OSI

11679 HAYDEN STREET
HIRAM, OHIO 44234

OSI MODEL 300 MANUAL

VERSION II

OSI MODEL 300 COMPUTER TRAINER

THE MODEL 300 IS A COMPLETELY ASSEMBLED AND TESTED, READY TO USE, COMPUTER DESIGNED TO "BOOTSTRAP" THE STUDENT, HOBBYIST, AND ENGINEER INTO THE MICROPROCESSOR WORLD. THE UNIT IS BUILT AROUND THE MOS 6502 8 BIT MICROPROCESSOR AND USES A 128 WORD RAM. ITS CONTROLS INCLUDE 8 DATA SWITCHES, 7 ADDRESS SWITCHES, MEMORY LOAD, PROCESSOR RESET, PROCESSOR RUN, INTERRUPT, AND MEMORY WRITE PROTECT. THE DISPLAYS INDICATE DATA, ADDRESS, AND PROGRAM EXECUTION. TWO INPUT LINES AND ONE OUTPUT LATCH ARE PROVIDED FOR I/O PROGRAMMING. THE MODEL 300 COMES COMPLETE WITH A LABORATORY MANUAL WITH 20 EXPERIMENTS STARTING WITH SIMPLY LOADING AND READING MEMORY AND FINISHING WITH OUTPUTING TO A TELETYPE.

MODEL 300 COMPUTER TRAINER COMPLETELY ASSEMBLED WITH LAB MANUAL (REQUIRES +5VDC AT 500MA)..... $ 99.00

THE 6000 SERIES COMPUTER FAMILY (SUPERBOARD)

A COMPLETE MINICOMPUTER PC BOARD (DOUBLE SIDED EPOXY) WHICH ACCEPTS ANY 6000 SERIES PROCESSOR, SYSTEM CLOCK, 2- 1702 TYPE ROMS, 1K X 8 RAM (2102 TYPE), 1 PIA, 1ACIA, CURRENT LOOP AND PARALLEL INTERFACES AND HAS BUS EXPANSION CAPABILITIES. EACH SUPERBOARD COMES COMPLETE WITH DOCUMENTATION.

SUPERBOARD	BARE WITH MANUAL	29.00
6800	MICROPROCESSOR AND SUPERBOARD	69.00
6501	MICROPROCESSOR AND SUPERBOARD	49.00
6502	MICROPROCESSOR AND SUPERBOARD FEATURES INTERNAL CLOCK	54.00

ALSO AVAILABLE :
- ALL SYSTEM SUPPORT PARTS
- RAM - ROM MEMORY EXPANDER BOARD
- SUPERI/O BOARD CONTAINING CASSETTE INTERFACE; X, Y DISPLAY AND A/D CONVERTER.
- VIDEO GRAPHIC BOARD

COMING SOON:
- FIRMWARE BASIC BOARD (USES ROM AND CALCULATOR CHIP)

CALL (216) 653-6484 OR WRITE TODAY FOR OUR COMPLETE INFORMATION PACKAGE.

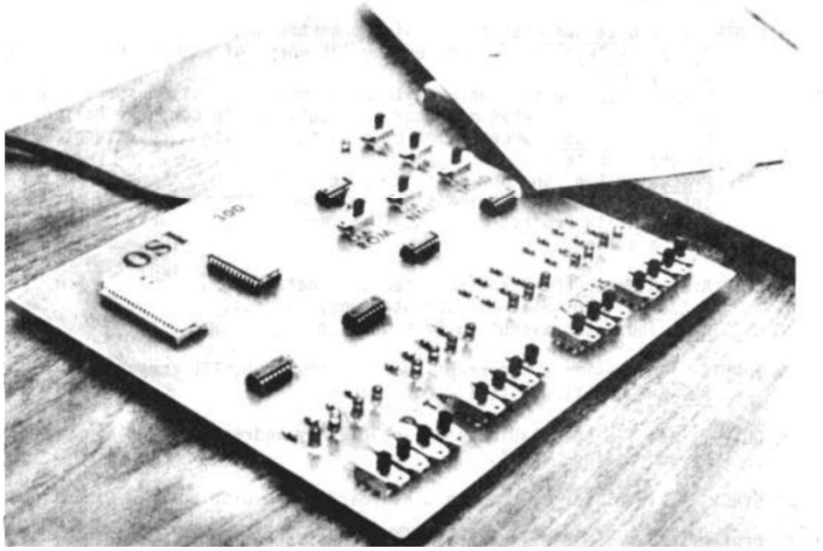
OSI

OHIO SCIENTIFIC INSTRUMENTS

P.O. BOX 374, HUDSON, OHIO 44236

MODEL 300 COMPUTER TRAINER

A COMPLETE, SELF CONTAINED COMPUTER ON A SINGLE PC BOARD



The Model 300 is a completely assembled and tested, ready to use, computer designed to "bootstrap" the student, hobbyist, and engineer into the micro-computer world. The unit comes complete with a 20 experiment lab manual written for use in a college physics, electronics, or computer course. Since the manual assumes no previous knowledge of computers or digital electronics, it is also ideal for self-teaching. The first experiment is simply loading and reading memory. The last experiment is interfacing the computer to a teletype.

The unit has 128 words of memory (8 bits wide) and is based on the MOS Technology 6502 microprocessor which has 55 basic instructions with over 145 variations. The processor does binary, twos complement, and binary coded decimal arithmetic. It also features interrupt capability and has 13 memory addressing modes. The lab manual covers virtually all instruction types and programming procedures.

OSI OHIO SCIENTIFIC INSTRUMENTS

11679 HAYDEN STREET, HIRAM, OHIO 44234

The unit also has a sync line for easy scoping of circuit points. Two input lines and one output latch are provided for I/O programming. The output latch can be connected to an audio amplifier to produce tones and even "tunes."

SPECIFICATIONS

CONTROLS: 8 Data Switches for loading programs and data.
7 Address Switches for addressing 128 words of memory and examining memory contents.
Memory Load deposits data switches at memory location specified by address switches when momentarily actuated with computer halted.
Processor Reset forces reset of computer and loads reset vector into program counter.
Processor Run runs and halts computer.
Interrupt forces a non-maskable interrupt when momentarily actuated.
Memory Write Protect converts the memory to a read only memory (ROM) when actuated.

DISPLAYS: 8 Data LEDs-display contents of data bus.
7 Address LEDs-display current memory address.
Run- indicates execution of a program.

INPUTS: IRQ- maskable interrupt request, low true, TTL compatable.
SO- Set Overflow bit, TTL compatable.

OUTPUT: one TTL compatable latch set high by addressing page 2 and set low by addressing page 1.

SYNC: TTL level for triggering a standard scope with op code execution.

PERFORMANCE: The 6502 is capable of operating at 500ns. cycle time (1.5usec. Jump Absolute). The Model 300 runs at approximately 2usec. sycle time but can be vared by the user.

MECHANICAL: 8" X 10" overall. G-10 epoxy board using standard industrial components.

POWER REQUIREMENTS: 5VDC at 500ma maximum. 350ma typical. The unit will operate for approximately 20 hours from 4 alkaline "D" cells.

GUARANTEE: 90 days materials and workmanship.

DELIVERY: 30 to 60 days.

PRICES (F. O. B. HIRAM, OHIO)
MODEL 300 COMPUTER TRAINER COMPLETELY ASSEMBLED AND TESTED WITH LAB MANUAL (requires +5VDC at 500ma) ..... $ 99.00
110VAC Calculator Type Power Supply ADD..... $ 10.00
MODEL 305 SAME AS MODEL 300 BUT LESS 6502 PROCESSOR... $ 74.00
MODEL 315 PLAN.....See Catalog for Details.

300 Manual Version III

OSI
11679 HAYDEN STREET
HIRAM, OHIO 44234

This Update is to clear up some problems related to the use of the Model 300.

Problem 1. ALWAYS bring reset low and then run low to stop the processor. ALWAYS bring reset high then run high, then reset low to start the processor. This is all stated in Experiment 3, but many people do not pay attention to it. If just the run switch is used to start and stop the processor, "garbage" will be loaded in memory and programs will be destroyed.

Problem 2. The OSI supplied Battery Eliminator will operate the trainer on line voltages from 108 to 125 volts. If the line voltage is above or below this, the unit will operate erratically. Secondly, the supply has very little energy storage. Any line transient which causes lights to flicker or dim will certainly cause a memory "lapse." If these problems occur, substitute a regulated +5 volt supply or batteries. The trainer should not be subjected to voltages above 5.5 volts for prolonged periods. This is not a problem when four D or C cells are used because the battery voltage quickly drops to about 5 volts in use. However, if large lantern batteries or wet cells are used, one or two diodes should be placed in series with the battery to drop the voltage to near 5 volts.

Problem 3. Early trainers use the 6810 or 6810L1 128 word memory by AMI and Motorola. These chips have been discontinued by the manufacturers and are being replaced by a "new improved version." We have a very good record over-all on trainer reliability (only 5% of the units are returned for in warranty repair). However, most of the hardware failures to date are 6810 memory failures due to chips dying or developing pattern sensitivities. The 6810 memory reliability can be greatly improved by returning all data switches to the high or 1s position after each load operation.

Problem 4. The reset switch on the trainer is not debounced which means that the programs may be executed several times when it is activated if precautions are not taken. This is not a problem with the experiments in the lab manual since multiple executions yield the same results. If a program such as a simple memory increment is to be run, it should be preceded by the following time delay routine. This provides sufficient time for switch debouncing before the main program is executed.

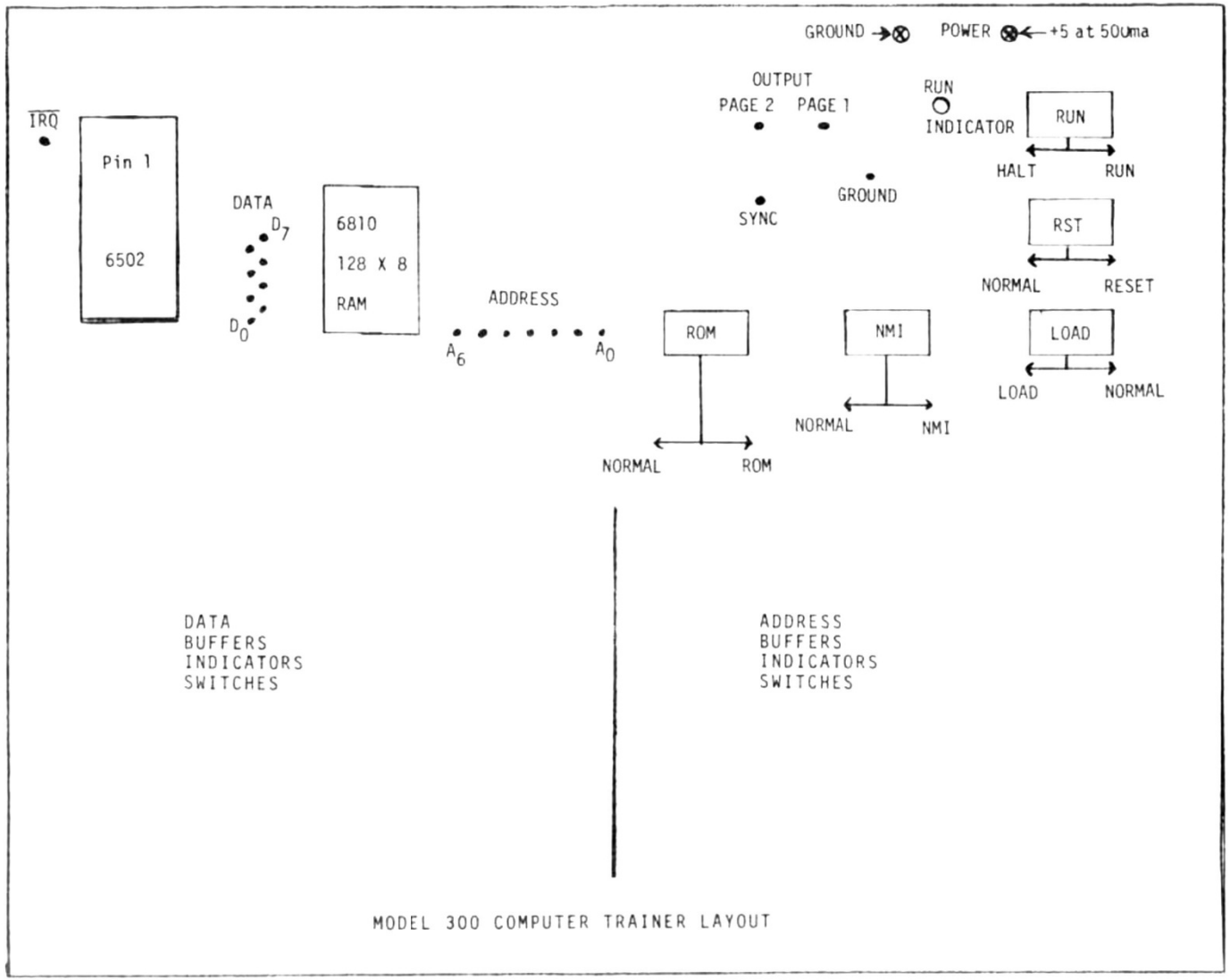
Delay Loop

Op Code	Location	Mnemonic	Note
A2	00	LDX # Oper	Load Counter
00	01	Data	to 255
E8	02	INX	Increment Counter
D0	03	BNE	Loop Back and Increment Again
FD	04	Data	if not Equal to 0.

CONTENTS

1. Binary, Decimal, and Hexadecimal Numbers
2. Reading and Writing in RAM
3. Running a Program on the Model 300 Computer Trainer
4. Binary Addition
5. Adding on the Computer
6. Conditional Branching
7. Adding Signed Numbers on the Computer
8. Subtraction
9. Binary Coded Decimal Arithmetic
10. Logical Operations
11. Double Precision Arithmetic
12. Stack Processing
13. Subroutines
14. Flowcharting
15. Multiplying Two Four Bit Binary Numbers
16. Compare
17. Addressing Modes
18. Interrupts
19. I/O Programming
20. Outputing to a Teletype

Appendix A 650X Specifications Sheets



MODEL 300 COMPUTER TRAINER LAYOUT

Power Connections

The Model 300 Computer Trainer requires +5volts at 500ma maximum, 350ma typical. The unit can be powered from the available battery eliminator accesory , any TTL logic compatable power supply or batteries. The Computer Trainer will work with supply voltages from 6 to 4 volts. Alkaline or Ni-Cad batteries should be used since conventional batteries will last only minutes. The following table gives approximate battery life for fresh alkaline batteries.

4	"D" Cells	16 to 20 hours
4	"C" Cells	4 to 6 hours

Care should be taken to connect the positive lead to the + or red side and negative to the - or black side of the power connections above the run switch and light. The power source should be stable since power "outages" will erase memory and the supply should not produce voltage spikes on turn on or off. In battery operated systems, the battery should be replaced when the run light begins to flicker. This condition occurs at about 4volts.

Guarantee

The Model 300 Computer Trainer is not a consumer product. It is an unprotected P.C. Board and should be treated as such. It can be damaged by improper power supply connections, overvoltages, malfunctioning Input/Output devices or test equipment connected to it, and careless handling.

The unit is mechanically fragile and should never be dropped or have books or other objects piled on it. Electrically it can be damaged by shorting circuit lines together.

OSI guarantees the Model 300 Computer Trainer for 90 days from the date of delivery against defects in materials and workmanship. This guarantee does not cover malfunctions induced by improper mechanical or electrical handling. Malfunctioning units should be sent to OSI with postage prepaid. Out of warranty sevice charges will be quoted for approval before repairs are made.

Scoping

The Model 300 Computer Trainer has pins for scoping the address and data lines and a sync output. Always connect test equipment and I/O device grounds before signal lines to avoid glitches in running programs. The sync line goes low every time an instruction is fetched from memory providing a trigger signal for repetitive programs such as the simple jump to jump program. Times 10 scope probes should be used when scoping circuit points since the address and data buses are somewhat susceptable to stray capacitance.

Manual Updates

If you did not purchase the Computer Trainer directly from OSI, please send us your name and address so that you will receive manual updates.

BINARY, DECIMAL, AND HEXADECIMAL NUMBERS

Introduction:

Expressing numbers in binary, decimal, and hexadecimal form is discussed.

Discussion:

The number system with which most persons are familiar is base 10. In base 10, numbers are expressed using ten digits, 0-9. When dealing with microprocessors, it is essential to also be acquainted with base 2, or binary numbers and with base 16, or hexadecimal numbers. To help simplify the process of converting numbers from base 10 to base 2 and base 16, a conversion table has been provided on the following page.

When using base 10, it is possible to write the number 7 as 0007 where the first three digits indicate that there are zero thousands, zero hundreds, and zero tens while showing that there are seven ones. This can also be thought of as being equal to the following: $10^3X0 + 10^2X0 + 10^1X0 + 10^0X7$. (Mathematical convention defines any number raised to the 0 power as being equal to 1) Each digit's location or "place" in a binary number differs from that of a base 10 or decimal number in that it represents a power of two instead of a power of ten. Remembering this, the binary number 0111 can also be expressed as $2^3X0 + 2^2X1 + 2^1X1 + 2^0X1$. To change the base 2 number number 1011 into a base 10 number, first write the number in exponential form. $2^3X1 + 2^2X0 + 2^1X1 + 2^0X1$. This is equal to 8 + 0 + 2 + 1 or 11.

Numbers can also be converted from base 10 into base 2. In order to do this efficiently, one needs to know the powers of two. It is advised that they be memorized! A partial list is given below.

$2^0 = 1$	$2^6 = 64$
$2^1 = 2$	$2^7 = 128$
$2^2 = 4$	$2^8 = 256$
$2^3 = 8$	$2^9 = 512$
$2^4 = 16$	$2^{10} = 1,024$
$2^5 = 32$	$2^{11} = 2,048$

The base 10 number 255 is converted to binary form to illustrate the procedure given below.

1. Find the largest power of two which is contained in the number. In this case, 128 is the largest.
2. Find the difference between the numbers being converted and the number found above. 255 - 128 = 127
3. Express the number as the sum of the largest power of two contained within the number and the difference between it and the original number. $2^7 + 127$

Table 1. A Conversion Table for Binary Numbers, Hexadecimal Numbers, and Decimal Numbers.

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

4. Find the next largest power of two going into the difference found above. Repeat the procedure above until the entire number is expressed as sums of powers of two. The numbers should be expressed so that the exponents appear in descending order.

$$2^7 + 2^6 + 63$$

$$2^7 + 2^6 + 2^5 + 31$$

$$2^7 + 2^6 + 2^5 + 2^4 + 15$$

$$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 7$$

$$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 3$$

$$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 1$$

$$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$$

5. The number can now be easily transformed into binary format. Recall that each digit of a binary number represents a power of two. Hence, the least significant digit represents 2^0, the next most significant 2^1, then 2^2, then 2^3, and so on. In the number above, each power of two represents one digit. The exponent represetns the place of the digit. Since 2^0 appears above, the least significant digit of the binary number is 1. If 2^0 did not appear above when the number was expressed in terms of powers of two, the least significant digit would be 0. 255 expressed in binary form is 1111 1111.

Another example, converting 20 to binary form, is given below.

20

$2^4 + 4$

$2^4 + 2^2$

Since no 2^0 appears above, the least significant digit is 0. Since no 2^1 appears, the next digit is also 0. Since 2^2 appears, the following digit is 1. Since 2^3 does not appear, the next digit is 0. Since 2^4 appears, the next digit is 1. The binary form of 20, then, is 10100, or 0001 0100.

Base 16 or hexadecimal numbers operate using 16 digits. They are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,A,B, C, D, E, and F. Instead of using two digits after reaching 9 as is done in base 10, hexadecimal numbers use the first six letters of the alphabet and then begin to use two digits. This can be seen by examining the base conversion table (Table 1.). For most microprocessor applications, one will need to know how to change hexadecimal numbers into binary numbers and vice versa, but will rarely need to change hexadecimal numbers to base 10 or vice versa. Table 1 shows the basic pattern that is used to change binary numbers into hexadecimal. It also gives the decimal equivalent for each number. It is highly advantageous to memorize this table.

When converting a binary number into hexadecimal, begin by starting with the right-hand most digit and marking the digits off into blocks of four digits each. For example, the binary number 1100111101 would be marked off as 11'0011'1101. Next, add as many zeros as are needed to the left-hand end of the number so that it contains four binary digits in the last section. The above number is then written as 0011'0011'1101. To begin converting the binary number to hexadecimal (hex), find the hex number in the chart that corresponds to the last four digits of the binary number. 1101 in base 2 equals D in hex. Continue this process by finding the hex equivalents for each of the sets of four binary digits. 0011 equals 3 in hex. 0011 equals 3 in hex. The hex equivalent of 0011'0011'1101' is, therefore, 33D. Each hex digit has simply replaced one set of four binary digits.

Converting the hex numbers into binary format is also quite simple. Merely convert each digit into its binary code. The hex number A47 is easily changed. 7 hex equals 0111 binary. 4 hex is the same as 0100 in binary and A in hex equals 1010 in binary code. A47, when changed to binary notation, is 1010'0100'0111 or 101001000111 (Note that the binary number is much easier to read when every four digits are set apart. Therefore, in this manual, binary numbers will have a space between every four digits, i.e., 1010 0100 0111.).

READING AND WRITING IN RAM

Introduction:

Information is stored and retrieved from RAM.

Discussion:

The 300 series Computer Trainer uses a type of computer memory refered to as Random Access Memory (RAM). With RAM, the user can both read and retrieve information from memory and write or store information in memory. This is different from Read Only Memory (ROM) which is pre-programmed so that the user can only use the information which is already stored.

Before reading and writing in memory, one needs to know something about binary numbers and organization of RAM and the computer.

Modern digital circuitry is capable of determining only if a line is "high" or "low", that is, "on" or "off." To accomodate this, numbers can be translated into binary code which is represented using only ones and zeros. Ones are used to represent on states or logical highs while zeros are used to symbolize logical lows or off states. The chart in the previous section (Table 1.) shows the base ten or decimal number and its binary equivalent for the numbers from 0 to 15. The binary numbers have all been written with four digits. When the digits at the left-hand end are zero they are acting as placeholders. These placeholders will prove to be essential when operating the Computer Trainer. Each on or off symbol represents one bit. Therefore, each of the numbers on the chart is represented as a four bit binary number.

The memory is organized in pages, each page being a unit of 256 words. Each word is essentially an eight bit binary number or byte. Each byte or word of memory is given a specific binary address or location. The first location is location 0000 (0 in decimal). The second is 0001 (1 in decimal), the third is 0010 (2 in decimal) and so on.

Along the bottom of the front panel of the computer, there are a series of switches. They are arranged so that the eight switches on the left are data switches while the seven switches on the right are address switches. When a switch is up (towards the red lights (LEDs)), it indicates a logical high. A logical low is indicated when a switch is down. The indicator lights above the switches indicate a logical high when lit and a logical low when not illuminated.

Set all of the switches as follows:

Run to the left
RST to the left
Load to the right
NMI to the left
ROM to the left

All data and address switches down, i.e., away from the LED indicators. Note that the run switch must be to the left for the load, the data, and the address switches to operate.

Plug the computer in. The LEDs above the eight switches on the right-hand side indicate the memory location which is being addressed. They should indicate 000 0000, the first memory location. The lights on the left-hand side of the board labeled "data" display the contents of the address selected by the switches on the right-hand side of the board.

Since the user hasn't stored anything in memory at this location, there is no way of predicting what it contains. Change the address switches so that

they indicate 0000 0001, the second location in memory. The LEDs to the right are now displaying the contents of the second memory location. It is possible to continue changing the address switches to see what is contained in other memory locations. Jot down the memory locations and their contents in a form similar to the following:

Data	Address
	000 0000
	000 0001
	000 0010
	000 0011
	000 0101

Unplug the computer. Plug it in again. Compare the contents of specific memory locations with those jotted down. They are probably the same. When each memory location looses power completely and then receives it again, it will tend to have the same contents that it had when it was initially powered up (before anything was ever stored in it). These contents vary from one memory chip to another, but tend to be constant for each chip.

In addition to looking at or reading the contents of memory locations, it is possible to write or place information in memory. Set the address switches to 000 0000. The contents of location 000 0000 are now displayed. Set the data switches (the switches on the left-hand side) with the contents desired for location 000 0000, say 0000 1111. Now push the "Load" switch to the left and then back to the right. This will load the contents of the data switches into the memory location indicated by the address switches. Always return the load switch to the right after each load operation. If the loaded pattern, when displayed, does not duplicate the data switch settings, try moving the load switch back and forth once or twice. Occasionally, the load switch may "glitch", that is, may introduce a false pulse so that it is a good practice to always check the data switch positions against the data light readout.

Try loading other memory locations with data. A fairly simple data choice is to load each location with the number that is the same as its address. Go through and read the memory locations that have just been written in.

Information can be stored in the form of a memory look-up table. For example, a binary addition table such as the one below can be stored in memory.

Push all the address and data switches to the "low" position. Consider the two right-hand most address switches to represent one binary number. The two address switches adjacent to the first switches represent a second binary number. The sum of the two numbers will be placed in memory as data.

DATA	MEMORY LOCATION
0000 0000	000 0000
0000 0001	000 0001
0000 0010	000 0010
0000 0011	000 0011
0000 0001	000 0100
0000 0010	000 0101
0000 0011	000 0110
0000 0100	000 0111
0000 0010	000 1000
0000 0011	000 1001
0000 0100	000 1010
0000 0101	000 1011
0000 0011	000 1100
0000 0100	000 1101
0000 0101	000 1110
0000 0110	000 1111

Load memory with the table on the preceding page. "Read" memory to check and be sure that the table is properly loaded and is ready for use.

The look-up table which has just been loaded into memory can be used to perform some simple binary addition. For example, say one wishes to find the sum of 01 and 11. To do this, treat the two least significant bits of the address switches as the first number and the next two switches as the value for the second number which is to be added. In other words, set the memory address switches to 000 0100. The LEDs above the data switches will indicate the results; 0000 0100.

For a second example, the sum of 10 and 10 can be found. The address switches are set to 000 1010. The LEDs above the data switches should then indicate the results which are 0100.

In actual computer programming, the look-up table is a valuable technique for storing information in as little space as possible and is used frequently for a number of applications.

The procedure for reading and writing in RAM has just been discussed. This procedure is an essential part of using the Model 300 Computer Trainer.

TO PREVENT EXCESSIVE BOUNCE ON THE LOAD SWITCH ON NEW TRAINERS, BEAR DOWN ON IT WHEN RETURNING IT TO THE RIGHT.

RUNNING A PROGRAM ON THE MODEL 300 COMPUTER TRAINER

Introduction:

The procedure for running a program on the Model 300 Computer Trainer is described.

Discussion:

In order for the computer to perform a task such as adding several numbers, it must be instructed to perform a sequence of events which will lead to the desired results. The instructions are refered to as a program.

The program is stored in memory prior to execution. (This procedure is described in the previous section.).

One of the shortest and simplest programs which can be written for a computer is a jump program which jumps to itself. When executing such a program, the computer is given a jump or "go to" command which instructs the machine to begin executing the instruction which is located at the memory location specified by the jump command.

This can be accomplished by the Jump Absolute command. The Operation Code (op code) or actual machine instruction for this instruction is 4C. As is the case with any instruction in the absolute mode, the instruction uses three bytes or three consecutively addressed memory locations. The first word or byte consists of the op code while the second word is the low portion of the memory address. The third word is the page or high portion of the memory address. Assume that a jump command (4C) has been placed at location 01 on page 00. In order to have the jump jump to itself, the second byte of the instruction is 01 while the third byte is 00.(The actual code placed in the machine is translated from hex into binary.)
Note that on the Model 300 Computer Trainer, only page 00 exists.

Load the following program into memory:

Op Code	Location	Mnemonic	Note
4C	01	JMP Oper	Jump Instruction
01	02	Data	Memory location being jumped to
00	03	Data	Page being jumped to

To run the program, the following steps are taken. The user must let the computer "know" at what memory location the program begins. When the computer prepares to execute a program, it always begins running the program by starting at the memory location specified by the reset vector. This information is contained in memory locations 7C and 7D. Before running the program, the user must place the low order memory location, the location within a page, for the begining of the program at location 7C. This is done by loading memory location 7C with the memory location at which the program will begin. The high order or page is loaded into memory location 7D. (Note: In systems with the full 65K memory, the reset vector is located at memory locations FFFC and FFFD.) Next, the reset switch is pushed high (to the right). This causes the program counter to be initialized or set to the location specified by the restart vector. The program counter is a 16 bit register which contains the memory address for the next instruction or command. Following this, the run switch is pushed high (to the right). Then, the reset line is set low (pushed left) so that the computer will run.

While the program is being executed, the pattern formed by the LEDs should be noted. The set of LEDs on the left-hand side indicates the op code of the instruction which is being executed. If more than one instruction is

being performed, or if the instruction is more than one byte long, the data patterns will be superimposed upon each other. They should be lighted as in the diagram below (● indicates "light on").

0 ● 0 0 ● ● 0 ● 0 0 0 0 0 ● ●

The address lights indicate the memory locations at which the microprocessor is obtaining instructions. While it appears that the microprocessor is always addressing location 03, this is not actually the case. The LEDs are being turned on and off so rapidly that they appear to be on constantly!

To halt the processor, set the reset low or to the right. Then set the run switch to the low position (to the left).

Before proceeding with more complicated programs, write and execute several more jump absolute programs starting at different memory locations. It is crucial that one become totally familiar with the procedure for loading and running a program before going on to other experiments.

Another sample of a jump program is given below. Load it and run it before writing addition 1 programs.

Op Code	Location	Mnemonic	Note
4C	04	JMP Oper	Jump instruction
04	05	Data	Memory location being jumped to
00	06	Data	Page being jumped to

Before running the program, be sure to load the reset vector with 04 and 00 to indicate the location and page on which the program begins! While the program is running, the LEDs should be lighted as in the diagram below.

0 ● 0 0 ● ● 0 0 0 0 0 0 ● ● ●

Notice that the address LEDs which are lit are not all equally bright. This is because the machine is addressing locations which have the same LED as an indicator while each of the other LEDs is only lit during a portion of the commands. Since the Model 300 has only 128 bytes of memory, the page portion of each address does not affect program operation. However, the proper procedures should be used with respect to page addressing so that difficulty will not be encountered in programming on larger systems.

BINARY ADDITION

Introduction:

Binary addition is examined.

Discussion:

The 6502 microprocessor is capable of performing binary addition. Before programming the computer to perform this operation, it is necessary to know how to do it.

The addition of two binary numbers is quite simple. There are essentially four possible additions with three possible results. The table below shows that in binary addition 0 + 0 = 0, 1 + 0 = 1, 0 + 1 = 1, and 1 + 1 = 10. When two binary numbers, each having one digit are added, the procedure followed is the same as that for base ten numbers.

Table 1. Binary Addition Table

	0	1
0	0	1
1	1	10

One begins by adding the two right-hand most digits. If the addition results in a two digit binary number, the least significant digit is written as the least significant digit of the sum and the most significant digit is carried. An example is given below.

```
 1 0 1 1 1          1
 1 0 1 0 1      1 0 1 1 1
-------------   1 0 1 0 1
                -------
                      0
```

This procedure is followed for each of the columns of digits, as when working with base 10. Therefore:

```
    1 1 1
  1 0 1 1 1
  1 0 1 0 1
-------------
1 0 1 1 0 0
```

The sum of 10111 and 10101 is 101100.

Since binary numbers aren't always positive, a technique is needed for adding signed numbers. This can be accomplished by expressing numbers in twos complement form before performing the addition. The procedure below lists the steps necessary to change -7 into twos complement form.

1. Express the number in binary form: -0000 0111.

2. Change the value of each digit: if the digit is 0, express it as 1; if it is 1, express it as 0. This process is refered to as finding the complement. (Note: the negative sign is dropped.) 1111 1000

3. Add 1 to the number which has just been formed. Notic that the negative sign is not used. 1111 1001. 1111 1001 is the twos complement equivalent of -7.

To find the sum of two negative numbers, say -7 and -5, the two numbers are converted to twos complement form, added, and then converted back into decimal form.

-7 is -0000 0111 binary or 1111 1001 in twos complement form.

-5 is -0000 0101 binary or 1111 1011 in twos complement form.

The sum is then found:

```
  1 1 1 1  1 0 0 1
  1 1 1 1  1 0 1 1
-------------------
1 1 1 1 1  0 1 0 0
```

Since this number is negative and is in twos complement form, it must be converted to binary form. To do this:

1. Subtract 1 from the twos complement number (or add the twos complement equivalent of -1 which is 1111 1111).

2. Find the complement of the number formed above to form the binary number -0000 1100. Remember the negative sign! This is the binary form for -12.

Whenever signed numbers are being added, especially with respect to the 6502 microprocessor, it will be assumed that the results obtained by the machine are expressed as a binary eight bit number. When the eighth bit is high (is a 1), this means that the number is negative. When this bit, the most significant bit, is low (0), the number being expressed is positive. The procedure given above is followed only when the eighth bit is high.

The sum of a negative number and a positive number can also be found. To find the sum of 8 and -5, the following method, which can be used for finding the sum of any negative and positive number is used.

1. Convert the negative number to twos complement form. -5 decimal (base ten) equals -0000 0101 binary which is 1111 1011 in signed binary form.

2. Convert the positive number into twos complement form. 8 decimal is equivalent to 0000 1000 in twos complement form (Positively signed binary numbers are expressed the same as in binary form.)

3. Find the sum of the twos complement numbers found above in 1 and 2.

```
  1 1 1 1  1 0 1 1
  0 0 0 0  1 0 0 0
-------------------
1 0 0 0 0  0 0 1 1
```

Since the eighth bit is low, the number above is positive, and, therefore, is a positive 3 (Notice that the ninth bit is high. On the processor, this condition would cause the carry flag to go high, but this bit does not indicate sign) Had the eighth bit been high (1), the number above would be a negative number and would be expressed in twos complement form.

For pratice, find the following sums by converting the numbers to signed binary form and then performing the binary calculations. Give final results in decimal form.

-9, -7 :12, -8 : 6, -10 : 3, 8

ADDING ON THE COMPUTER

Introduction:

The programming steps needed to perform binary addition on the Model 300 Computer Trainer (machine language programming) are discussed.

Discussion:

Binary addition can be performed with the computer. The programming techniques required for this are somewhat more sophisticated than those used for a simple Jump Absolute program. It is also essential to know something about computer architecture.

The accumulator of the 6502 microprocessor is an eight bit wide register in which the results of most of the microprocessor's arithmetic functions are stored.

Another very important register is the Processor Status Register which is also eight bits wide. Each bit in the register is used as a flag indicator. Begining with the least significant bit and going to the most significant bit of the register, the bits indicate; carry, zero result, interrupt disable, decimal mode, break command, a bit which is reserved for future expansion, overflow, and negative result.

When writing a program to perform binary addition, the flags or bits of the status register with which one must be familiar are the carry flag, the decimal mode flag, the overflow flag, and the negative result flag.

The carry flag is set high when an addition resulting in a nine bit number has been performed. If the number is eight or fewer bits long, the carry bit is reset (set to 0).

The computer commands can be used to directly specify the state of the carry flag. To set the carry flag to 1 (high), the Set Carry Flag command is used. The mnemonic (letter code) for this is SEC and the hexidecimal op code is 38. The instruction requires only one byte of memory. Its addressing mode is implied, indicating that no further memory referencing is needed in the instruction. The instruction affects only the carry flag.

A second instruction which affects only the carry flag is the Clear Carry Flag instruction. This command sets the carry flag to 0 (low). Like the set carry flag instruction, it uses only one byte of memory and uses the implied addressing mode. The mnemonic for the Clear Carry Flag instruction is CLC and the op code is 18.

The 6502 is able to perform simple arithmetic operations in either binary or binary coded decimal format. When the decimal mode flag is high (1) addition and subtraction will be performed in the binary coded decimal form. The instruction used to accomplish this is Set Decimal Mode which has a mnemonic of SED and an op code of F8. The instruction uses one byte of memory and is in the implied mode. It affects only the decimal mode flag.

In order for binary arithmetic operations to take place in the microprocessor, the decimal mode flag must be set low. This is done by using the Clear Decimal Mode command which affects only the decimal mode flag by setting it low. D8 is the op code for this instruction which has a mnemonic of CLD.

The overflow flag is set high to indicate that the results of an arithmetic operation could not be contained in eight bits. It is extremely useful for work with signed numbers.

The programmer has direct access to the overflow flag through the use of the Clear Overflow Flag instruction. This one byte, implied mode instruction, has CLV as its mnemonic and an op code of B8.

The negative flag always has the same value as the eighth bit (D_7) of the accumulator. Being able to have direct access to this bit is useful when dealing with signed numbers.

Instructions other than those which affect the status of flags in the processor status register and the jump command are essential for fairly simple computer programming.

In order to perform operations using addition, it is necessary to transfer information which is stored in memory (such as numbers which are to be added) into the accumulator. This is done with a Load Accumulator with Memory instruction. The mnemonic associated with this command is LDA. There are eight possible addressing modes for this command, each having its own op code. The various modes of addressing will be discussed in a later section.

A Load Accumulator with Memory immediate mode instruction uses two bytes of memory. The first contains the op code which is A9. The mnemonic is LDA #Oper. When the immediate addressing mode is used, the second byte of the instruction contains data, that is, the actual number which is to be placed in the accumulator.

When an arithmetic operation has been completed and has been stored in the accumulator, it is necessary to be able to place the information which is in the accumulator in memory. This will make it possible for the programmer to gain access to the results of the operation. This is accomplished with the Store Accumulator in Memory command. This instruction places the contents of the accumulator in memory, but preserves the contents of the accumulator. The mnemonic for this instruction is STA and there are seven possible addressing modes. Since the Model 300 has memory only on page 0, the zero page mode of addressing is the most appropriate mode to use when learning the instruction. The op code for this is 85 and the mnemonic is STA Oper. The first byte of this two word instruction is the op code. The second is the memory location which MUST be located on page 0.

Once a mode of addition has been selected, that is, either binary or binary coded decimal arithmetic, memory contents can be added to the contents of the accumulator. The Add Memory to Accumulator with Carry instruction which has a mnemonic of ADC is used for this purpose. This instruction has eight possible addressing modes. The immediate mode uses a mnemonic of ADC #Oper and has an op code of 69. As previously stated, all instructions in the immediate mode are two words long with the first word being the op code and the second being the actual data. Care must be taken when using this instruction that proper preparation for its use has been made. The programmer must be careful to set the decimal mode flag. He must also be careful to clear the carry flag. Otherwise, if the carry flag is high, a one will be added to the results of his addition. The number which the programmer wishes to add to the number which is contained in the second byte of the ADC instruction must be stored in the accumulator PRIOR to the execution of this instruction. Otherwise, whatever was in the accumulator will have been added to the data contained in the instruction.

Enough information has been provided so that a simple addition program can be written and tested. The first step in writing any program which involves arithmetic is to determine whether the program will make use of binary arithmetic or binary coded decimal arithmetic. In this case, binary arithmetic will be used. To simplify matters, only positive numbers will be added. Once the form of arithmetic has been determined, the memory location

where the sum of the two numbers is to be stored can be decided. In this case, the results will be stored at location 0 on page 0. They could, of course, be stored at any other location within the available memory.

Before the actual addition can take place, the accumulator must contain one of the numbers which is to be added. At this point, the two numbers can be summed. Then it is desirable to have the results stored in memory at location 0 on page 0. Finally, some method of preventing the microprocessor from executing whatever it finds in memory following the desired program must be found. Since there is no way of predicting what might lie in other memory locations, it is quite possible that the processor would execute what appeared to it to be instructions which would result in destroying the results of the addition. Since there is no halt command for the 6502 processor, another approach will be used. Recall that the Jump Absolute instruction caused the processor to repeatedly jump to one memory location without affecting the contents of any memory locations or registers. This can be used as a means of preventing the processor from executing unintended instructions.

From the steps outlined above, a short list of what the computer is to do can be made. This list can then be used to write the program which the computer will execute.

1. Choose binary arithmetic mode.
2. Load the accumulator with the desired first number, XXXX XXXX.
3. Add the desired second number, YYYY YYYY to the accumulator.
4. Place the contents of the accumulator in memory location 0 on page 0.
5. Have the computer jump repeatedly to the same jump command.

This program will begin at location 1 on page 0. It can not begin at location 0 because that location is being used to store the results of the addition. Programs, of course, can be started at any memory location.

Op Code	Location	Mnemonic	Note
D8	01	CLD	Set in binary mode
18	02	CLC	Clear carry bit
A9	03	LDA #Oper	Load accumulator with first number
XXXX XXXX	04	Data	First number
69	05	ADC #Oper	Add immediate to accumulator
YYYY YYYY	06	Data	Second number
85	07	STA Oper	Store results
00	08	Data	Memory location where data to be stored
4C	09	JMP Oper	Jump to prevent further execution
09	0A	Data	Memory location to jump to
00	0B	Data	Page to jump to

Load the program into memory. Substitute 0000 0101 for XXXX XXXX and 0000 1000 for YYYY YYYY. First run the program with the memory in the read-only mode. This is accomplished by setting ROM to the right, then RST to the right, then RUN to the right, and then, finally, RST to the left. The RUN light should be on and the address and data lights should indicate that the computer is in the jump loop at the end of the program. If this does not occur, stop the computer by bringing RST to the right and the RUN to the left. Recheck the program which was entered. Once the program successfully gets to the jump routine, run the program with the ROM switch to the right. It is always best to first test a program in ROM mode and then use it in Read/Write mode. The ROM switch prevents the processor from writing into memory. This protects the stored program from being completely erased if it has a 'bug' in it. After running the program

in Read/Write mode, stop the processor using the standard procedure.

Check your results by reading memory location 0 because this location will contain the results of the addition (The sum is 13 decimal or 0000 1101 binary.). Try substituting other binary numbers for XXXX XXXX and YYYY YYYY. The results should again appear in location 0. To do this, merely load the new numbers in the data locations designated for X and Y and run the program. There is no need to reload the entire program!

CONDITIONAL BRANCHING

Introduction:

The concept of conditional branching is discussed.

Discussion:

During the execution of a program, it is often desirable to be able to perform one sequence of steps if a specific situation is true and to execute a different sequence if the situation is false. This is made possible through the use of conditional branch instructions.

A conditional branch instruction essentially checks the Processor Status Register to determine the status of a specified bit. If the condition specified by the instruction is true, the processor will skip the instructions for the number of memory locations specified by the second byte of the instruction. This number is added to the value contained in the program counter. The value contained in the programmer at the time of the addition is the location of the instruction immediately following the second byte of the conditional branch instruction. If the condition is false, the processor will execute the instruction immediately following the second byte of the conditional branch instruction.

Conditional branch instructions are in the relative addressing mode. Each instruction is two bytes long with the first byte being the op code. The second byte of the instruction is the number of memory locations which are to be "skipped" if the branch does occur.

Instructions which branch on the following conditions; carry bit clear, carry bit set, result zero (zero bit set), result not zero , negative bit high, negative bit low, overflow bit clear, and overflow bit set.

The following is a list of the mnemonic, op code, and a description of each of the conditional branch instructions.

Mnemonic	Op Code	Description
BCC Oper	90	Branch on carry clear
BCS Oper	B0	Branch on carry set
BEQ Oper	F0	Branch on result zero
BMI Oper	30	Branch on result minus
BNE Oper	D0	Branch on result not zero
BPL Oper	10	Branch on result plus
BVC Oper	50	Branch on overflow clear
BVS Oper	70	Branch on overflow set

It is possible to branch forward or backward in memory: going forward up to 127 locations or going back 128 locations. When branching, the processor treats the number which indicates how far to branch as a signed number. That is, if the 8th bit is high, the processor deals with the number as a negative number. When branching to a lower address, the number of memory locations is expressed in negative form. As with positive values, it is added to the program counter, which contains the address for the memory location immediately following the second byte of the conditional branch instruction. To demonstrate what a conditional branch instruction does, a short program can be written. The accumulator will be loaded with a number, X, which will be chosen by the programmer. If the number has bit 8 (D_7) high, the negative flag will be set and the accumulator will be stored in location 00 of memory. If the negative flag is not affected or is reset, the contents of 00 will remain unchanged. Steps which can be used to write this program are as follows.

1. Reserve location 00 for results. Be sure that it initially contains 00.
2. Load the accumulator with XXXX XXXX.
3. Branch if the negative flag is high (to step 5).
4. Jump absolute to prevent further execution.
5. Load the memory location 00 with the accumulator.
6. Jump absolute to prevent further execution.

The following program can be written from the steps given above.

Op Code	Location	Mnemonic	Note
00	00	Data	Load location 00 with 00
A9	01	LDA #Oper	Load accumulator with X
XXXX XXXX	02	Data	Number being placed in accumulator
30	03	BMI Oper	Branch if negative flag high
03	04	Data	# of locations to s ip
4C	05	JMP Oper	Jump Absolute to prevent further execution
05	06	Data	Location to jump to
00	07	Data	Page to jump to
85	08	STA Oper	Store accumulator in memory
00	09	Data	Memory location to hold accumulator
4C	0A	JMP Oper	Jump absolute to prevent further execution
0A	0B	Data	Memory location to jump to
00	0C	Data	Page to jump to

After the program has been loaded, set the vector to location 01 on page 00. Eachtime the program is going to be run, location 00 should be loaded with 00 prior to running the program.

Conditional branch instructions enable the processor to execute one set of instructions if a given condition is true, and to execute a different set of instructions if the condition is false.

ADDING SIGNED NUMBERS ON THE COMPUTER

Introduction:

Adding signed numbers on the 6502 is discussed.

Discussion:

When dealing with signed numbers, the accumulator can handle numbers with an absolute value of up to 127. The bits of the accumulator are designated as bits 0 through 7 with the zero bit representing the least significant digit and the eighth bit representing the most significant bit. When dealing with signed numbers, the eighth (D_7) bit is used to indicate whether a number is positive or negative. When the eighth bit is high, it indicates a negative number, and when low, it indicates that the accumulator contains a positive number. For example, if -6 were added to 3, the expected results would be -3. Converting -6 to signed binary form gives 1111 1010. When 3 is converted into twos complement form, 0000 0011 is obtained.

```
  1 1 1 1  1 0 1 0
  0 0 0 0  0 0 1 1
-------------------
  1 1 1 1  1 1 0 1
```

Note that the eighth bit is one, indicating that the results are negative. This also indicates that the results must be converted to binary format before they can be converted to decimal form. Converting 1111 1101 into binary form gives -0000 0011. However, the eighth bit will sometimes be high when the results are positive. In such a case, the overflow flag will have been set, indicating that the allowable range of values has been exceeded in the operation. For example, the sum of 126 and 126 would yield:

```
  1 1 1 1  1 1 1 0
  1 1 1 1  1 1 1 0
-------------------
  1 1 1 1  1 1 0 0
```

Since the eighth bit is high, a negative number is indicated. However, if one were to test the overflow flag, it would be found to be high, indicating that the results were not valid. A similar situation can occur when two negative numbers are summed. Therefore, whenever signed numbers are being dealt with, the overflow flag should always be tested.(Note: In the case of 1111 1110 + 1111 1110, the computer can handle only eight digits at a time. It will, therefore, with numbers as large as the above, leave off the most significant digit.)

It is also possible to test the eighth bit automatically to determine if the number is negative or positive. This can be done by testing the negative flag which will test high if the eighth bit is high (this flag will always have the same value as the eighth bit of the accumulator).

When a number, as the results of an operation, has caused the overflow flag to be set high, the computer is able to indicate this through the use of a conditional branch instruction.(The use of conditional branch instructions is discussed in a previous section.) To determine the status of the overflow bit, the Branch Overflow Set instruction can be used. The mnemonic is BVS and the op code is 70. The first byte of the instruction is the op code and the second tells how many memory locations are to be skipped if the overflow flag is high.

The basic steps for writing a program to add signed numbers are:

1. Load location 01 with 0000 0000. If this remains unchanged during the execution of the program, it will indicate that the overflow flag was not set high.
2. Set in binary mode.
3. Clear the carry flag.
4. Load the first number (XXXX XXXX) into the accumulator.
5. Add the second number (YYYY YYYY) to the first number.
6. Place the results in location 00.
7. Test to see if the overflow flag is set, if it is, skip to step 9.
8. Jump Absolute. This will, essentially , prevent the processor from executing instructions which lie beyond those which the user intends it to execute.
9. Increment or add 1 to location 01 to indicate that the overflow flag is high. This is accomplished with the Increment Memory instruction which has a mnemonic of INC Oper and an op code of E6. The first byte of this two byte instruction is the op code and the second is the memory location which is being incremented.
10. Jump Absolute. If the computer has executed the branch true portion of the program (step 9), the Jump Absolute instruction contained in step 8 will not affect the processor. To prevent the computer from executing information contained later in memory as if it were part of the program, a Jump Absolute is used.

Once the sequence of programming steps has been established, the program can be easily written.

Op Code	Location	Mnemonic	Note
00	01	Data	location 01 to indicate overflow status
D8	02	CLD	Set in binary mode
18	03	CLC	Clear carry flag
A9	04	LDA #Oper	Load first number in accumulator
XXXX XXXX	05	Data	First number being added
69	06	ADC #Oper	Add 2nd number to accumulator
YYYY YYYY	07	Data	Number being added to accumulator
85	08	STA Oper	Store sum in memory
00	09	Data	Memory location of sum
70	0A	BVS	Branch if overflow high
03	0B	Data	#of addresses to skip if overflow high
4C	0C	JMP Oper	Prevent further execution
0C	0D	Data	Location to jump to
00	0E	Data	Page to jump to
E6	0F	INC Oper	Increment to indicate true overflow
01	10	Data	Memory location to increment (Page 0)
4C	11	JMP Oper	Prevent further execution
11	12	Data	Location to jump to
00	13	Data	Page to jump to

Initially, substitute -3 for XXXX XXXX and 9 for YYYY YYYY. Remember to place the numbers in twos complement form. The binary results obtained should be 0000 0110 or 6 decimal. The overflow flag shouldn't be affected so the contents of memory location 01 should still be 0000 0000.

On the second run of the program, substitute 1111 1111 for XXXX XXXX and 0000 1111 for YYYY YYYY. The overflow flag should be set after the program has been run and memory location 01 should contain 0000 0001. Before adding another set of numbers, take care to load location 01 with 0000 0000, or after the next addition, location 01 will contain either 0000 0001 (if the overflow is low) or 0000 0010 (if the overflow is high)!

Before going on to the next section, try performing other signed number additions. Work them out on paper as well as on the machine as this will help one to develop binary arithmetic skills.

SUBTRACTION

Introduction:

Subtraction is discussed.

Discussion:

Subtraction as well as addition can be performed by the computer. Before having the computer execute a subtract instruction, the carry flag must be set high. To carry out subtraction, the computer will automatically complement the number being subtracted. The carry bit is then added to the complement just obtained. The number from which the subtraction is to occur is added to the number which has been dealt with in the manner above.

After the subtraction has been performed, the status of the carry bit must be examined. If it is high, this is an indication that no borrow has taken place and that the result of the subtraction is a positive number. A negative number is indicated if the status of the carry flag is low.

A demonstration of the technique used by the microprocessor for subtraction is quite simple. Below, 5 is subtracted from 9.

```
  0 0 0 0  1 0 0 1
  0 0 0 0  0 1 0 1
```

Before combining the numbers, the complement of 0000 0101 must be found.

```
  0 0 0 0  1 0 0 1
  1 1 1 1  1 0 1 0
```

Next, the carry bit is added to the complemented number.

```
  0 0 0 0  1 0 0 1
  1 1 1 1  1 0 1 1
```

The operation is then performed.

```
  0 0 0 0  1 0 0 1
  1 1 1 1  1 0 1 1
1 0 0 0 0  0 1 0 0
```

The 1 on the left-hand side doesn't fit in the accumulator as it can contain only eight bits. The ninth bit is expressed as the carry bit. Since the carry bit is high, it indicates that no borrow has taken place. In other words, the difference is a positive number, in this case, 4.

For an example of a negative difference, subtract 8 from 2.

```
  0 0 0 0  0 0 1 0
  0 0 0 0  1 0 0 0
```

Find the complement of 8:

```
  0 0 0 0  0 0 1 0
  1 1 1 1  0 1 1 1
```

Add the carry bit:

```
  0 0 0 0  0 0 1 0
  1 1 1 1  1 0 0 0
```

Add the numbers obtained on the previous page:

```
  0 0 0 0  0 0 1 0
  1 1 1 1  1 0 0 0
  ---------------
  1 1 1 1  1 0 1 0
```

Since no ninth or carry bit has been generated, a borrow has occured. The results must be converted from twos complement to binary form giving -0000 0010. The decimal equivalent is -6.

The following steps can serve as a guideline for a computer program.

1. Reserve memory location 00 for results and set location 01 to 0 as the carry flag indicator.
2. Clear the decimal flag
3. Set the carry bit high.
4. Load the accumulator with the number which will have a number subtracted from it (XXXX XXXX).
5. Use the subtract with Borrow instruction with mnemonic SBC and op code E9 in the immediate mode.
6. Store the difference in location 00.
7. If the carry bit is high, go to step 9.
8. Jump Absolute to prevent the processor from executing further instructions.
9. Increment memory location 01.
10. Jump Absolute to prevent execution of further instructions.

Load the program given below into memory. The first time through, substitute 1 for XXXX XXXX and 9 for YYYY YYYY. The second time the program is run, subtract 7 from 9.

Op Code	Location	Mnemonic	Note
00	01	Data	Carry flag indicator
D8	02	CLD	Clear decimal flag
38	03	SEC	Set carry flag
A9	04	LDA #Oper	Load accumulator with first number
XXXX XXXX	05	Data	First number
E9	06	SBC #Oper	Subtract 2nd# from First #
YYYY YYYY	07	Data	Second number
85	08	STA Oper	Store results in memory
00	09	Data	Memory location
B0	0A	BCS	Branch if carry flag high
03	0B	Data	Addresses to skip if branching
4C	0C	JMP Oper	Prevent further execution
0C	0D	Data	Memory location to jump to
00	0E	Data	Page to jump to
E6	0F	INC Oper	Increment memory to show carry high
01	10	Data	Memory location to increment
4C	11	JMP Oper	Prevent further execution
11	12	Data	Memory location to jump to
00	13	Data	Page to jump to

Load the restart vector with location 02 on page 00. (This is placed at location 02 because this is the location which contains the first executable instruction.) Run the program. After the fist time through, be sure to reset memory location 01 to 00. Otherwise, there will be now way of telling if the subtraction affected the carry flag.

Work out subtractions on paper as well as by computer. This will help to familiarize the programmer with binary arithmetic.

BINARY CODED DECIMAL ARITHMETIC

Introduction:

Binary Coded Decimal arithmetic is examined.

Discussion:

As well as being capable of performing binary addition, the 6502 can also deal with Binary Coded Decimal (BCD) numbers. The actual greatest absolute value of numbers which can be expressed in this format in eight bits is less than that which can be expressed in simple binary form. For some applications, this is offset by the ease with which base 10 numbers can be converted to and from binary decimal code.

In binary decimal code, each digit of a base 10 number is expressed as a four digit binary number as listed below.

Decimal	Binary Coded Decimal (BCD)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Notice that the binary numbers 1010, 1011, 1100, 1101, 1110, and 1111 are not used when numbers are expressed in BCD. The eight bits of the accumulator can be used to express any base 10 two digit number. Examples:

79	0111 1001
84	1000 0100
56	0101 0110
2	0000 0010

As in binary mode, when the results of an addition produce a binary number which is longer than eight bits, the carry flag will be set high. If 55 and 54 are added, 109 is obtained. In BCD, this is expressed as:

```
  0 1 0 1  0 1 0 1
  0 1 0 1  0 1 0 0
---------------------
1 0 0 0 0  1 0 0 1
```

Remember that in BCD code, the binary number 1010 is not used. The sum of 0101 and 0101 is 0001 0000. In the case where 54 and 55 are added in BCD code on the computer, the results are stored in the eight bit accumulator and are 0000 1001 or 9. To obtain the entire results, the carry flag must be checked. If the carry flag is high, a third digit in the results is indicated (the sum will be 1XX). A two digit sum will reset the carry flag to 0.

When a flag is tested, some way of showing the results of the test is desired. One simple method is to store a known number in a specified memory location. For example, 0 could be stored in memory location 01. If the flag tests high, the memory location can be incremented, that is, the value in that location can be increased by one. This is accomplished by an Increment Memory by One instruction which has INC as its mnemonic.

There are four possible addressing modes. When the zero page mode is used, the corresponding op code is E6. The instruction is two bytes long with the first byte containing the op code and the second containing the address for the memory location which is to be incremented.

The steps below are a guide for writing a program to add two binary coded decimal numbers. Note that a provision is being made for results which generate a carry.

1. Set aside location 00 for results and set location 01 to 0 as the carry flag indicator.
2. Set the decimal flag to BCD mode.
3. Clear the carry flag.
4. Load the accumulator with the first number.
5. Add the second number to the accumulator.
6. Store the results of the addition at location 01.
7. Check to see if the carry flag is high.
8. If the carry flag is high, go to step 10.
9. Jump absolute to prevent execution of further instructions.
10. Increment memory location 01 to indicate that the carry flag is high.
11. Jump absolute to prevent further execution of instructions.

Load the program which follows into memory.

Op Code	Location	Mnemonic	Note
00	01	Data	Reserved for flag indicator
F8	02	SED	Set decimal flag
18	03	CLC	Clear carry flag
A9	04	LDA #Oper	Load accumulator with first number
XXXX XXXX	05	Data	First number
69	06	ADC #Oper	Add 2nd number
YYYY YYYY	07	Data	Second number
85	08	STA Oper	Store results in memory
00	09	Data	Memory location to store results at
B0	0A	BCS Oper	Check carry flag
03	0B	Data	Distance to branch if carry bit high
4C	0C	JMP Oper	Prevent further execution
0C	0D	Data	Memory location to jump to
00	0E	Data	Page to jump to
E6	0F	INC	Indicate carry bit high by memory inc.
01	10	Data	Memory location to be incremented
4C	11	JMP Oper	Prevent further execution
11	12	Data	Memory location to jump to
00	13	Data	Page to jump to

Set the restart vector to location 02 on page 00. Run the program adding 50 + 64. The results that appear in memory location 00 will be 14 or 0001 0100. Location 01 will have been incremented indicating that the addition has resulted in a three digit number; 114.

Try adding other BCD numbers. Be careful to use only numbers that are acceptable in BCD form (eg. don't use 1111). Before adding each new set of numbers, load location 01 with 0. This will insure that location 01 will contain a 0 unless it has been incremented, indicating that the carry flag has been set.

LOGICAL OPERATIONS

Introduction:

Logical operations and their applications are introduced.

Discussion:

The 6502 microprocessor is able to perform three different logic operations. They are AND, OR and exclusive OR. The results of each can be expressed by a truth table.

AND			OR			EXCLUSIVE OR		
	1	0		1	0		1	0
1	1	0	1	1	1	1	0	0
0	0	0	0	1	0	0	0	1

A four bit example of each operation is given below.

AND	OR	EXCLUSIVE OR
1 0 0 1	1 0 1 0	0 0 1 1
1 1 0 0	0 1 1 0	1 0 1 0
1 0 0 0	1 1 1 0	0 1 0 0

The most frequent application of the logical AND instruction is to set specific bits in the accumulator low. If, for example, a programmer wishes to clear bits 5 and 6, he can AND those two bits with 0. When either a high or a low is ANDed with a low, the results are low. To preserve each of the other bits in the accumulator, they are ANDed with 1. Since a high ANDed with a high remains high and a low ANDed with a high remains low, the other bits are unchanged. Below an example is given. To clear bits 5 and 6 of 1010 1010, this number is anded with 1100 1111.

```
1010 1010
1100 1111
1000 1010
```

The AND instruction can be used in eight modes. In the immediate mode, the mnemonic is AND #Oper and the op code is 29.

The OR instruction, unlike the AND command, is used mainly to set bits high. Supose that a programmer wishes to set bits 1, 2, and 3 high. To do this, he will OR bits 1, 2, and 3 with 1. This will cause these bits to be set high regardless of their initial state. All other bits will be ORed with 0, causing them to retain their initial state.

```
1010 1010
0000 0111
1010 1111
```

Like the AND instruction, the OR instruction has eight different modes of address. The op code for the immediate mode is 09 and the mnemonic is ORA #Oper.

The third logical operand is the exclusive OR. The primary application of this operand is to find the complement of binary numbers. This is extremely valuable for converting numbers to and from their negative values. When exclusive ORed with a low, a high goes low while a low goes high. The number used above is easily complemented using the exclusive OR operand.

```
1010 1010
0000 0000
0101 0101
```

To obtain the negative value of the original number in twos complement form, a one is added to its complement.

To convert a number which is negative into positive form, one is subtracted from the number and the difference is exclusive ORed with 0000 0000 resulting in the positive value.

The programming steps needed to convert a negative number into positive format are given below.

1. Clear the decimal mode flag.
2. Set the carry flag high.
3. Load the negative number into the accumulator (XXXX XXXX).
4. Subtract 1 from the accumulator.
5. Exclusive OR the accumulator with 0000 0000.
6. Store the contents of the accumulator in memory.

A program written from these steps is given below.

Op Code	Location	Mnemonic	Note
D8	01	CLD	Clear decimal flag
38	02	SEC	Set carry flag high
A9	03	LDA #Oper	Load the accumulator
XXXX XXXX	04	Data	Number to be converted
E9	05	SBC #Oper	Subtract one from the accumulator
01	06	Data	One- the number being subtracted
49	07	EOR #Oper	Exclusive OR accumulator
00	08	Data	number accumulator being EORed with
85	09	STA Oper	Store accumulator in memory
00	0A	Data	memory location where accumulator stored
4C	0B	JMP Oper	Prevent further execution
0B	0C	Data	Location to jump to
00	0D	Data	Page to jump to

Substitute -3 for XXXX XXXX, expressing -3 in negative form (twos complement) or 1111 1101 and load the program into memory. The memory restart vector is at location 01 on page 00. The results of the first program run, as contained in memory location 00, should be 0000 0011.

This program, with slight modifications, can be added to a signed arithmetic program. Some provision must be made to indicate if the results are negative or positive.

DOUBLE PRECISION ARITHMETIC

Introduction:

The procedure for double precision arithmetic is explored.

Discussion:

It is often useful, when working with the computer, to deal with numbers of a greater magnitude than can be handled within eight bits. To do this, two eight bit binary numbers can essentially be treated as one 16 bit number. It is also possible to utilize more than two bytes when it is desired to form even larger numbers.

The numbers 1099 and 9855 can be converted to binary coded decimal form and added.

1099 becomes 0001 0000 1001 1001
9855 becomes 1001 1000 0101 0101
The sum is 1 0000 1001 0101 0100 which is 10,954 decimal.

These BCD numbers can not be directly added in the accumulator because it is capable of working with only eight bits at a time. To get around this, each number is broken up into eight bit segments. The first addition is performed between the eight least significant bits of each number and is stored in memory. If this addition causes the carry flag to be set high, the carry bit will automatically be added to the next set of numbers being added in the accumulator (providing that other operations affecting the status of the carry flag are not performed first). In other words, the Add with Carry instruction automatically causes the contents of the carry flag to be added to any numbers being added in the accumulator.

The programming steps for a BCD double precision addition are given below. The sum is being stored in location 00 and 01 with 00 containing the eight least significant bits while 01 contains the eight most significant bits of the sum. The first number is XXXX $XXXX_2$ XXXX $XXXX_1$ and the second number is designated as YYYY $YYYY_2$ YYYY $YYYY_1$.

1. Set aside locations 00 and 01 for the sum. Load location 02 with 0000 0000 to serve as the carry flag indicator for the last portion of the addition.
2. Set the decimal mode flag.
3. Clear the carry flag.
4. Load the accumulator with XXXX $XXXX_1$.
5. Add YYYY $YYYY_1$ to the accumulator.
6. Store the results in location 00.
7. Load the accumulator with XXXX $XXXX_2$.
8. Add YYYY $YYYY_2$ to the accumulator.
9. Store the results in location 01.
10. If the carry flag is high, go to step 12.
11. Prevent the processor from executing further instructions with a Jump Absolute instruction.
12. Increment memory location 02 to indicate that the carry flag is set high.
13. Prevent further execution of instructions with a Jump Absolute.

The restart vector for the program appearing on the following page is location 03 on page 00. This is because the first executable instruction is located at location 03. Each time the program is run after the first run, care should be taken to load location 02 with 0 so that the status of the carry flag can be determined after the program has been run.

Op Code	Location	Mnemonic	Note
00	02	Data	Carry flag indicator location
F8	03	SED	Set to decimal mode
18	04	CLC	Clear carry flag
A9	05	LDA #Oper	Load accumulator
XXXX $XXXX_1$	06	Data	First half of first number
69	07	ADC #Oper	Add with carry
YYYY $YYYY_1$	08	Data	First half of second number
85	09	STA Oper	Store accumulator in memory
00	0A	Data	Memory location where acc. stored
A9	0B	LDA #Oper	Load Accumulator
XXXX $XXXX_2$	0C	Data	Second half of first number
69	0D	ADC #Oper	Add with carry
YYYY $YYYY_2$	0E	Data	Second half of second number
85	0F	STA Oper	Store results in memory
01	10	Data	Memory location for partial sum
B0	11	BCS	Branch if carry flag is high
03	12	Data	# of addresses to skip if carry set
4C	13	JMP Oper	Prevent further execution
13	14	Data	Memory location to jump to
00	15	Data	Page to jump to
E6	16	INC Oper	Increment memory to indicate carry set
02	17	Data	Memory location to be incremented
4C	18	JMP Oper	Prevent further execution
18	19	Data	Memory location to jump to
00	1A	Data	Page to jump to

Subtraction can also be carried out in double precision. The procedure for double precision subtraction is quite similar to that for single precision subtraction. Prior to subtracting, the carry bit must be set high. However, between subtractions involving two multibyte numbers, the carry bit should not be set or reset.

As with double precision addition, provisions must be made to indicate the final status of the carry flag. If it is 0, the difference is negative and is expressed in twos complement format. If the carry bit is high, the difference is positive and is expressed in twos complement form which requires no conversion (for positive numbers) for binary form.

STACK PROCESSING (SUBROUTINES TO BE COVERED LATER)

Introduction:

The use of the stack for information storage is discussed.

Discussion:

When intermediate results are found during a series of operations, it is often inconvenient to place these results into directly specified memory locations. The programmer must use two to three bytes to retrieve it. In addition to this, he must keep track of all specified memory locations. If the programmer wishes to store information contained in the Processor Status Register, even more valuable memory space is used. He must test for specific conditions such as overflow high, and so forth. Then, if the specific condition is met, he must branch to another part of the program which will leave an indicator in memory. If the programmer later decides to add a single instruction, many program alterations may be needed.

Fortunately, an alternative to the above is available. The Program Stack can be used to hold the contents of the accumulator, the Processor Status Register, and the X and Y Registers. The contents of the stack can then be placed in the accumulator, the Status Processor Register, or either the X Register or the Y Register.

The stack is normally on page 01, but on the Model 300 is on page 00. The stack will begin at the high order memory location on the page chosen by the programmer, or location 7F (FF on pages with a full page of memory). The first information placed on the stack will be placed at location 7F. The next byte placed on the stack will be at 7E. A third word would be stored at location 7D. When information is to be retrieved from the stack, the information most recently stored will be the first to be recalled. If three bytes have been stored, they will be retieved in the following order: contents of 7D, then 7E, and then 7F.

It should be noted that the programmer must take care not to place program instructions in the memory locations which are being used for the stack.

Before the stack is used, the stack pointer must be initialized. The stack pointer is a 16 bit counter containing the memory address for the stack. This is initialized in the following manner. The X Register is loaded with the desired contents of the stack pointer. The Load X immediate instruction is followed directly by Transfer Index X to Stack Pointer command. This makes the contents of the stack pointer register equal to those of Register X. The mnemonic for Loading X with memory in the immediate mode is LDX #Oper and the op code is A2. As with other immediate instructions, this is a two byte instruction with the first byte containing the op code while the second contains the data which is to be placed in Register X. The Transfer Index X to Stack Pointer instruction is a one byte instruction in the implied mode. The op code is 9A and the mnemonic is TXS.

Data is stored in the stack using either the Push Processor Status on Stack or the Push Stack Accumulator on Stack instruction. These two instructions transfer data to the next location on the stack and decrement the stack pointer so that it points to the memory location with an address one less than the one just loaded with data.

The Push Accumulator on Stack instruction places the data contained in the accumulator on the stack. The mnemonic is PHA and the op code is 48. Since this instruction is in the implied mode, it uses only one byte of memory. The implied mode of addressing is also used by the Push Processor Status on Stack instruction. PHP is the mnemonic for the instruction which has an op code of 08.

When information is retrieved from the stack, it is placed directly into the accumulator or into the Process Status Register. Two single byte implied mode instructions are available to "pull" data from the stack. The Pull Accumulator from the stack instruction which has a mnemonic of PLA increments the stack pointer by one and uses the new value to address a location in the stack and load its contents into the accumulator. The op code is 68. The Pull Processor Status from Stack instruction is like the Pull Accumulator from Stack instruction except that it places data in the Process Status Register rather than in the accumulator. It has a mnemonic of PLP and an op code of 28.

These instructions make it possible to store the contents of the Processor Status Register into the stack and then into the accumulator. This gives the programmer access to information about the overflow flag, the carry flag, and so on. The flags of the status register from the most significant bit to the least significant bit are: negative result, overflow, reserved for future expansion, break command, decimal mode, interrupt disable, zero result, and carry. This is important for the begining programmer as it enables him to see what conditions have been created in the Processor Status Register as the result of any specific accumulator operation. One might wish, for example, to see if the results obtained in the accumulator are zero and if the machine is operating in decimal mode. Below, the steps for a program to allow the user to examine the Processor Status Register and the difference after a subtraction is performed are outlined.

1. Initialize the stack pointer by loading X with the location where the stack is to begin. Transfer the value from the X register to the stack pointer.
2. Set decimal mode.
3. Set the carry flag.
4. Load the accumulator with the number which will have number subtracted from it.
5. Subtract
6. Store the difference in location 00
7. Place the contents of the Processor Status Register in the stack.
8. Jump Absolute to prevent further execution

A program to do this follows.

Op Code	Location	Mnemonic	Note
A2	01	LDX #Oper	Load Reg. X with stack pointer address
7F	02	Data	Stack pointer address
9A	03	TXS	Load Register X in stack pointer
F8	04	SED	Set decimal flag
38	05	SEC	Set carry flag
A9	06	LDA #Oper	Load accumulator with first number
XXXX XXXX	07	Data	First number
E9	08	SBC #Oper	Subtract 2nd number from first
YYYY YYYY	09	Data	Second Number
85	0A	STA Oper	Store difference in location 00
00	0B	Data	Location for data storage
08	0C	PHP	Push Register on Stack
4C	0D	JMP Oper	Jump to prevent further execution
0D	0E	Data	Memory location to jump to
00	0F	Data	Page to jump to

Load the program above. Set the reset vector at location 01 on page 00. After running the program, the difference will be stored in location 00. The contents of the Processor Status Register after the subtraction was performed will be contained in memory location 7F which is where the stack has been

loaded. Notice that if one were to place several numbers on the stack, it would be necessary to place the stack below the reset vector. This is necessary on the Model 300 Computer Trainer because with the limited memory, data would soon be stored in the memory at location 7C and at location 7D, erasing the desired reset vector. If it becomes necessary to actually have the stack write in the reset vector locations, care must be taken to reload the reset vector each time that the program has been run.

Initially, run the above program with 77 being subtracted from 77. After the program has been run, 0000 0000 should be contained in location 00. The contents of the Processor Status Register will be contained in 7F. It should contain 00XX 1X1$_0$ where X represents an unknown or "don't care" state. From left to right, the digits represent negative result, overflow, future expansion, break command, decimal mode, interrupt disable, zero result, and carry flag.

The contents of the accumulator can also be placed on the stack. This makes it fairly easy to perform a series of arithmetic or logical operations, store the results and retrieve them after executing another series of instructions which will alter the contents of the accumulator.

The contents of the accumulator can be examined just as the contents of the Processor Status Register are examined in the previous program. This is easily accomplished by changing the contents of location 0C from 08 to 48. In other words, the Push Status Register on Stack command is changed to the Push Accumulator on Stack Instruction.

SUBROUTINES

Introduction:

The use of subroutines and a technique for programming with them on the 6502 processor is discussed.

Discussion:

In long programs, there is frequently a set of instructions which is used repeatedly. Rather than rewrite such a set of instructions numerous times, a subroutine can be used. A subroutine is, essentially, a program within a program. It is also possible to "nest" subroutines, that is, have a subroutine within a subroutine.

The processor instruction set includes a jump to subroutine and a return from subroutine instruction.

The Jump to Subroutine instruction is a three byte instruction in the absolute addressing mode with the first byte containing the op code, the second byte containing the memory location of the first instruction of the subroutine, and the third byte indicating the page on which the subroutine begins. This command causes the processor to begin executing instructions at the location specified by the second and third bytes of the instruction. It also loads the address of the third byte of the Jump to Subroutine instruction on the stack. This will make it possible for the program to "return" to the memory location immediately following the jump when the subroutine has been completed. The instructions immediately following the Jump to Subroutine instruction will be carried out when a Return from Subroutine instruction has been executed. The mnemonic for the Jump to Subroutine command is JSR and the op code is 20.

The Return from Subroutine instruction has RTS as its mnemonic and has an op code of 60. The command is in the implied mode and uses one byte of memory. This command causes the processor to return to the memory location immediately following the Jump to Subroutine instruction which caused the processor to begin executing the subroutine. When this command is executed, the stack pointer will be incremented by two. Keep in mind that the stack was decremented by two when the original Jump to Subroutine instruction was performed. Two bytes were loaded with information: the first was loaded with the memory location of the last byte of the Jump to Subroutine instruction and the second was loaded with the page.

One of several methods for performing multiplication on the microprocessor involves successive addition. This technique can use a subroutine. An example of a procedure for having the microprocessor do this is covered below.

1. Clear the carry flag.
2. Set the decimal mode flag.
3. Initialize the stack. Load X immediate with starting location which is 7F. Then, transfer the contents of X to the stack pointer.
4. Load the accumulator with the first number, XXXX XXXX.
5. Load the Index Register X with the contents of the accumulator.
6. Load Index Register Y with the second number, YYYY YYYY.
7. Decrement Register Y.
8. If Y $\neq$ 0, Jump to the subroutine
9. Store the accumulator in memory
10. Jump Absolute to prevent further execution.

THE SUBROUTINE

11. Add the contents of Register X to the Accumulator.
12. Decrement Index Register Y.
13. If Register Y ≠ 0, go to step 11
14. Return from subroutine.

Notice that a program to do this can easily be written without the use of a subroutine. The subroutine is used here to demonstrate the procedure used for programming with subroutines.

The program on the following page is based on the procedure outlined above. The program itself begins at location 02. The subroutine starts at location 50.

Load the program into memory. Set the reset vector for location 02 on page 00. Before running the program a second time, check to see that the restart vector is still properly loaded. If it isn't, interference between it and the stack may have occured. In this case, reload it before running the program again.

Use the program above to multiply any two numbers where the product isn't above 199. Also note that the program has no provision for multiplying by zero.

Generally, subroutines are used in situations where one set of instructions is repeated so many times that it is impractical to write them each time. A program that proves to be useful as a portion of several different programs can also be used as a subroutine.

Op Code	Location	Mnemonic	Note
18	02	CLC	Clear carry flag
38	03	SED	Set decimal bit
A2	04	LDX #Oper	Load X with stack location
7F	05	Data	Location for the stack
9A	06	TXS	Load X in stack pointer to complete stack initialization
A9	07	LDA #Oper	Load Accumulator with first number
XXXX XXXX	08	Data	First Number
85	09	STA Oper	Store accumulator in memory
01	0A	Data	Memory location
A0	0B	LDY #Oper	Load Register Y with 2nd number
YYYY YYYY	0C	Data	Second number
88	0D	DEY	Decrement Register Y
F0	0E	BEQ	Branch if Y is 0
03	0F	Data	# of instructions to skip
20	10	JSR	Jump to Subroutine
50	11	Data	Memory location
00	12	Data	Page
85	13	STA Oper	Store accumulator (product)
00	14	Data	Memory location
4C	15	JMP Oper	Jump Absolute to prevent further execution
15	16	Data	Memory location to jump to
00	17	Data	Page to jump to
NOTE FOLLOWING MEMORY LOCATION			
65	50	ADC Oper	Add memory to accumulator
01	51	Data	Memory location added to accumulator
88	52	DEY	Decrement Register Y
F0	53	BEQ	If Register Y=0, go to location 58
03	54	Data	Number of locations to skip
4C	55	JMP Oper	Jump to 50
50	56	Data	Memory location to jump to
00	57	D ata	Page to jump to
60	58	RTS	Return from Subroutine

FLOWCHARTING

Introduction:

The use of flowcharts for programming is introduced.

Discussion:

In previous sections, the steps needed in a program have been written out in step by step fashion. This method can become rather tedious and somewhat confusing, especially for programs with several conditional branches. Flowcharting, a technique that is frequently used to represent a sequence of steps, makes it much easier to visualize what the program will do.

In flowcharting, each instruction or step is placed in one "box." For the purposes of this manual, there are five types of boxes. When used in a flowchart, each box will contain the mnemonic or a description of the instruction it represents.

Each instruction which indicates an operation which involves no decision and does not involve input/output (I/O) is placed in a rectangular box. ()

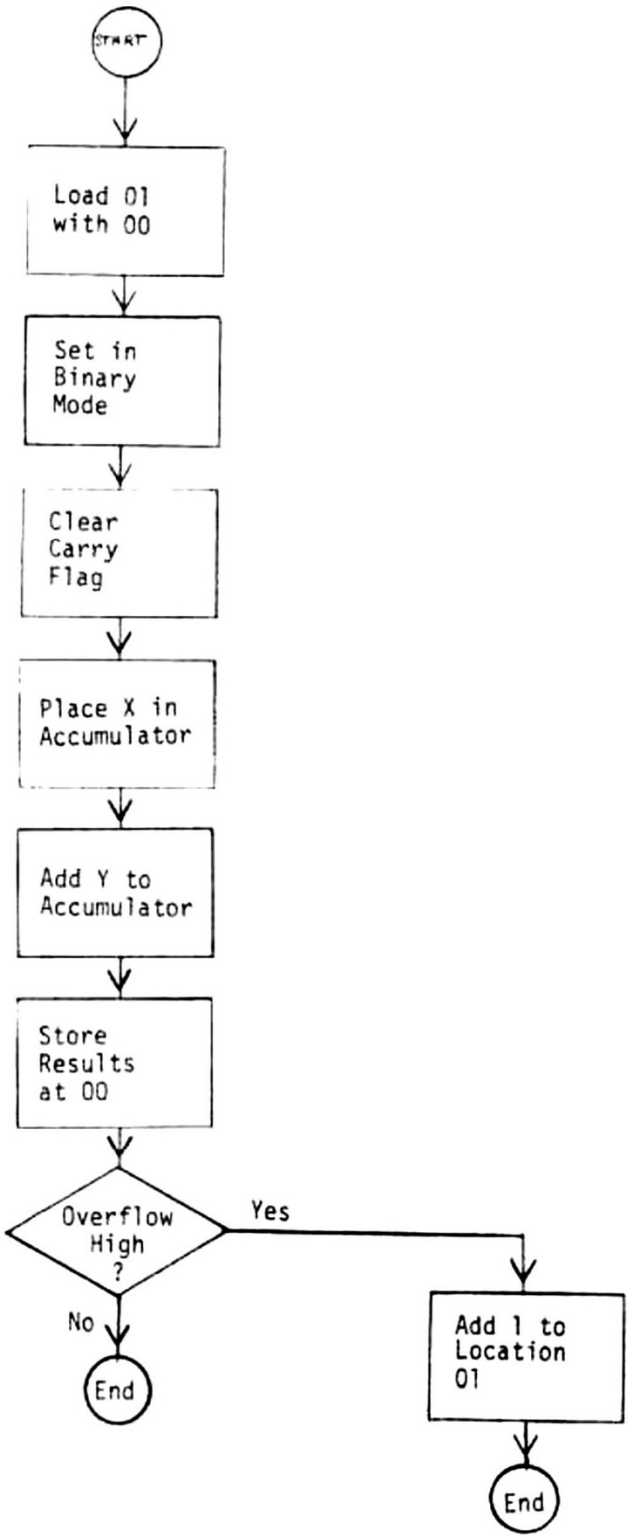
A command involving operations involving a decision are placed in a diamond shaped box. () On the 650X, instructions of this type are the conditional branch instructions. The processor must "decide" if the condition on which it is going to jump is true or false.

When output to a peripheral (not the LEDs on the Model 300) is involved, a box which has a curved base is used. ()

Situations where the processor is accepting data from an I/O device are indicated by a rectangle with a small diagonal slice removed from the upper left-hand corner. ()

To indicate the begining of a flowchart, a circle with "start" within it is used. () The end of a flowchart is shown by a circle containing the word "end." () In this manual, the end box will be used to represent the Jump Absolute instruction where it is used to prevent the processor from executing instructions in locations beyond the program.

The boxes within a flowchart are connected by lines. An arrow is placed between each box indicating the direction of flow. To help illustrate how a flowchart is formed, a flowchart which could be used to write a program for adding two signed numbers is give. This flowchart could have been written for a previous section in which such a program is executed.


START
Load 01 with 00
Set in Binary Mode
Clear Carry Flag
Place X in Accumulator
Add Y to Accumulator
Store Results at 00
Overflow High ?
Yes
No
End
Add 1 to Location 01
End

MULTIPLYING TWO FOUR BIT NUMBERS

Introduction:

Another procedure for multiplying two numbers is examined.

Discussion:

One technique for multiplying binary numbers has already been examined: successive addition. Another method quite similar to that normally used in base 10 arithmetic can be used. The numbers are first written in standard binary form. In this case, 6 X 9 will be multiplied to demonstrate the technique.

```
  0 1 1 0
  1 0 0 1
```

The next step is to multiply the top-most number by the right-hand most digit of the lower number.

```
  0 1 1 0
  1 0 0 1
  0 1 1 0
```

When this value has been found, the top number is multiplied by the digit next to the right-hand most digit of the lower number. The partial product is written under the previous partial product with the least significant digit directly under the second digit (from the right) of the first partial product.

```
    0 1 1 0
    1 0 0 1
  ---------
    0 1 1 0
  0 0 0 0
```

The same procedure is used for the next two digits of the bottom number. Then the partial products are added.

```
      0 1 0 0
      1 0 0 1
  -----------
      0 1 1 0
    0 0 0 0
  0 0 0 0
0 1 1 0
-------------
0 1 1 0 1 1 0
```

To "check" the solution, it can be converted into decimal form:

0110110 equals

$$0(2)^6 + 1(2)^5 + 1(2)^4 + 1(2)^3 + 0(2)^2 + 0\ (2)^1 + 0(2)^0$$
$$= 64 + 0 + 16 + 8 + 0 + 0 + 0 = 88$$

Note that each time a partial product is found, the product is either zero or it is identical to the top line. This fact can be used in programming the microprocessor to perform multiplication.

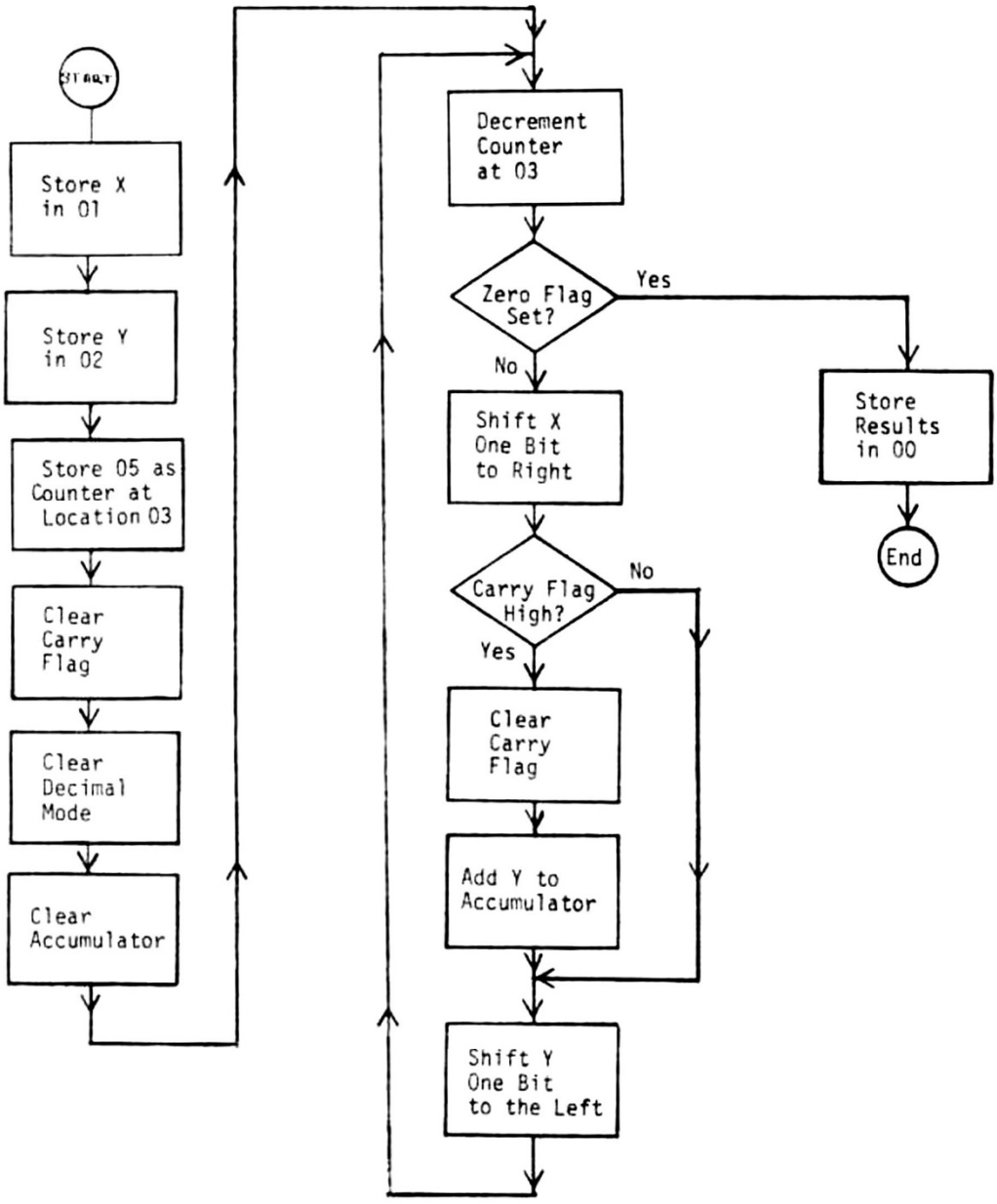
It should also be observed that each partial product, instead of being placed directly under the partial product above it, has been shifted one digit to the left. This must be done prior to adding the partial products together. An operation called a shift memory or accumulator left one bit permits this to be done in the microprocessor. (It is also possible to shift the contents of the accumulator or memory to the right one bit.) When shifting the contents of the accumulator or memory one bit to the right or left, one bit is "pushed out" of the eight bit memory location or the eight

bit accumulator. It is then stored as the carry bit. This proves to be very valuable for multiplication. The bottom number (of the two which are being multiplied) can be shifted, one bit at a time, to the right. After each shift, the carry bit can be examined to determine its status. If it is high, the top number will be used as a partial product for the digit which was just placed in the carry bit portion and the top number. If the carry bit is low, the resulting partial product is zero.

The flow chart on the following page shows steps which can be used to multiply two numbers. Memory location 00 will be used to store the results. Locations 01 and 02 contain the numbers which are to be multiplied together. Location 03 holds a counter. Each time the program has dealt with one partial product, the counter will be decremented. When all four partial products have been found, the counter will have been decremented to zero. At this point, the sum of the partial products will be stored at location 00.

The flow chart on the following page can be easily translated into the following program.

Op Code	Location	Mnemonic	Note
XXXX XXXX	01	Data	First number
YYYY YYYY	02	Data	Second Number
05	03	Data	Counter
18	04	CLC	Clear carry flag
D8	05	CLD	Place in binary mode
A9	06	LDA #Oper	Load accumulator with zero
00	07	Data	0 being loaded into accumulator
C6	08	DEC Oper	Decrement counter
03	09	Data	Location of counter
F0	0A	BEQ	Branch if counter reaches 0
0B	0B	Data	How far to skip if counter = 0
46	0C	LSR Oper	Shift X one bit to the right.
01	0D	Data	Memory location of X
90	0E	BCC	Branch if carry bit is low
03	0F	Data	How far to skip if carry bit low
18	10	CLC	Clear carry bit before addition
65	11	ADC Oper	Add Y to accumulator
02	12	Data	Location of Y
06	13	ASL Oper	Shift Y one bit to the left
4C	14	JMP Oper	Go back to location 08
08	15	Data	Memory location to jump to
00	16	Data	Page to jump to
85	17	STA Oper	Store final product
00	18	Data	Memory location 00
4C	19	JMP Oper	Jump Absolute to prevent further execution
19	1A	Data	Memory location to jump to
00	1B	Data	Page to jump to


START
Store X in 01
Store Y in 02
Store 05 as Counter at Location 03
Clear Carry Flag
Clear Decimal Mode
Clear Accumulator
Decrement Counter at 03
Zero Flag Set?
Yes
No
Store Results in 00
End
Shift X One Bit to Right
Carry Flag High?
No
Yes
Clear Carry Flag
Add Y to Accumulator
Shift Y One Bit to the Left

Load the program. The restart vector should be loaded with locatin 04 on page 00. The first time the program is run, let the first number be 5 and the second be 7.

Notice how the shift instructions have been used in the program above. They can also be used directly in the multiplication and division process. Shifting any binary number once to the right is the same as dividing the number by two, just as shifting any base 10 number once to the right is the same as dividing by 10. Shifting a binary number once to the left is the same as multiplying it by two. Shifting twice to the left is multiplying by four and shifting three times to the left is multiplying by eight. This information is quite useful when one wishes to multiply a series of numbers by a constant.

COMPARE

Introduction:

The use of the Compare instruction for determining conditions of inequality and equality is described.

Discussion:

At times, it is desirable to determine the relationship between two numbers, i.e., greater than, less than, and so on.

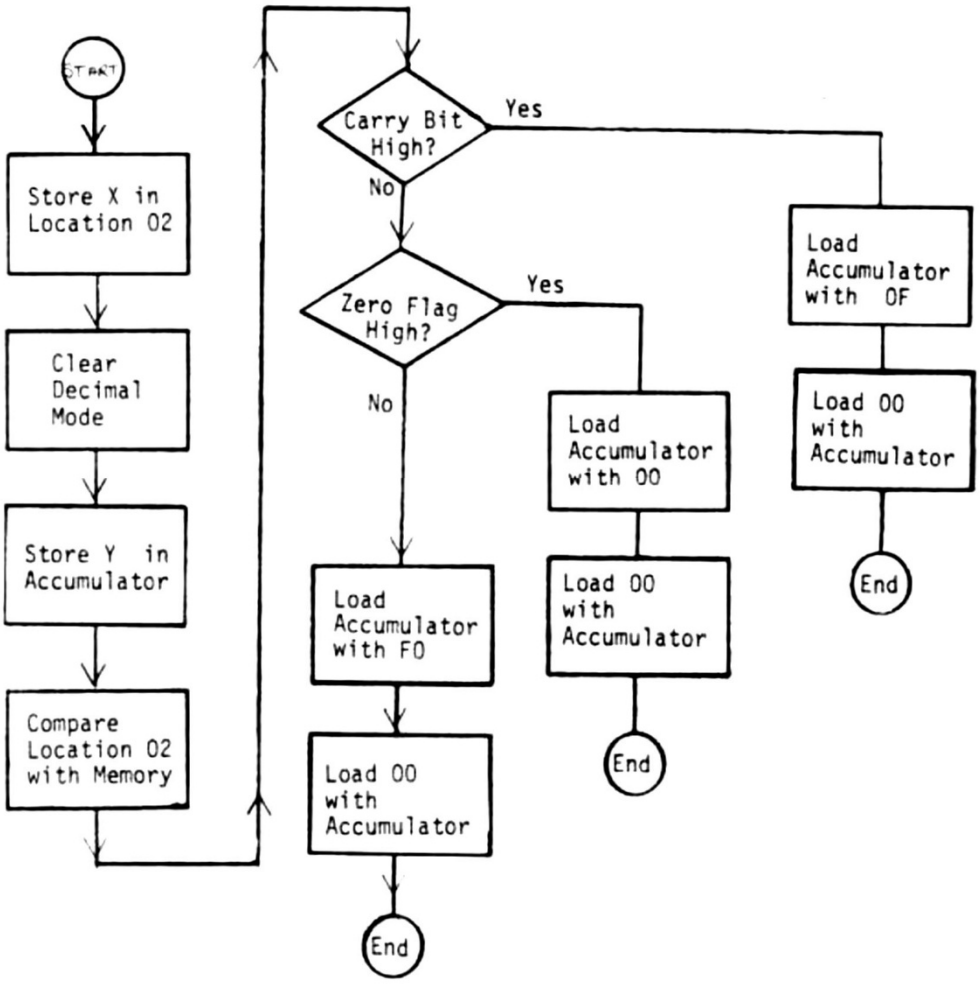
This can be accomplished through the use of the Compare instruction on the 650X microprocessors. The contents of a specified memory location are subtracted from the contents of the accumulator by the Compare command. While the results of this subtraction are not stored in the processor, they do affect the Processor Status Register. The results of the compare operation are indicated in this register as follows:

	negative bit	carry	Zero	Overflow
Accumulator less than memory	X	0	0	Not affected
Accumulator equals memory	0	1	1	Not affected
Accumulator greater than memory	X	1	0	Not affected

The flow chart on the following page can be used to write a computer program which can be used to determine the relationship between two numbers. The outcomes will be stored in location 00 with 0000 1111 indicating that the accumulator is less than the memory, 0000 0000 indicating that the accumulator and memory contain equal values, and 1111 0000 indicating that the accumulator is greater than the memory.

The program below can be written from the flowchart.

Op Code	Location	Mnemonic	Note
XXXX XXXX	01	Data	First number to be compared
D8	02	CLD	Clear decimal mode
A9	03	LDA #Oper	Load Accumulator
YYYY YYYY	04	Data	2nd number to be compared
C5	05	CMP	Compare accumulator and memory
01	06	Data	Memory location
B0	07	BCS	Branch if carry high
09	08	Data	Number of instructions to branch
F0	09	BEQ	Branch if zero bit is high
14	0A	Data	Number of instructions to branch
A9	0B	LDA #Oper	Load Accumulator with number for memory
F0	0C	Data	Number to be stored at 00
85	0D	STA Oper	Store accumulator in memory
00	0E	Data	Memory location
4C	0F	JMP Oper	Jump to prevent further execution
0F	10	Data	Memory location to jump to
00	11	Data	Page to jump to
A9	12	LDA #Oper	Load accumulator with number for memory
0F	13	Data	Number to be stored at 00
85	14	STA Oper	Store accumulator in memory
00	15	Data	Memory location
4C	16	JMP Oper	Jump to prevent further execution
16	17	Data	Location to jump to
00	18	Data	Page to jump to


START
Store X in Location 02
Clear Decimal Mode
Store Y in Accumulator
Compare Location 02 with Memory
Carry Bit High?
Yes
No
Zero Flag High?
Yes
No
Load Accumulator with OF
Load 00 with Accumulator
End
Load Accumulator with 00
Load 00 with Accumulator
End
Load Accumulator with F0
Load 00 with Accumulator
End

Op Code	Location	Mnemonic	Note
A9	19	LDA #Oper	Load accumulator with number
00	1A	Data	Number to be stored at 00
85	1B	STA Oper	Store accumulator in memory
00	1C	Data	Memory location
4C	1D	JMP Oper	Jump to prevent further execution
1D	1E	Data	Location to jump to
00	1F	Data	Page to jump to

Load the program given above and on the first page. The restart vector for the program above is location 02 on page 00. To fully test the program, each possible condition should be tested for. In one case, the first and second number should be equal. In a second case, the first should be greater than the second, and in the third case, the first number should be less than the second.

If the programmer wishes to test for a greater than or equal to condition (where the accumulator is greater than or equal to the memory), he will test to see if the accumulator has a value less than that of the specified memory location. If it doesn't, a greater than or equal to condition exists. After performing a compare between the accumulator and memory, this condition will be tested by checking the carry flag. If the carry flag is high, a greater than or equal to relationship exists.

Testing for a less than or equal to condition is accomplished by testing to determine if the value stored in the accumulator is greater than that of a specified memory location. If it is, the value in the accumulator is less than or equal to that in the memory location. After a compare is executed between the accumulator and specified memory location, the less than ot equal to relationship is tested by checking carry and zero bits. If the carry bit is high while the zero bit is low, the less than or equal to relationship <u>does not exist</u>.

ADDRESSING MODES

Introduction:

Various addressing modes available on the 6502 microprocessor are described.

Discussion:

In previous discussions, the existence of more than one addressing mode for various instructions has been mentioned. The addressing modes are implied, immediate, absolute, zero page, relative, absolute indexed, zero page indexed, indexed indirect, and indirect indexed.

An instruction using the implied addressing mode is expressed in one byte. Commands using this mode of addressing are dealing with flag bits, the accumulator, or the X or Y Registers, but do not refer to any memory location.

In addition to the accumulator, the microprocessor has two registers: X and Y. These, like the accumulator, are not assigned a memory address location. One byte instructions can be used to transfer information between these registers and the accumulator. One byte instructions are also available for the X and Y registers to increment and decrement their contents.

All instructions involving setting or clearing a CPU status register flag use the implied mode.

A NOP or No Operation instruction is also in the implied mode. This instruction is used by the programmer when he wishes to be able to place additional steps in a program at a later date without changing the memory locations of other instructions. It can also be used when a programming step needs to be deleted and a change in memory locations for other commands is undesirable.

Instructions in the immediate mode require two bytes. The first contains the op code for the instruction and the second consists of the data which is being acted upon. For example, a load accumulator immediate instruction will devote the first byte to the op code. The second byte contains the information that is to be loaded into the accumulator. This addressing mode is convenient to use when one knows the exact value of the data being dealt with. It is also quite efficient in terms of memory usage because it uses only two bytes of memory and does not require that further information be stored at another memory location.

Types of instructions which can be used in the immediate mode are logical operations, arithmetic operations, compare instructions, load accumulator or the X or Y Register.

When an instruction is given in the absolute mode, the memory and page location of the data which is to be acted upon is specified. The instruction is three bytes long with the first byte indicating the op code, the second showing the memory location, and the third indicating the memory page. Instructions available in the absolute mode are those involving arithmetic operations, rotating or shifting data in a memory location left or right, memory bit testing, compare operations, incrementing or decrementing memory, logical operands, unconditional jump, and storing the accumulator or a register in memory.

The zero page addressing mode is very similar to the absolute mode except that it is only two bytes long. As with the absolute mode, the first byte is the op code and the second byte is the memory location. When the zero mode is used, no third byte is needed as the page location is automatically 00. The advantage of the zero page addressing over the absolute addressing is that it uses one less word of memory and it is somewhat faster. It does, however, limit the user to storing data on page 00. (It should be noted that all of the memory on the Model 300 Computer Trainer is located on page 00.)

Instructions which can be used in the zero page addressing mode are arithmetic operations, logical operations, memory bit testing, compare operations, memory increment and decrement, loading the accumulator or Register X or Y from memory, rotating bits left or right, and storing the accumulator or a register in memory.

Relative addressing is used with conditional branch instructions. Instructions using this mode require two bytes. The first contains the op code. The second byte contains the number of memory locations that will be skipped if the branch condition is met. This addressing mode enables the programmer to write a program without needing to decide where it will be located until it is entered into the machine. He can even add or delete instructions with the only changes being made, other than the addition or deletion, in the second byte of any conditional branch instruction. This mode of addressing is used only with conditional branch instructions and is the only addressing mode used by these instructions.

It should be noted that as well as enabling the processor to conditionally "jump ahead" in memory, the relative addressing mode's instructions also make it possible to branch backwards in memory.

Absolute indexed addressing is the same as absolute addressing except that the contents of either Register X or Register Y are added to the absolute address before the operation is performed.

Instructions which can be used in the absolute indexed addressing mode using Register X are; addition, AND, shift left one bit memory, compare memory and accumulator, exclusive OR, load accumulator from memory, increment memory, decrement memory, load Y with memory, shift memory right one bit, OR memory with accumulator, rotate memory one bit left, subtract memory from accumulator, and store accumulator in memory.

Using Register Y in the absolute indexed addressing mode can be accomplished for add with carry, AND, compare, exclusive OR, load accumulator with memory, load Register X from memory, OR, subtract with carry, and store accumulator in memory.

Zero page indexed addressing is identical to absolute indexed addressing except that the third instruction byte is omitted as the page location is automatically 00. Instructions which can be used in this mode are add and subtract, logical operands, increment and decrement, store accumulator in memory, store Register Y in memory, compare memory with accumulator, rotate one bit left, shift right one bit, and shift left one bit.

Indexed indirect addressing is primarily used to obtain data from tables which the programmer has stored in memory. Instructions in this mode use two bytes of memory, the first being the op code. The second byte of the instruction is added to the contents of Register X. The resulting sum is treated as a memory location on page 00. The contents of this location will be used as the memory location of the actual data. The sum mentioned earlier will be incremented and treated as the memory location containing the page number on which the actual data appears.

Due to the memory and register space needed to utilize this addressing mode, it is very impractical to use it with the Model 300 Computer Trainer. It also takes more time to execute the instruction than it does to perform an instruction in the absolute mode. This type of addressing can be used with the following instructions; logical operands, addition and subtraction, compare accumulator with memory, load accumulator from memory and store accumulator in memory.

Indirect indexed addressing uses two bytes of memory. The first contains the op code. The second contains a page 00 address. The contents of this address are added to the contents of Register Y. The results indicate the memory location of the actual data which the instruction will act upon. The address contained in the second byte of the instruction is incremented to obtain a page 00 address. The contents of this address indicate the page on which the data used by the instruction is contained.

Instructions which can use this addressing mode are add with carry, logical operands, subtract with borrow, compare memory with accumulator, load accumulator with memory, and store accumulator in memory.

This form of addressing is useful when one of several values could be used in a subroutine (program within a program). It is not practical to use it in the Model 300 Computer Trainer because of the memory space restrictions which are placed on the programmer.

Using the op code listing in Appendix A, write a short program for each addressing mode which moves some memory location into location 10. The routine for zero page mode is given as an example.

Op Code	Location	Mnemonic	Note
A9	00	LDA #Oper	Load accumulator with data to be stored
FF	01	Data	Data to be stored
85	02	STA Oper	Store accumulator (zero page)
10	03	Data	Location on page 00 to store data
4C	04	JMP Oper	Jump to prevent further execution
04	05	Data	Memory location to jump to
00	06	Data	Page to jump to

INTERRUPTS

Introduction:

The use of interrupts is discussed.

Discussion:

Interrupts are used to literally "get the processor's attention" when it is running a program. Ideally, an interrupt causes the processor to temporarily stop executing a program and jump to a new routine. At the completion of this routine, the processor returns to the original program and continues execution. Interrupts are used mainly inconjunction with I/O devices (Input/Output). Each I/O device of a large computer system generates interrupts when it wants to input data to the processor and, in many cases, interrupts are generated when I/O devices are ready to accept data.

The 6502 has two external interrupts. These are a non-maskable or unconditional interrupt and a maskable interrupt which is controlled by D_2 of the Processor Status Register. The (NMI) line is controlled by the NMI switch on the Model 300. When it is to the left, the NMI line is not activated. When it is moved to the right, the non-maskable interrupt is activated. The NMI line is edge triggered so that the switch may be left to the right without ill effects during program execution.

When the NMI switch is activated, the existing program is stopped. The program counter and Processor Status Register are pushed on to the stack and the processor jumps to a new program specified by the NMI vector located at FFFA and FFFB (7A and 7B on the Model 300). If one wishes to completely save the original program, the accumulator and registers X and Y should be stored on the stack immediately in the NMI program. At the end of the interrupting routine, X, Y and the accumulator should be pulled from the stack before the execution of the Return from Interrupt instruction which will then completely restore the original program.

The maskable interrupt differs from NMI in that it can be disabled by D_2 of the Processor Status Register and its vector is located at FFFE and FFFF (7E and 7F on the Model 300). The line associated with the maskable interrupt is called IRQ (interrupt request) and is available for external use at the left of the processor on the Model 300. Its use is discussed in the next section.

NMI can be demonstrated. A simple program which increments memory location 00 each time the NMI switch is activated is outlined below. It is actually two programs; the mainline routine and the interrupt routine. Be sure to set both vectors when executing the program.

Mainline Program

1. Start at location 01.
2. Initialize stack pointer.
3. Jump to start of program.

Interrupting Routine

1. Increment memory location 00.
2. Return from interrupt.

I/O PROGRAMMING

Introduction:

The I/O capabilities of the Model 300 Computer Trainer are explored.

Discussion:

As mentioned in the last section, the Computer Trainer has the IRQ line available for external input. This line is pulled up via a 4.7K resistor. Momentarily bringing this line low inconjunction with D_2 of the Processor Status Register being low will cause an interrupt. This line is not edge triggered so holding this line low for more than approximately 10useconds will cause multiple interrupts and will, most likely, cause the stack to overwrite the program.

Another input can be provided on the Computer Trainer by removing the jumper from pin 38 to ground (located directly above the processor). Pin 38 is set overflow which sets the overflow bit in the Processor Status Register when high. For this reason, it should normally be low, but, it can be used as a very simple one line input.

The Computer Trainer also has a one bit output latch. This latch is set whenever a location on page 02 is addressed and is reset whenever a location on page 01 is addressed. It has both a high true output (to the left of the output label and a low true output (to the right). A high frequency square wave can be obtained from this output by using the following program.

Op Code	Location	Mnemonic	Note
AD	00	LDA Oper	Accumulator loaded in absolute mode
00	01	Data	so that page 02 can be addressed,
02	02	Data	setting the latch high
ADD NOPS HERE LATER (SEE TEXT)			
4C	03	JMP Oper	Dummy jump designed to balance time
06	04	Data	
00	05	Data	
AD	06	LDA Oper	Accumulator loaded in absolute mode
00	07	Data	so that page 01 can be addressed,
01	08	Data	setting the latch low
ADD NOPS HERE LATER (SEE TEXT)			
4C	09	JMP Oper	Jump back to the begining of the
00	0A	Data	program
00	0B	Data	

This loop takes 14 machine cycles to execute. The Model 300 has a machine cycle time of two to four microseconds so the output period will be 28 to 56usec. or 20 to 40KHz . By measuring the output period with a scope or frequency meter and dividing by 14, the exact cycle time can be found.

By adding 10 NOPs at the locations specified in the preceding program, the loop will require 54 cycles to execute, producing an output in the audio range. The output latch can be connected directly to the auxilary or Tuner input of a standard audio amplifier. Be careful to install the ground connection before installing the signal connection.

More sophisticated timing loops can be used to generate specific audio notes as outlined on the following page.

1. Set the output latch high.
2. Load a memory location into Register X.
3. Decrement Register X until it is zero and then proceed.
4. Set the output latch low.
5. Repeat step 2.
6. Repeat step 3.
7. Jump to the begining of the program.

With this loop, the output frequency is determined by a specific memory location. By placing this loop inside another timing loop, the duration of the note or tone can be specified. This can then be used inconjunction with a look-up table in memory to play music or with a random number routine to generate "music."

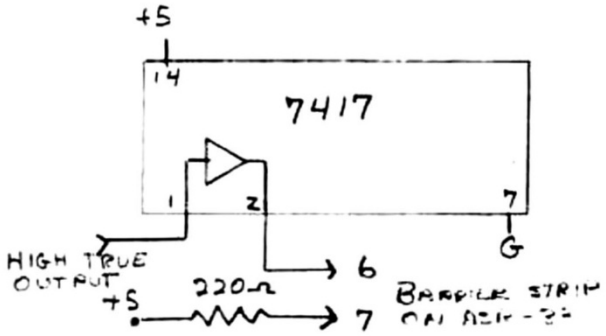
OUTPUTING TO A TELETYPE

Introduction:

The basics of outputing via a 20ma current loop are discussed.

Discussion:

The standard computer terminal uses a one line serial communications technique which is either a 20ma current loop or RS-232C specification. The most common terminal, the ASR-33 Teletype, uses a 20ma current loop. The Model 300 Computer Trainer's output latch can be easily interfaced to an ASR-33 via the diagram below.



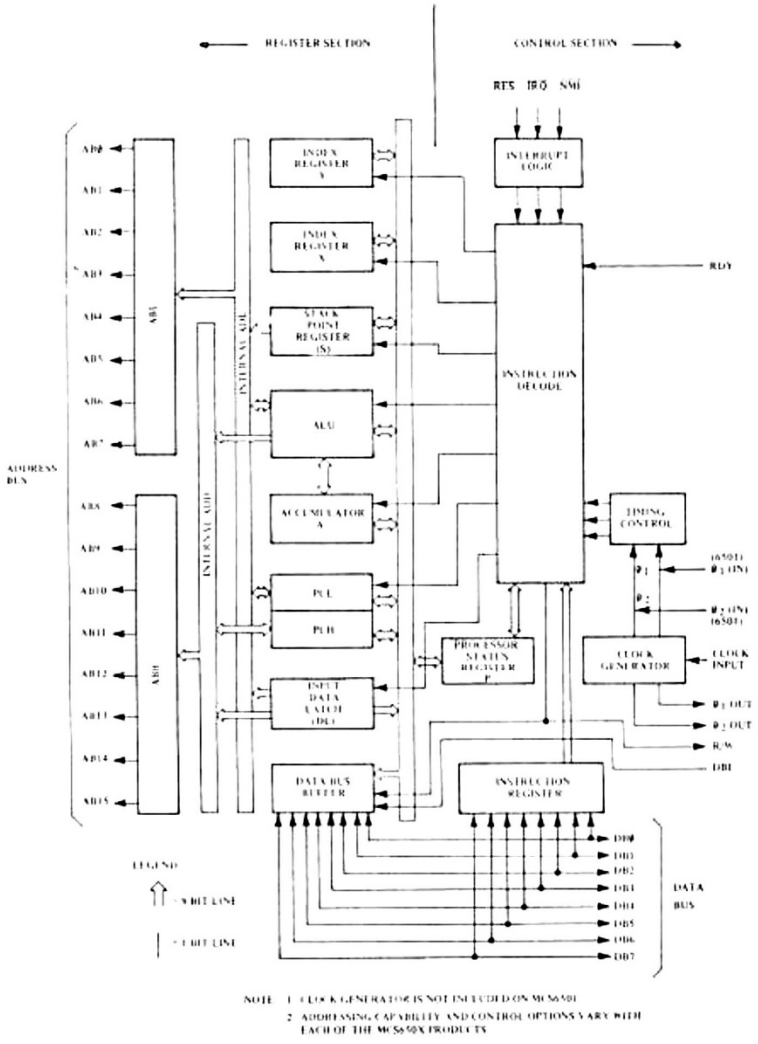
The software for outputing a character on the teletype is very similar to the music routines of the last experiment. The teletype communicates at 110 baud or 10 characters per second. Each character is made up of a mark, eight data periods, and a double length mark so that there are a total of 11 periods per character. Each period is 9090usec. long. The marks are ones or current on conditions. The zeros or offs are called spaces. The eight bit serial code transmitted between marks determines the character which will be printed on the teletype. This code is called ASCII and is listed in hex•decimal form on the 650X programming card.

To print an "A" on a teletype, for example, the software must output a mark, then 0100 0001, then a two period mark with each period being 9090usec. long. As with the music program, routines should be designed to use data from memory so that words and sentences can be outputed.

Comments on the Data Sheet

This data sheet describes the first five members of the MCS650X microprocessor family. The data sheet is constructed to review first the basic "Common Characteristics" - those features which, unless specifically stated otherwise, are common to all of the MCS6501 - MCS6505 microprocessors. Subsequent to a review of the family characteristics will be sections devoted to each member of the group with specific features of each.

COMMON CHARACTERISTICS



NOTE 1. CLOCK GENERATOR IS NOT INCLUDED ON MCS6501
2. ADDRESSING CAPABILITY AND CONTROL OPTIONS VARY WITH EACH OF THE MCS650X PRODUCTS

MCS6501 – MCS6505 Internal Architecture

COMMON CHARACTERISTICS

INSTRUCTION SET – ALPHABETIC SEQUENCE

ADC Add Memory to Accumulator with Carry	DEC Decrement Memory by One	PHA Push Accumulator on Stack
AND "AND" Memory with Accumulator	DEX Decrement Index X by One	PHP Push Processor Status on Stack
ASL Shift Left One Bit (Memory or Accumulator)	DEY Decrement Index Y by One	PLA Pull Accumulator from Stack
		PLP Pull Processor Status from Stack
BCC Branch on Carry Clear	EOR "Exclusive-or" Memory with Accumulator	
BCS Branch on Carry Set		ROL Rotate One Bit Left (Memory or Accumulator)
BEQ Branch on Result Zero	INC Increment Memory by One	ROR Rotate One Bit Right (Memory or Accumulator)
BIT Test Bits in Memory with Accumulator	INX Increment Index X by One	RTI Return from Interrupt
BMI Branch on Result Minus	INY Increment Index Y by One	RTS Return from Subroutine
BNE Branch on Result not Zero		
BPL Branch on Result Plus	JMP Jump to New Location	SBC Subtract Memory from Accumulator with Borrow
BRK Force Break	JSR Jump to New Location Saving Return Address	SEC Set Carry Flag
BVC Branch on Overflow Clear		SED Set Decimal Mode
BVS Branch on Overflow Set	LDA Load Accumulator with Memory	SEI Set Interrupt Disable Status
	LDX Load Index X with Memory	STA Store Accumulator in Memory
CLC Clear Carry Flag	LDY Load Index Y with Memory	STX Store Index X in Memory
CLD Clear Decimal Mode	LSR Shift One Bit Right (Memory or Accumulator)	STY Store Index Y in Memory
CLI Clear Interrupt Disable Bit		
CLV Clear Overflow Flag	NOP No Operation	TAX Transfer Accumulator to Index X
CMP Compare Memory and Accumulator		TAY Transfer Accumulator to Index Y
CPX Compare Memory and Index X	ORA "OR Memory with Accumulator	TSX Transfer Stack Pointer to Index X
CPY Compare Memory and Index Y		TXA Transfer Index X to Accumulator
		TXS Transfer Index X to Stack Pointer
		TYA Transfer Index Y to Accumulator

ADDRESSING MODES

ACCUMULATOR ADDRESSING - This form of addressing is represented with a one byte instruction, implying an operation on the accumulator.

IMMEDIATE ADDRESSING - In immediate addressing, the operand is contained in the second byte of the instruction, with no further memory addressing required.

ABSOLUTE ADDRESSING - In absolute addressing, the second byte of the instruction specifies the eight low order bits of the effective address while the third byte specifies the eight high order bits. Thus, the absolute addressing mode allows access to the entire 65K bytes of addressable memory.

ZERO PAGE ADDRESSING - The zero page instructions allow for shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high address byte. Careful use of the zero page can result in significant increase in code efficiency.

INDEXED ZERO PAGE ADDRESSING - (X, Y indexing) - This form of addressing is used in conjunction with the index register and is referred to as "Zero Page, X" or "Zero Page, Y". The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally due to the "Zero Page" addressing nature of this mode, no carry is added to the high order 8 bits of memory and crossing of page boundaries does not occur.

INDEXED ABSOLUTE ADDRESSING - (X, Y indexing) - This form of addressing is used in conjunction with X and Y index register and is referred to as "Absolute, X", and "Absolute, Y". The effective address is formed by adding the contents of X or Y to the address contained in the second and third bytes of the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

IMPLIED ADDRESSING - In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

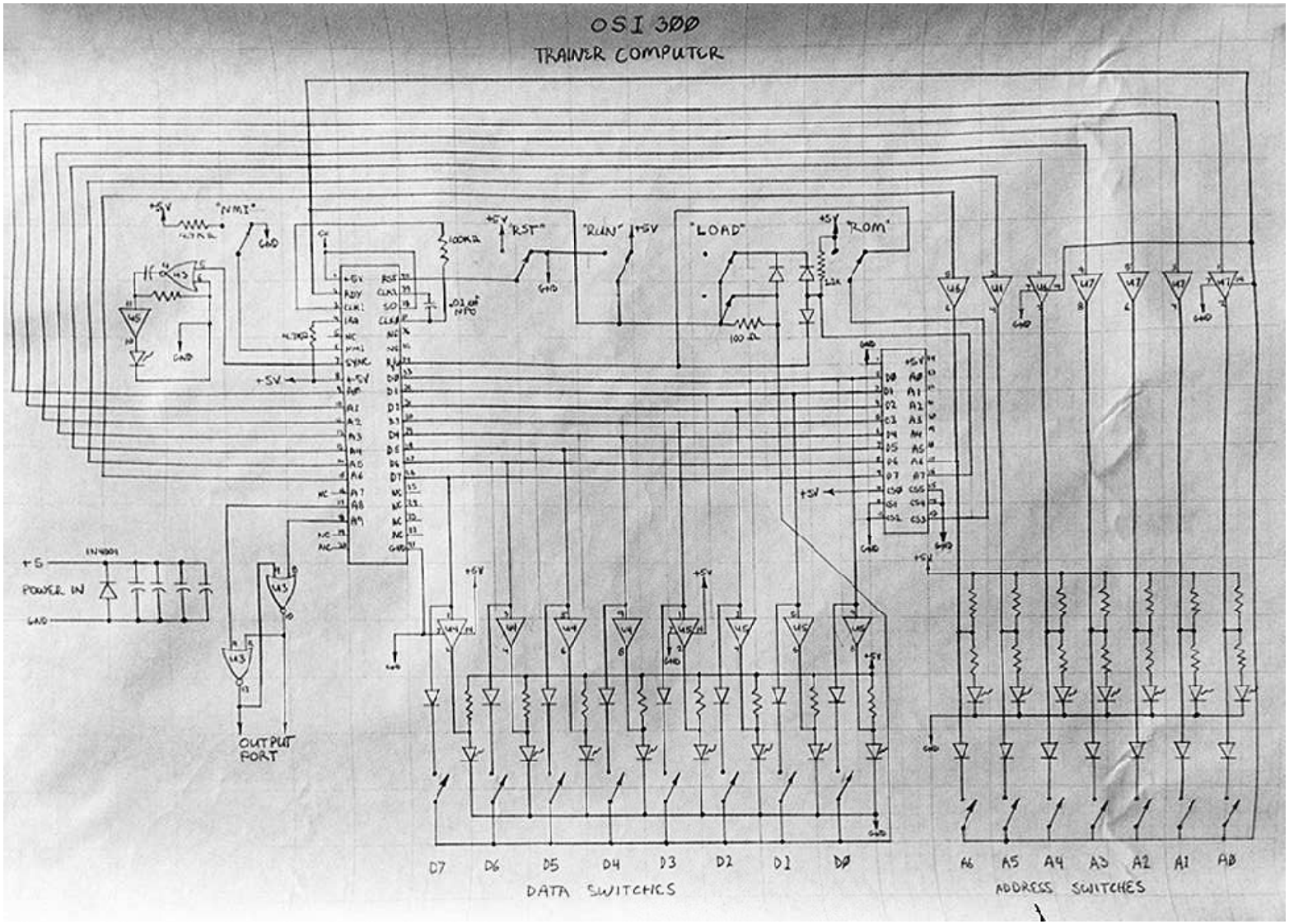
RELATIVE ADDRESSING - Relative addressing is used only with branch instructions and establishes a destination for the conditional branch.

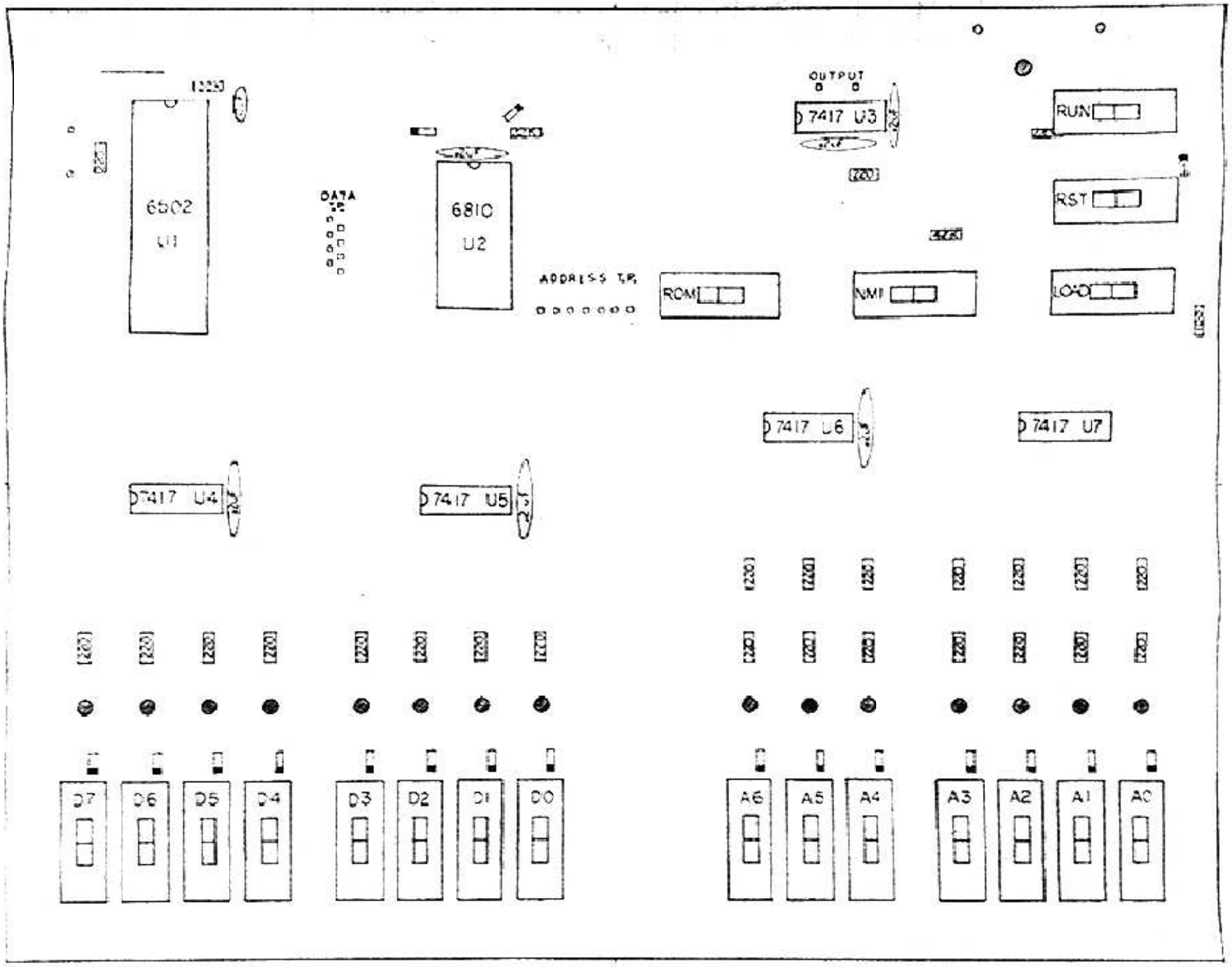
The second byte of the instruction becomes the operand which is an "Offset" added to the contents of the lower eight bits of the program counter when the counter is set at the next instruction. The range of the offset is -128 to +127 bytes from the next instruction.

INDEXED INDIRECT ADDRESSING - In indexed indirect addressing (referred to as (Indirect,X)), the second byte of the instruction is added to the contents of the X index register, discarding the carry. The result of this addition points to a memory location on page zero whose contents is the low order eight bits of the effective address. The next memory location in page zero contains the high order eight bits of the effective address. Both memory locations specifying the high and low order bytes of the effective address must be in page zero.

INDIRECT INDEXED ADDRESSING - In indirect indexed addressing (referred to as (Indirect),Y), the second byte of the instruction points to a memory location in page zero. The contents of this memory location is added to the contents of the Y index register, the result being the low order eight bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high order eight bits of the effective address.

ABSOLUTE INDIRECT - The second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits of that memory location is contained in the third byte of the instruction. The contents of the fully specified memory location is the low order byte of the effective address. The next memory location contains the high order byte of the effective address which is loaded into the sixteen bits of the program counter.


OSI 300
TRAINER COMPUTER
"NMI"
"RST"
"RUN"
"LOAD"
"ROM"
+5V
GND
100KΩ
100 Ω
1.2K
.01 μF NPO
OUTPUT PORT
POWER IN
D7 D6 D5 D4 D3 D2 D1 D0
DATA SWITCHES
A6 A5 A4 A3 A2 A1 A0
ADDRESS SWITCHES


6502
U1
6810
U2
DATA
ADDRESS TP.
OUTPUT
7417 U3
RUN
RST
ROM
NMI
LOAD
7417 U4
7417 U5
7417 U6
7417 U7
D7
D6
D5
D4
D3
D2
D1
D0
A6
A5
A4
A3
A2
A1
A0