction, the carry flag always turns on, regardless of the result, merely to indicate the completion of the instruction.



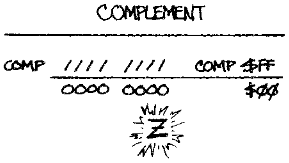
The logical Exclusive-OR instruction is next. This is a command used to "toggle" individual bits. You understand how logical AND, OR and NOT work. Just for review, two binary values ANDed together give a one result only if both values are one, as I mentioned above. Two binary values ORed together give a one result if either value is a one. Logical NOT simply flips one bits to zero, and zero bits to one, as in the COMplement statement you've already tried.



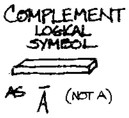
Logical Exclusive-OR gives a true result if either, but *not both*, of the premises are true. That's a little hard to analogize to real life, but since I'm still here in the kitchen, it might go something like this: "If I eat this full-course Chinese dinner Exclusive-OR if I eat this full-course Italian



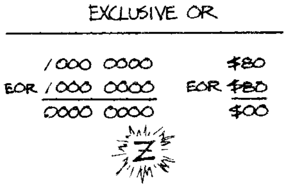
dinner, then I will be content." If I eat neither, I won't be content; if I eat both, I'll probably explode. Logical Exclusive-OR is the equivalent of the quantity (A and NOT B) ORed with the quantity (NOT A and B) . . . but that's not very revealing either.



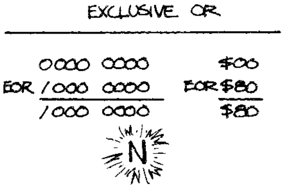
Try it this way: if two bits are different, the result will be one. If the bits are the same, the result will be zero. What makes this idea useful is that information can be "toggled" back and forth between numbers. Turn to the program for a visual example.



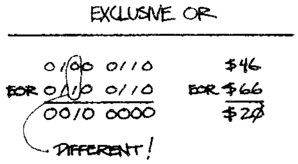
Hit <ENTER> and touch 6. You're going to toggle between, say, **$80** and **$00**. Enter **$80** into the accumulator. Pause here and think about hex **$80** and **$00**. In binary, **$80** is **1000 0000**, and **$00** is **0000 0000**. Only bit 7 is different here. You need to find a value that, when Ex-ORed with **1000 0000**, gives **0000 0000**. Recall how Exclusive-OR works: to get a zero result, the two bits being Ex-ORed must be the same. That suggests that **1000 0000** Ex-ORed with **1000 0000** should give an all-zero result. So the hex equivalent of **1000 0000** is what you want . . . and that's **$80**. Enter **$80**, and look at the binary display. Incidentally, the zero flag flipped on.



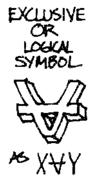
Hit <ENTER> and touch 6 again. This time, enter the result from the Ex-OR you just did. Enter **$00** into the accumulator. And enter **$80** as the operand. The result is **$80**. Here's why Exclusive-OR is called a toggle function. When value X is Ex-ORed with value Q, the result is value Y. When value Y is Ex-ORed with value Q, the result is value X. Under the Exclusive-OR function, value Q becomes a toggle, flipping back and forth between values X and Y.



Remember the flashing "F" at the top of the screen when you load cassettes into the Color Computer? This alternates value **$46** with value **$66**. Hit <ENTER> and touch 6 again. Enter **$46** into the accumulator, and **$66** as the operand. The result should be **$20**. **$20** can then be used in a program as a toggling value. **$46** Exclusive OR **$20** is **$66, $66 EOR $20** is **$46**. Uppercase F becomes lowercase F, and vice versa. And the advantage to a toggling value is this: you don't have to know *which* state the original value is in to toggle it. That's ideal, because in this example, the tape-loading program doesn't have to keep track of which "F" it's displayed.



But enough of Exclusive-OR. You can try it at the end of this lesson.



Shifts and rotates are interesting commands. Essentially, they are binary multiplication or division by two. In the decimal system, a left shift is multiplication by ten, a right shift is division by ten. If that doesn't make immediate sense, consider the number 247. Shift it to the left and it becomes 2470; shift 247 to the right and it becomes 24.7 . . . multiplication and division by ten. The difference between types of binary shifts in the 6809 has to do with what happens to the bits on either end of the byte.

* Problem: A contains $00. Execute DECA.

Answer: A contains $FF. Negative flag is set. Zero flag is reset. Carry flag not affected.

* Problem: B contains $FE. Execute INCB.

Answer: B contains $FF. Negative flag is set. Zero flag is reset. Carry flag not affected.

* Problem: B contains $FF. Execute INCB.

Answer: B contains $00. Zero flag is set. Negative flag is reset. Carry flag not affected.

* Problem: B contains $00. Execute INCB.

Answer: B contains $01. Negative and zero flags are reset. Carry flag not affected.

* Problem: B contains $01. Execute NEGB.

Answer: B contains $FF. Negative and carry flags are set. Zero flag is reset.

* Problem: B contains $00. Execute NEGB.

Answer: B contains $00. Zero flag is set. Negative and carry flags are reset.

* Problem: B contains $80. Execute NEGB.

Answer: B contains $80. Negative and carry flags are set. Zero flag is reset.

* Problem: A contains $AA. Execute NEGA.

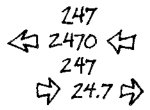
Answer: A contains $56. All flags are reset.

Left and right shifts

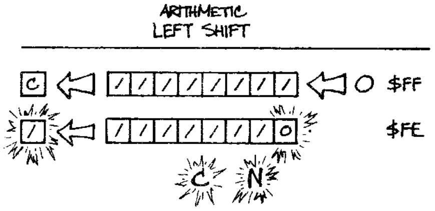
*** What is the purpose of the condition code register?**

The condition code register provides information about the most recent instruction executed by the processor.

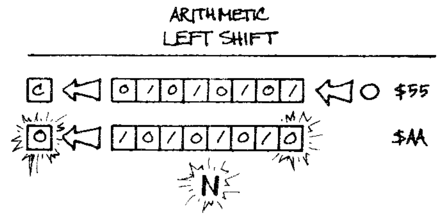
An arithmetic shift to the left puts a zero into the rightmost position; a similar shift to the right leaves a trail of the value of the leftmost bit. The bit that is shifted out the end of the byte falls into the carry flag; in a situation like this, the carry flag is sometimes called a "bit bucket". A logical shift left is identical to an arithmetic shift, but a logical shift right leaves a zero in the leftmost position. Again, the bit falling off the end drops into the carry flag. Finally, a rotate command is circular, as the bits move left or right through the carry flag. Try the arithmetic shift here.

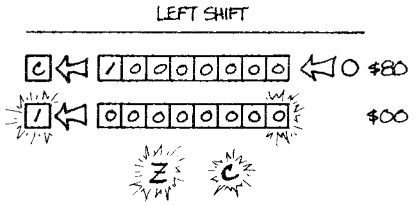


Hit <ENTER>, and touch 3. You've got a prompt for an arithmetic shift. Do the left shift first; touch L. Put a hex value **$FF** into the accumulator. The row of bits is shifted left, a zero follows from the right, and the leftmost bit ends up in the carry flag. Notice that since bit 7 is high, the negative flag also goes up.

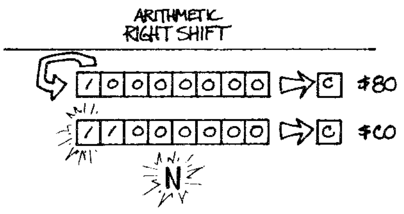


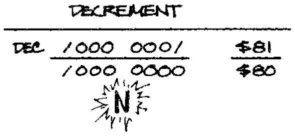
Hit <ENTER>, and touch 3. Touch L again. Put **$55** into the accumulator. Notice how the bits all move left. This number turns negative (becoming **$AA**), but the carry flag is zero. You can explore all those details later; try a right shift now.



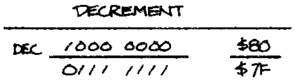


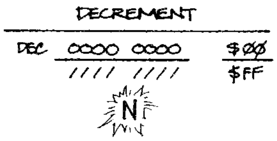
Hit <ENTER>, touch 3, and touch R. Put **$80** into the accumulator. This time observe bit 7, the leftmost bit. It begins to leave a trail of ones behind it; the value after shifting is **$C0**. Hit <ENTER>, touch 3, touch R, and enter **$C0**. The trail of ones continues to follow.



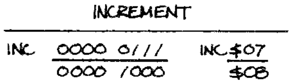


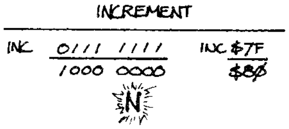
I'm going to skip doing the Logical Shifts and Rotates in this explanation; you can check out selection 8 and selection B on your own at the end of the lesson.



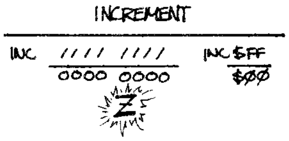


Move on to the next 6809 processor command. Hit <ENTER>, and touch 5. This is a decrement by one command. Enter **$81** into the accumulator. **$81** minus one is **$80**. The negative flag is on. Decrement it one more time; hit <ENTER>, touch 5, and put **$80** into the accumulator. The value becomes **$7F**; the negative flag is off. One more thing to notice with the decrement command. Hit <ENTER>, touch 5, and put **$00** into the accumulator. **$00** minus 1 is **$FF**. The negative flag flips on.

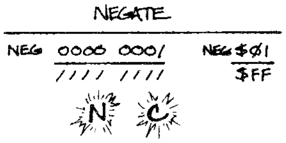




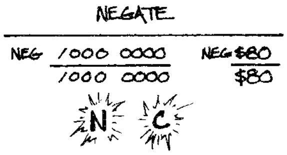
The opposite of the decrement is the increment, also a straightforward command. Hit <ENTER>, and touch 7. Enter **$07** into the accumulator. The value is incremented by one to **$08**. Not much there; all flags are off. Hit <ENTER>, and touch 7. This time put **$7F** into the accumulator. The value increments from **$7F** to **$80**. The negative flag flips on. Finally, hit <ENTER> and touch 7 again. Enter **$FF** into the accumulator. The number increments with the result being **$00**.



There's just one selection left, and that's NEGate, selection 9. Hit <ENTER>, and touch 9. Enter **$01** into the accumulator. The negative of **$01** is **$FF**. If you recall from an earlier lesson, you counted backwards from zero in one programming example, and it makes sense that one less than zero, –1 in decimal, would by **$FF** in 8-bit data.



Hit <ENTER> and touch 9. Put **$80** into the accumulator. The result is — **$80**! I'll leave you to check the flags and ponder that result.



Please review this lesson, spend some time with pages 30 and 31 of the MC6809E data booklet, and — most of all — keep using this program. Try every example; work the results out on paper, and see if you agree with the final display. Examine how the binary data works, how the instructions perform, and what the flags mean.

9.

Making things happen on your 6809-based Color Computer is the point of all this. I've created this series because your computer is a special machine — not just an isolated microprocessor, but an interrelated group of components capable of video, sound, storage and communication, with add-ons like joysticks and disks and printers. So while you're making your way through the intricacies of the 6809 itself, I'm also going to provide you with the information you need to use the whole computer.

That means I've got to talk about two things specific to the Color Computer: memory maps and smart components.

The memory map of your computer describes the way its 65,536 individual addresses are organized . . . what goes where. Simplicity is always important in laying out a memory map, and that holds true for the Color Computer. I've reproduced the Color Computer memory maps in the documentation so you can follow along.

There are a few special considerations in this machine, but the memory map I'll describe is what's set up when you turn the power on. Read/write memory — also known as random-access memory, or RAM — is located (talking in hexadecimal now) from address **$0000** to **$7FFF**. That's 32K of memory; if you have a 16K computer, your RAM ends at **$3FFF**. **$4000** to **$7FFF** remains unused until you fill it.

The BASIC language is made up of machine-language instructions and data, so it too occupies part of the memory map. BASIC is broken up into two halves, each half 8K long. From hex **$8000** to **$9FFF** you will find Extended Color BASIC, and from hex **$A000** to **$BFFF** you will find Color BASIC.

Starting at **$C000** is a blank space. As far as the processor is concerned, no memory is "blank" per se, but an off-the-shelf Color Computer doesn't have anything connected at **$C000**. However, when you plug in a ROMpack cartridge,

Practical application of your 6809 learning means knowing something about this particular 6809 environment. And that means knowing the Color Computer better. It's not the only 6809 machine there is, so you'll need to learn all new details if you purchase a Whatzit-99 or the CompuBlob.

* What do you call the description of how the computer's designers have arranged its memory?

A memory map.

* How many memory locations are there in the Color Computer?

65,536 locations.

* What is the address range of the Color Computer, in hex?

$0000 to $FFFF.

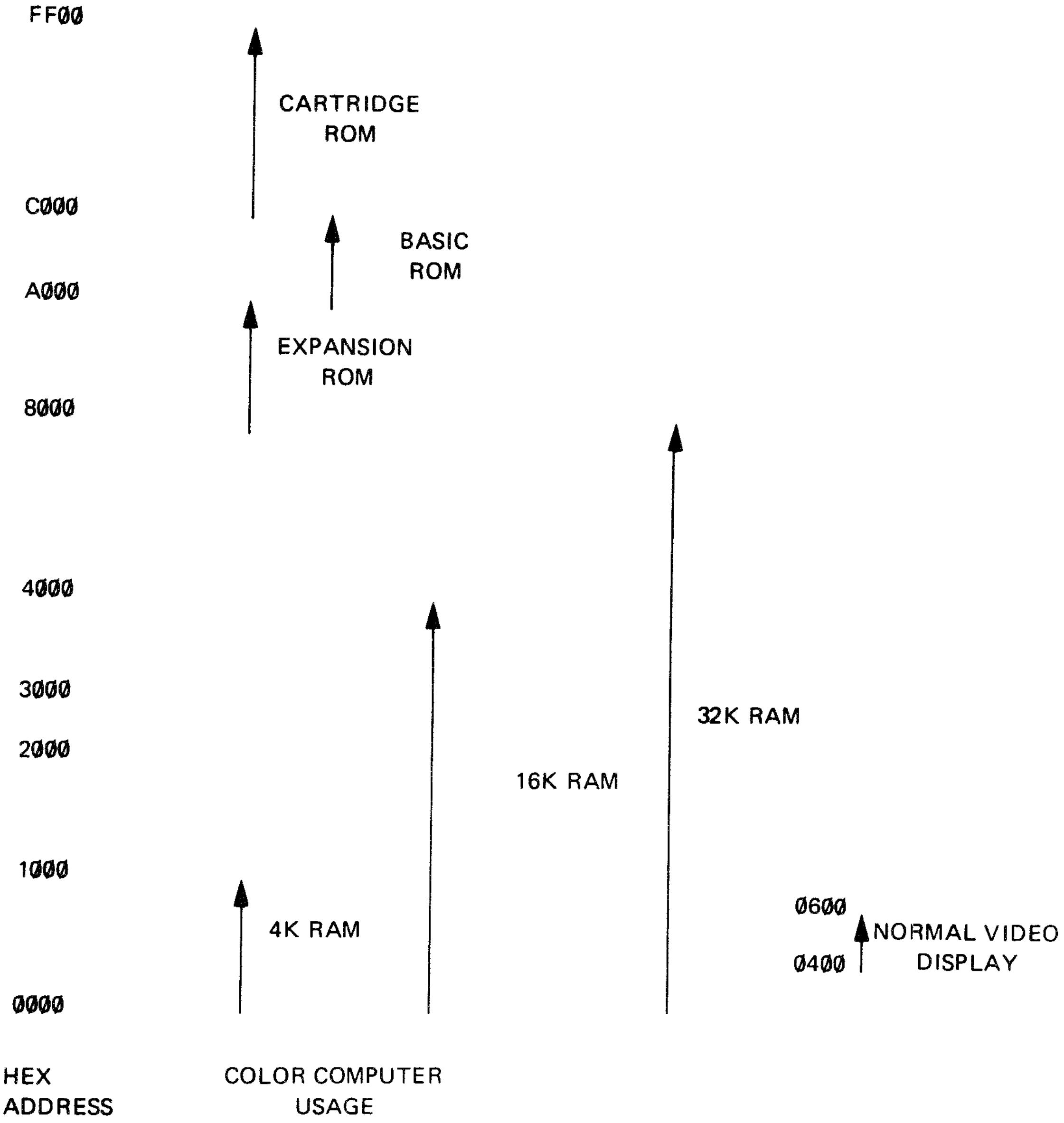
* How many "K" is the address range of the Color Computer?

64K.

* Where in the memory map is read-write (random-access) memory, or RAM, in the Color Computer on a 16K machine?

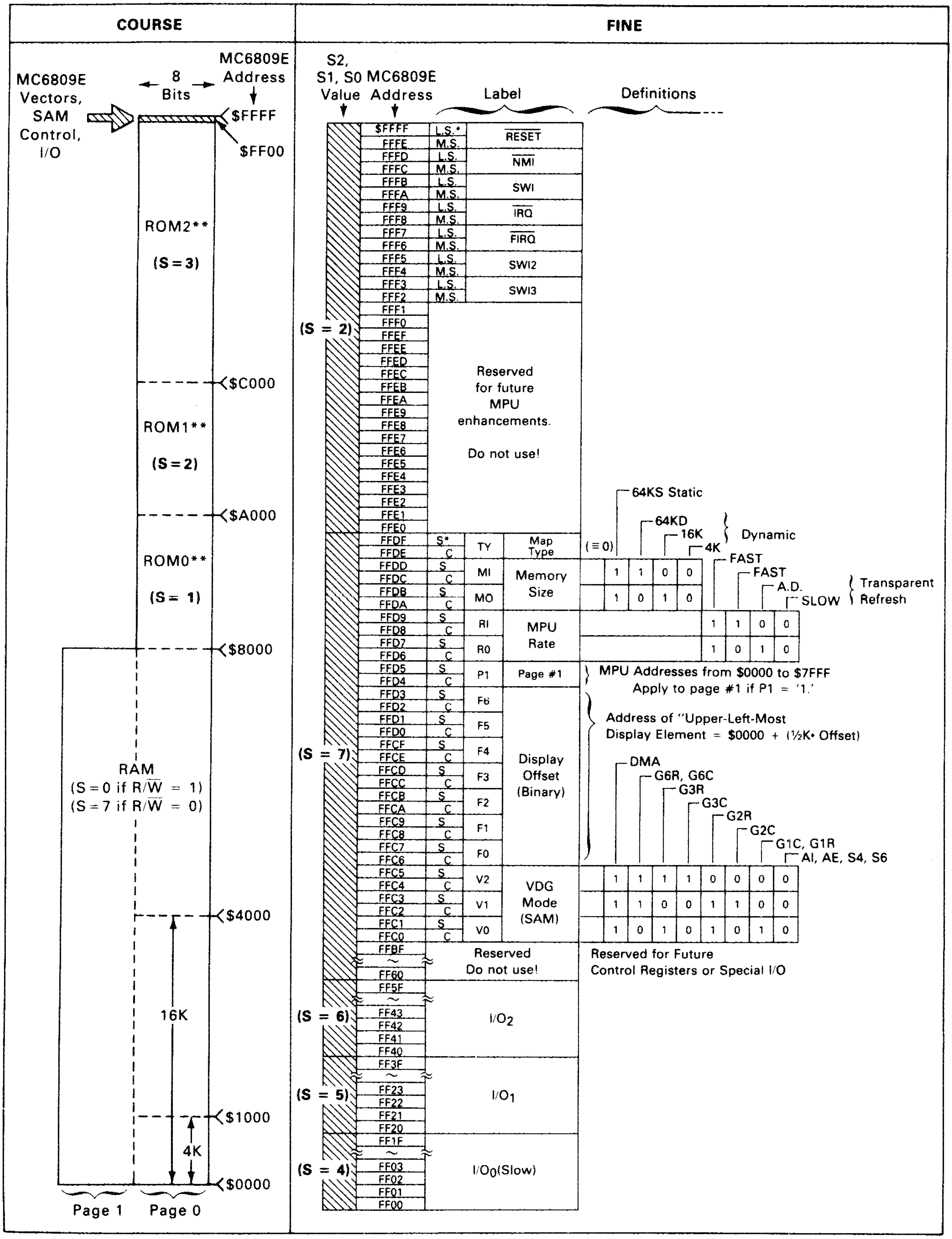
RAM is located from $0000 to $3FFF.

COLOR COMPUTER MEMORY MAP



FFFF OR BFFF	27	RESET VECTOR LSB
FFFE OR BFFE	A0	RESET VECTOR MSB
FFFD OR BFFD	09	NMI VECTOR LSB
FFFC OR BFFC	01	NMI VECTOR MSB
FFFB OR BFFB	06	SWI1 VECTOR LSB
FFFA OR BFFA	01	SWI1 VECTOR MSB
FFF9 OR BFF9	0C	IRQ VECTOR LSB
FFF8 OR BFF8	01	IRQ VECTOR MSB
FFF7 OR BFF7	0F	FIRQ VECTOR LSB
FFF6 OR BFF6	01	FIRQ VECTOR MSB
FFF5 OR BFF5	03	SWI2 VECTOR LSB
FFF4 OR BFF4	01	SWI2 VECTOR MSB
FFF3 OR BFF3	00	SWI3 VECTOR LSB
FFF2 OR BFF2	01	SWI3 VECTOR MSB

MEMORY MAP



*Note:
M.S. = Most Significant
L.S. = Least Significant

S = Set Bit
C = Clear Bit
(All bits are cleared when SAM is reset.)

S = Device Select value = 4 x S2 + 2 x S1 + 1 x S0

**May also be RAM

COLOR COMPUTER MEMORY MAP (cont'd)

FFØØ – FFØ3 PIA U8

FFØØ

BIT Ø = KEYBOARD ROW 1 and right joystick switch
BIT 1 = KEYBOARD ROW 2 and left joystick switch
BIT 2 = KEYBOARD ROW 3
BIT 3 = KEYBOARD ROW 4
BIT 4 = KEYBOARD ROW 5
BIT 5 = KEYBOARD ROW 6
BIT 6 = KEYBOARD ROW 7
BIT 7 = JOYSTICK COMPARISON INPUT

FFØ1

BIT Ø
BIT 1
Control of the Horizontal sync clock (63.5 microseconds) Interrupt Input
Ø=IRQ* to CPU Disabled
1=IRQ* to CPU Enabled
Ø=Flag set on the falling edge of HS
1=Flag set on the rising edge of HS
BIT 2 = Normally 1: Ø=Changes FFØØ to the data direction register
BIT 3 = SEL 1: LSB of the two analog MUX select lines
BIT 4 = 1 Always
BIT 5 = 1 Always
BIT 6 Not Used
BIT 7 = Horizontal sync interrupt flag

FFØ2

BIT Ø= KEYBOARD COLUMN 1
BIT 1= KEYBOARD COLUMN 2
BIT 2= KEYBOARD COLUMN 3
BIT 3= KEYBOARD COLUMN 4
BIT 4= KEYBOARD COLUMN 5
BIT 5= KEYBOARD COLUMN 6
BIT 6= KEYBOARD COLUMN 7
BIT 7= KEYBOARD COLUMN 8

FFØ3

BIT Ø
BIT 1
Control of the field sync clock 16.667 Ms Interrupt Input
Ø=IRQ* to CPU Disabled
1=IRQ* to CPU Enabled
Ø= sets flag on falling edge FS
1= sets flag on rising edge FS
BIT 2 = NORMALLY 1: Ø= changes FF02 to the data direction register
BIT 3 = SEL 2: MSB of the two analog MUX select lines
BIT 4 = 1 Always
BIT 5 = 1 Always
BIT 6 Not Used
BIT 7 = Field sync interrupt flag

COLOR COMPUTER MEMORY MAP (Cont'd)

FF2Ø – FF23   PIA   U4

FF2Ø

BIT Ø = CASSETTE DATA INPUT
BIT 1 = RS-232 DATA OUTPUT
BIT 2 = 6 BIT D/A LSB
BIT 3 = 6 BIT D/A
BIT 4 = 6 BIT D/A
BIT 5 = 6 BIT D/A
BIT 6 = 6 BIT D/A
BIT 7 = 6 BIT D/A MSB

FF21

BIT Ø
BIT 1
Control of the CD RS-232 status Input

Ø = FIRQ* to CPU Disabled
1 = FIRQ* to CPU Enabled

Ø = set flag on falling edge CD
1 = set flag on rising edge CD

BIT 2 = Normally 1:   Ø = changes FF2Ø to the data direction register
BIT 3 = Cassette Motor Control: Ø = OFF   1 = ON
BIT 4 = 1 Always
BIT 5 = 1 Always
BIT 6 Not Used
BIT 7 = CD Interrupt Flag

FF22

BIT Ø = RS-232 DATA INPUT		
BIT 1 = SINGLE BIT SOUND OUTPUT		
BIT 2 = RAM SIZE INPUT	LOW = 4K	HIGH = 16K
BIT 3 = VDG CONTROL OUTPUT		CSS
BIT 4 = VDG CONTROL OUTPUT		GMØ & $\overline{\text{INT}}$/EXT
BIT 5 = VDG CONTROL OUTPUT		GM1
BIT 6 = VDG CONTROL OUTPUT		GM2
BIT 7 = VDG CONTROL OUTPUT		$\overline{\text{A}}$/G

FF23

BIT Ø
BIT 1
Control of the Cartridge Interrupt Input

Ø = FIRQ* to CPU Disabled
1 = FIRQ* to CPU Enabled

Ø = sets flag on falling edge CART*
1 = sets flag on rising edge CART*

BIT 2 = Normally 1:   Ø = changes FF22 to the data direction register
BIT 3 = Six BIT Sound Enable
BIT 4 = 1 Always
BIT 5 = 1 Always
BIT 6 = Not Used
BIT 7 = Cartridge Interrupt Flag

* What is the range of RAM on a 32K machine?

RAM is located from $0000 to $7FFF.

* The Color Computer's operating language is located in what kind of memory?

Read-only memory, or ROM.

* The Color Computer's operating language is in two linked parts. What are they called?

Color BASIC and Extended Color BASIC.

* Where is Color BASIC in the memory map?

From $A000 to $BFFF.

* Where is Extended Color BASIC in the memory map?

From $8000 to $9FFF.

* What is located from $C000 to $FEFF on an off-the-shelf the Color Computer?

Nothing; the space is reserved.

* What is the space from $C000 to $FEFF reserved for?

For plug-in cartridge ROM, also called ROMpacks or program cartridges.

* What is located in the memory map from $FF00 to $FFFF?

MC6809E vectors, SAM control and I/O.

* What is the SAM?

The Synchronous Address Multiplexer.

* What does I/O mean?

I/O means input/output.

the addresses from **$C000** to **$FEFF** are decoded for use by the ROMpack. Notice I said **$C000** to **$FEFF**.

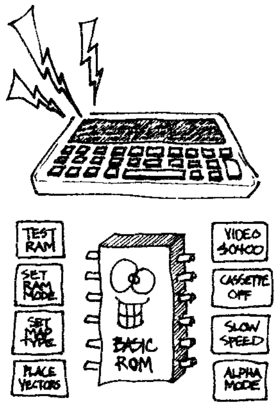
There is a block of memory from **$FF00** to **$FFFF** that is very special. In the back of your documentation, find the data booklet entitled MC6883, and turn to page 17. Here is a table marked Memory Map Type #0. Look at the left half, marked "course" (meaning a course breakdown of the memory map). You can see the layout of the address blocks I've described so far, and at the very top, a small block called "MC6809E vectors, SAM control, I/O". A blowup of this tiny block is shown on the right side of the figure, marked "fine".

Before looking at the detailed map, I want to tell you about the SAM. You may have heard this term before; I was mystified the first time I encountered it. You're holding the SAM's data booklet now. SAM means "Synchonous Address Multiplexer", a mouthful that breaks down to three simple concepts. It's synchronous because it is completely synchronized with the operation of the 6809 processor itself, as well as with the video display, memory, and so forth. It deals with addresses, its main task. And it is a multiplexer because it is the traffic cop, sending the proper addresses to the correct memory blocks. If that doesn't interest you, then let me say that, all because of the SAM, your Color Computer is a 96K computer.

On to the map. Start from the bottom of the "fine" map. You'll see three blocks from **$FF00** to **$FF5F** marked I/O, meaning input/output. At these addresses — and more on this later — are found the keyboard, joystick inputs, cassette input and output, printer input and output, cassette motor control, various high-resolution color modes, and other computer control information. Also, the plug-in disk pack and different peripheral devices use these input/output addresses. That's a lot to know about, but the many capabilities of the Color Computer are found in these input/output blocks.

Next up on the map is a group of addresses ($FF60 to **$FFBF**) which are not defined yet by the manufacturer of the Synchronous Address Multiplexer, the SAM.

Up from there at address **$FFC0** begin a unique series of SAM registers. There was a standard joke among memory engineers that they'd developed the read/write memory — where information could be stored and retrieved — and the read-only memory, where information was permanently fixed and could only be retrieved — but hadn't developed the write-only memory, where information could be stored but couldn't be retrieved. Well, the SAM's got it. Actually, these memory locations are called write-only registers, and their job is to perform computer control functions. Your program keeps track of what's been done, since these are infrequently accessed items. Interestingly, what data you store in these registers is completely irrelevant . . . all that matters is that you store *something* there.



Included in the write-only registers are six addresses to set and reset the eight graphics display modes; 14 addresses to define the area of memory to be displayed on the screen; and 12 addresses to define which 32K block or RAM will be used in a 96K machine, what processor speed will be used, how much memory is available, and which memory map arrangement will be used.

All of these registers are set up by Color BASIC when the power is turned on, but you *can* change them at any time. I've got a little BASIC program to play around with the video graphics modes. Get it loaded, and then I'll tell you about it.

Program #14, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
1 REM * USING ALL VIDEO MODES
2 REM * PORT $FF22 SELECTS VIDEO
3 FORX=8TO128STEP8
4 POKE&HFF22,(X OR 4)
5 REM * ADDRESSES TO CLEAR MODE
6 C1=&HFFC0:C2=&HFFC2:C3=&HFFC4
7 REM * ADDRESSES TO SET MODE
8 S1=&HFFC1:S2=&HFFC3:S3=&HFFC5
9 REM ************************
10 REM * BEGIN CHANGING MODES *
11 REM ************************
12 POKEC1,0:POKEC2,0:POKEC3,0
13 GOSUB30
14 POKES1,0:POKEC2,0:POKEC3,0
15 GOSUB30
16 POKEC1,0:POKES2,0:POKEC3,0
17 GOSUB30
18 POKES1,0:POKES2,0:POKEC3,0
19 GOSUB30
20 POKEC1,0:POKEC2,0:POKES3,0
21 GOSUB30
22 POKES1,0:POKEC2,0:POKES3,0
23 GOSUB30
24 POKEC1,0:POKES2,0:POKES3,0
25 GOSUB30
26 POKES1,0:POKES2,0:POKES3,0
27 GOSUB30
28 NEXT
29 END
30 FORN=1TO300:NEXT
31 RETURN
```

LIST lines 1 to 4. Line 2 says "Port **$FF22** selects video". What's port **$FF22**? This is another bit of jargon. The electronic circuits which allow the 6809 processor in the Color Computer to use its keyboard, cassette, video, etc., are called "peripheral interface adaptors". There are two peripheral interface adaptors, or PIAs, built into the

* What are the I/O (input/output) addreses?

$FF00 to $FF5F.

* Name some of the input/output devices located at the I/O addresses from $FF00 to $FF5F.

Keyboard, joystick inputs, cassette input and output, printer input and output, cassette motor control, sound output, high-resolution color mode control, and other computer control information.

* What does SAM mean?

Synchronous Address Multiplexer.

* The SAM contains memory locations reserved for control; what kind of registers are these?

Write-only registers.

* Name some of the purposes of these write-only registers.

To set or reset eight graphics display modes; to define the area of memory to be displayed on the screen; to define which 32K block of RAM will be used in a 96K machine; to determine what processor speed will be used; to indicate how much memory is available; to specify which memory map arrangement is to be used.

* What does SAM mean?

Synchronous Address Multiplexer.

* What does PIA mean?

Peripheral Interface Adaptor.

* What is the proper term for "setting up" a computer device.

Configuring.

* What is the term for memory addresses that open to the outside world?

Ports.

* How many ports does the Color Computer have?

Four.

* What are the Color Computer port addresses?

$FF00, $FF02, $FF20 and $FF22.

* What is the term for "setting up" a computer device?

Configuring.

* What four PIA addresses configure the four port addresses?

$FF01, $FF03, $FF21 and $FF23.

* What are the port addresses configured as?

Input or output.

* How does the processor send or receive information (input or output information) with respect to the outside world?

By loading or storing data at the port memory addresses.

* At port $FF22, what is the purpose of bit 3?

To choose one of two color sets.

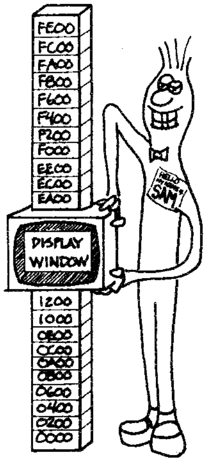
computer, and each is given four memory addresses. The first PIA, for example, uses addresses **$FF00** through **$FF03**. These addresses — and I won't spend a lot of time on this right now — have two functions. **$FF01** and **$FF03** — the odd-numbered registers — are called "control registers", and are used for setting up (the word for that is "configuring") the PIAs. The even-numbered addresses **$FF00** and **$FF02** open to the outside world. They are called "ports".

What this means is that ports **$FF00** and **$FF02** of the first PIA are configured by addresses **$FF01** and **$FF03**. They are configured as input or output. That way the processor can receive or send information to the outside world when it executes machine-language instructions which load or store data at those memory addresses.

In this example, the processor can address the second PIA at **$FF20**, **$FF21**, **$FF22**, and **$FF23**. The PIA configuration using **$FF21** and **$FF23** has already been done at power-up, so that's not your concern for the moment. What you need to know is this: in address **$FF22** are the video graphics modes. One of the two color sets is selected by bit 3; graphics mode zero is turned on or off by bit 4; graphics mode one is turned on or off by bit 5; graphics mode two is turned on or off by bit 6; and the alphanumeric or graphic choice is made by bit 7. So each of the most-signficant 6 bits of address **$FF22** has a different purpose in setting up the video display.

Unless you've spent a lot of time cracking your brains over your BASIC manuals, I probably just dropped another bucket of unknowns in your lap — the graphics modes. It turns out that the Color Computer is a chain of "smart" circuits — the 6809E processor connects to the 6883 synchronous address multiplexer which in turn connects to the 6847 video display generator and the 6821 peripheral interface adaptors. Forget all those numbers. Just dig out your old COLOR BASIC manual — that's the COLOR BASIC manual, not the Extended Color one — and turn to page 256. RUN the program you've got in your computer now, and while it's running, read pages 256 through 266. If you've been spoiled by the Extended Color BASIC graphics modes, then you probably forgot all about these pages in that old Color BASIC manual. So dig in now.

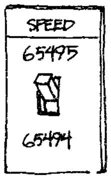
6809
PRESIDENT
CPU
6883
GENERAL
MGR.
SAM
6847
PUBLICITY
MGR.
VDG
6821
PRODUCTION
DEPT.
PIA
6821
CONTROLLER
PIA
BASIC
POLICY
DIVISION
ROM
4164
STOCK
ROOM
RAM



By now I expect that the use of decimal numbers in the Color BASIC manual obscures rather than illuminates how all this works. You'd probably like to take a break, but don't do it yet. While this information is still fresh, I'd like you to RUN once again the program in the computer.

What you see when you run the program are all the possible combinations of alphanumeric and graphic modes that can be created by the combination of the synchronous address multiplexer (that is, the SAM) and the video display generator (that is, the VDG). I've already mentioned about port **$FF22** in the memory map. Just to review, bits 3 through 7 of that byte can be used to select one of two color sets; turn graphic modes one, two and three on or off; and select between alphanumerics and graphics.

The choice of bits you turn on or off at port **$FF22** can then be combined with the SAM's video registers to offer additional possibilities for display. To get at them, though, you have to understand how the SAM's peculiar "write-only" registers work. You still have that BASIC program in place. LIST lines 5 through 8. I've defined six variables here. C1, C2 and C3 mean clear 1, clear 2 and clear 3, and are defined as the three even-numbered addresses **$FFC0**, **$FFC2** and **$FFC4**. S1, S2 and S3 mean set 1, set 2, and set 3, and are defined as the three odd-numbered addresses **$FFC1**, **$FFC3** and **$FFC5**. It turns out that writing to an *address*, no matter what the data stored, either sets or resets a condition within the SAM.



Some of you may have used the high-speed mode on your Color Computer, sometimes called the Vitamin Q poke. You probably wrote it, POKE65495,0 and to get normal speed, POKE 65494,0. When you did that POKE, you were actually executing a Store Accumulator to memory location **$FFD7** for high speed and **$FFD6** for normal speed.

Flip to the SAM data booklet (the booklet marked MC6883), and return to page 17. Locate addresses **$FFC0** through **$FFC5**. These are the video display modes, the VDG modes. At the right of the addresses, the mode combinations are shown in binary. To turn on any of these modes, the binary data has to be expressed as a trio of addresses — either the clear address (the even ones) or the set addresses (the odd ones).

Likewise, locate addresses **$FFD6** and **$FFD7**. They are part of a group of addresses that affect speed of the computer. At power up, your computer is in the "slow" mode. By writing to **$FFD7**, you set the "A.D.", or address dependent, mode. In that mode, your BASIC ROM zipped along at double speed, and your RAM just stayed the way it was. Had you poked **$FFD9**, you would have gone into the "fast RAM" mode, losing both the video display and the refresh your memory needs to keep its information.

You don't need the BASIC program now, so <BREAK> out of it if it's still running. I want to show you what happens when you use the "fast RAM" mode at address **$FFD9**.

* What is the purpose of bits 4 through 6?

To select among the graphics modes.

* What is the purpose of bit 7?

To select either alphanumerics or graphics.

* What does PIA mean?

Peripheral Interface Adaptor.

* What is the term for memory addresses which open to the outside world?

Ports.

* What does SAM mean?

Synchronous Address Multiplexer.

* What sets or resets a condition within the SAM?

Writing to a SAM address (register).

* What sets or resets video display modes?

Writing to the SAM video display addresses (registers).

* What are the SAM video display registers?

$FFC0 through $FFC5.

* What changes the computer's processing speed?

Writing to the SAM clock rate addresses (registers).

* What are the SAM clock rate registers?

$FFD6 through $FFD9.

Display offset

* What is the normal speed of the Color Computer ?

.89 MHz (894,886 pulses per second).

* Where is the normal video display screen on the Color Computer (in decimal and hex)?

At 1024 ($0400 hex).

* What does VDG mean?

Video Display Generator.

* What determines the screen being displayed?

The SAM display offset addresses (registers).

* What are the display offset registers?

$FFC6 through $FFD3.

* How many bits of the 16-bit address are selected by the display offset registers?

Seven.

* How many combinations of 7 bits are possible?

128.

* How many display screens are possible by using the SAM's display offset addressing technique?

128.

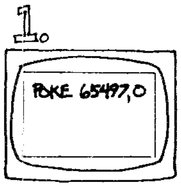
* How do you create a display offset address?

By writing to the SAM display offset registers.

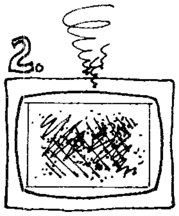
* How do you create the offset address 0000000?

By writing to all the even-numbered SAM display offset addresses (registers).

$FFD9 is 65497 decimal. So type POKE 65497,0 and hit <ENTER>. POKE 65497,0.



Screen freaked out, right? Hit your Reset button on the back right to get back your screen. Whether or not the program is still intact depends, for technical reasons, on whether you have a 16K, 32K or 64K machine.



There's some more to find out about the SAM, so I have another program for you.

Program #15, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/0 error occurs, rewind to the start and try again. For severe loading problems, see the Appendix.

```
1 CLS
2 PRINT" REDIRECTING THE VIDEO DISPLAY":PRINT
3 C0=&HFFC6:C1=&HFFC8:C2=&HFFCA:C3=&HFFCC:C4=&HFFCE:C5=&HFFD0:C6
=&HFFD2
4 S0=&HFFC7:S1=&HFFC9:S2=&HFFCB:S3=&HFFCD:S4=&HFFCF:S5=&HFFD1:S6
=&HFFD3
5 INPUT"THE NORMAL SCREEN IS LOCATED AT $0400 TO $05FF.  THE SAM
 ALLOWS THE SCREEN TO POINT TO ANY PLACEIN MEMORY.  THERE ARE 12
8 SCREENSIN ALL.  ENTER A NUMBER FROM 0 TO 127 TO DISPLAY A SCRE
EN";A$
6 A=VAL(A$):IFA(0ORA)127THENCLS:GOTO5
7 B6=FIX(A/64)
8 B5=FIX((A-(B6*64))/32)
9 B4=FIX((A-(B6*64)-(B5*32))/16)
10 B3=FIX((A-(B6*64)-(B5*32)-(B4*16))/8)
11 B2=FIX((A-(B6*64)-(B5*32)-(B4*16)-(B3*8))/4)
12 B1=FIX((A-(B6*64)-(B5*32)-(B4*16)-(B3*8)-(B2*4))/2)
13 B0=FIX(A-(B6*64)-(B5*32)-(B4*16)-(B3*8)-(B2*4)-(B1*2))
14 IFB0=0THENPOKEC0,0ELSEPOKES0,0
15 IFB1=0THENPOKEC1,0ELSEPOKES1,0
16 IFB2=0THENPOKEC2,0ELSEPOKES2,0
17 IFB3=0THENPOKEC3,0ELSEPOKES3,0
18 IFB4=0THENPOKEC4,0ELSEPOKES4,0
19 IFB5=0THENPOKEC5,0ELSEPOKES5,0
20 IFB6=0THENPOKEC6,0ELSEPOKES6,0
21 FORN=1TO2000:NEXT
22 GOTO1
```

The object of this program is to manipulate the SAM "display offset" registers. This nifty technique makes it possible to display 128 entirely different screens of information, each 512 (hex **$200**) bytes long.

RUN this program, and enter 2 in response to the prompt. There is a pause, and the cursor is back. Of the 128 possible screens, the one you normally look at the screen #2. Now enter 0. Aha. A screen full of garbage and wiggly characters appears before you. Try that again; enter 0. Screen #0 is what you see, and screen #0 reveals pages **$00** and **$01** of your memory. Remember the Direct Page register? The Color Computer's BASIC sets the DP register to **$00**, meaning what you're seeing is all the down-and-dirty work BASIC does to count, calculate, delay, and so on.

Now I'll show you what's happening there. Turn once again to page 17 of the SAM data booklet, where the detailed memory map is shown. Addresses **$FFC6** to **$FFD3** are called a display offset value, and a strange formula is given, reading "Address of upper-left-most display element = **$0000** + (1/2K * offset)". Obscurity won't triumph, I'll tell you. What this means is that you can display any area of memory directly on the screen, in even 512-byte blocks.

Addresses **$FFC6** to **$FFD3** are those write-only SAM registers again, used here to create the most-significant 7 bits of an address. Writing to the even-numbered registers starting with **$FFC6** clears bits to zero; writing to the odd-numbered registers sets bits to one. So if you store information in all the even-numbered registers, you create the binary number **0000 000** . . . 7 bits long. If you store information in all the even-numbered registers except **$FFC8**, but store information in the odd-numbered register **$FFC9**, and you create the binary number **0000 010**. Those are the most significant seven bits of addresses **0000 0100 0000 0000** through **0000 0101 1111 1111**. Those binary addresses translate into **$0400** to **$05FF** — the address of the normal video screen.

That's all I have for you this time. I would like you to LIST this program, and get an idea of how to manipulate the addresses. Take a break, play with the program, and then come back for the next session; you'll be translating these concepts into an assembly-language subroutine.

To review: the Color Computer is more than a smart 6809 processor, and so effective programming on this machine requires knowing the rest of the smart devices inside it. These devices include a video display generator (VDG) to provide alphanumeric and graphic displays in several colors; a synchronous address multiplexer (SAM) to coordinate and synchronize events involving input/output, display, and memory addressing; and two peripheral interface adapters (PIAs) to provide input and output for keyboard, cassette, printer, video, sound, and other computer control functions.

These smart devices all have control signals which are connected into the memory map and given specific addresses. By storing information at these addresses, your programs can have control of all the computer's functions.

Please review this lesson, and familiarize yourself with the programming aspects presented in the data booklets for the MC6883 SAM, the MC6847 VDG, and the MC6821 PIA.

After you've finished trying out and examining this program, there's one more at the end of the lesson. Load, LIST and RUN it. It should give you some ideas.

* What are the even-numbered display offset registers?

$FFC6, $FFC8, $FFCA, $FFCC, $FFCE, $FFD0 and $FFD2.

* How do you create the display offset address 1111111?

By writing to all the odd-numbered SAM display offset registers.

* What are the odd-numbered SAM display offset registers?

$FFC7, $FFC9, $FFCB, $FFCD, $FFCF, $FFD1 and $FFD3.

* How do you create the diplay offset address 0110110?

By writing to a combination of odd and even addresses: $FFC6, $FFC9, $FFCB, $FFCC, $FFCF, $FFD1 and $FFD2.

* What is the address of the first byte displayed on the screen with the offset address 0110110?

The first byte (the upper-left-most byte) displayed is $6C00.

* What does VDG mean?

Video Display Generator.

* What does PIA mean?

Peripheral Interface Adaptor.

* What does SAM mean?

Synchronous Address Multiplexer.

* What is located in the lower half of the Color Computer's memory map (from $0000 to $7FFF)?

Read/write memory (random-access memory), or RAM.

Program #16

* What is located from $8ØØØ to $9FFF?

Extended Color BASIC in read-only memory (ROM).

* What is located from $AØØØ to $BFFF?

Color BASIC in read-only memory (ROM).

* What is located from $CØØØ to $FEFF?

Nothing unless a cartridge read-only memory (ROM) pack is plugged in.

* What is located from $FFØØ to $FFFF?

MC6809E vectors, SAM control, and I/O.

* What do you call the description of how the computer's designers have arranged its memory?

The memory map.

Program #16, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
1 CLS:CLEAR2ØØ.16292:PCLEAR4:X=&HØ4ØØ
2 GOSUB43:GOSUB86:GOSUB99
3 GOSUB55:GOSUB86:GOSUB99
4 GOSUB65:GOSUB86:GOSUB94:GOSUB99
5 GOSUB76:GOSUB86:GOSUB94:GOSUB99
6 GOSUB86:GOSUB99
7 GOSUB94:GOSUB99
8 GOSUB97:GOSUB99
9 DATA B7,FF,C7,B7,FF,C9,B7,FF,CA,B7,FF,CC,39
1Ø DATA B7,FF,C6,B7,FF,C8,B7,FF,CB,B7,FF,CC,39
11 DATA B7,FF,C7,B7,FF,C8,B7,FF,CB,B7,FF,CC,39
12 DATA B7,FF,C6,B7,FF,C9,B7,FF,CB,B7,FF,CC,39
13 DATA B7,FF,C7,B7,FF,C9,B7,FF,CB,B7,FF,CC,39
14 DATA B7,FF,C6,B7,FF,C8,B7,FF,CA,B7,FF,CD,39
15 DATA B7,FF,C7,B7,FF,C8,B7,FF,CA,B7,FF,CD,39
16 FORX=16293 TO 16383:READA$:A=VAL("&H"+A$):POKEX,A:NEXT
17 DEFUSR1=16293
18 DEFUSR2=163Ø6
19 DEFUSR3=16319
2Ø DEFUSR4=16332
21 DEFUSR5=16345
22 DEFUSR6=16358
23 DEFUSR7=16371
24 FORA=1TO4Ø
25 GOSUB1ØØ:GOSUB1Ø8
26 GOSUB1Ø1:GOSUB1Ø8
27 GOSUB1Ø2:GOSUB1Ø8
28 GOSUB1Ø3:GOSUB1Ø8
29 NEXT
3Ø FORA=1TO2Ø
31 GOSUB1Ø3:GOSUB1Ø8
32 GOSUB1Ø4:GOSUB1Ø8
33 NEXT
34 FORA=1TO2Ø
35 GOSUB1Ø4:GOSUB1Ø8
36 GOSUB1Ø5:GOSUB1Ø8
37 NEXT
38 FORA=1TO2Ø
39 GOSUB1Ø5:GOSUB1Ø8
4Ø GOSUB1Ø6:GOSUB1Ø8
41 NEXT
42 GOTO24
43 REM
44 PRINT@Ø,"*   *   *   *   *   *   *   *   ";
45 PRINTSTRING$(31,32)"*";
46 PRINT:PRINT:PRINT"*"
47 PRINTSTRING$(31,32)"*";
48 PRINT:PRINT:PRINT"*"
49 PRINTSTRING$(31,32)"*";
5Ø PRINT:PRINT:PRINT"*"
51 PRINTSTRING$(31,32)"*";
52 PRINTSTRING$(32,32);
53 PRINT" *   *   *   *   *   *   *   * ";
54 RETURN
55 PRINT@Ø," *   *   *   *   *   *   *   *"
56 PRINT:PRINTSTRING$(31,32)"*";
57 PRINT"*":PRINT:PRINT
58 PRINTSTRING$(31,32)"*";
59 PRINT"*":PRINT:PRINT
6Ø PRINTSTRING$(31,32)"*";
```

```
61 PRINT"*":PRINT:PRINT
62 PRINTSTRING$(31,32)"*";
63 PRINT"*   *   *   *   *   *   *   *  ";
64 RETURN
65 PRINT@0,"  *   *   *   *   *   *   *   *"
66 PRINT:PRINT"*"
67 PRINTSTRING$(31,32)"*";
68 PRINT:PRINT:PRINT"*"
69 PRINTSTRING$(31,32)"*";
70 PRINT:PRINT:PRINT"*"
71 PRINTSTRING$(31,32)"*";
72 PRINT:PRINT:PRINT"*"
73 PRINT"   *   *   *   *   *   *   *   ";
74 POKE1535,106
75 RETURN
76 PRINT@0,"   *   *   *   *   *   *   *   *";
77 PRINT"*":PRINT:PRINT
78 PRINTSTRING$(31,32)"*";
79 PRINT"*":PRINT:PRINT
80 PRINTSTRING$(31,32)"*";
81 PRINT"*":PRINT:PRINT
82 PRINTSTRING$(31,32)"*";
83 PRINT"*":PRINT
84 PRINT"  *   *   *   *   *   *   *   *";:POKE1535,96
85 RETURN
86 PRINT@68,"the message can be made";
87 PRINT@132," TO FLICKER AND FLASH";
88 PRINT@196,STRING$(23,191);
89 PRINT@260,"       GREEN <<";
90 PRINT@292,"       MOUNTAIN";
91 PRINT@324,"       >> MICRO";
92 PRINT@388,STRING$(23,191);
93 RETURN
94 PRINT@196,STRING$(23,207);
95 PRINT@388,STRING$(23,207);
96 RETURN
97 PRINT@132," TO flicker AND flash";
98 RETURN
99 X=X+&H200:Y=&H0400:FORQ=X TO X+512:POKEQ,PEEK(Y):Y=Y+1:NEXT:R
ETURN
100 M=USR1(0):RETURN
101 M=USR2(0):RETURN
102 M=USR3(0):RETURN
103 M=USR4(0):RETURN
104 M=USR5(0):RETURN
105 M=USR6(0):RETURN
106 M=USR7(0):RETURN
107 FORN=1TO500:NEXT:RETURN
108 FORN=1TO4:NEXT:RETURN
```