Tektronix®
COMMITTED TO EXCELLENCE

Supplement to

8002A

μProcessor Lab Assembler Users Manual
070-3454-00

6500/1 ASSEMBLER SPECIFICS

Option 09

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

Serial Number ____________________

070-2790-00

First Printing APR 1980

WARRANTY

Tektronix warrants that this product, excluding customer-supplied equipment, is free from defects in materials and workmanship. The warranty period is ninety (90) days from the date of shipment. Tektronix will, at its option, repair or replace the product if Tektronix determines it is defective within the warranty period. CRTs are warranted for one (1) year. During the nine (9) months following expiration of the product warranty, Tektronix will replace defective CRTs at no charge for the material.

In the forty-eight (48) contiguous United States, the District of Columbia, and in other areas where Tektronix normally offers on-site service for this product, Tektronix will provide this service at no charge during the product warranty period described above. In areas where Tektronix does not offer on-site service for this product, warranty service will be provided at no charge if the product is returned, freight prepaid, to a service center designated by Tektronix.

Tektronix may use the original vendor's service organization to service any product that is supplied but not manufactured by Tektronix.

Tektronix is not obligated to furnish service under this warranty

a. to repair damage resulting from attempts by personnel other than Tektronix representatives to install, repair, or service the product;

b. to repair damage resulting from improper use or from connecting the product to incompatible equipment;

c. if personnel other than Tektronix representatives modify the hardware or software.

There is no implied warranty of fitness for a particular purpose. Tektronix is not liable for consequential damages.

Section 12A
6500/1 ASSEMBLER SPECIFICS

ILLUSTRATIONS

Section 12A

6500/1 ASSEMBLER SPECIFICS

DEMONSTRATION RUN

Introduction

This Demonstration Run shows you how to enter, modify, assemble, link, and load a simple program and subroutine.

The purpose of this demonstration is to give you the basic information and experience you will need to begin using the assembler, linker, and library generator.

For your convenience, the sample program and subroutine are short. Only a few features of the assembler and linker are demonstrated, and the library generator is not discussed.

This Demonstration Run uses the following conventions:

1. Underlined—Underlined characters in a command line must be entered from your system terminal. Those characters not underlined are system output.
2. <CR>—Each command line ends with a carriage return. When a carriage return is to be entered, the symbol <CR> is used.

Preparation

To do this Demonstration Run you should have a basic understanding of the 8002A μProcessor Lab and the TEKDOS Text Editor. If you need to review how the 8002A and its editor work, refer to the Learning Guide in the 8002A System User manual.

You will need about 60 minutes to complete this Demonstration Run.

Start up your 8002A system. Make sure your system disc has a write-enable tab and is inserted into disc drive 0.

Enter the LDIR command, so that you may examine your disc directory. Make sure you have at least 10 blocks available for the files created during this demonstration:

```
> LDIR <CR>


FILE NAME  BLKS

TEKDOS      16
.          173
COPYSYS      1
NEWDISC      1

TOTAL FILES        4
TOTAL BLOCKS USED 191
BLOCKS AVAILABLE  113  ◄—— Must be at least 10 blocks
TOTAL BAD BLOCKS    0

>
```

If there are not at least 10 blocks available on your system disc, you must make some room by copying some of your non-system files to another disc:

1. Insert another write-enabled disc into disc drive 1.
2. Repeat the following three commands until you have at least 10 blocks available.

```
> COPY filename/0 filename/1 <CR>
> DELETE filename/0 <CR>
> LDIR 0 <CR>
```

(**filename** represents any non-system file.)

Examine the Sample Subroutine and Main Program

Figure 12A-1 lists the subroutine and program you will enter, assemble, link, and load in this Demonstration Run.

The subroutine outputs the ASCII character stored in the accumulator to an I/O address specified by the symbol PORTN. Note: The 6500/1 uses memory locations 80H to 83H for I/O ports.

The main program places a character in the accumulator, calls the subroutine to send the character to the I/O address, and then waits forever.

You can think of the subroutine as a carefully prepared component of a major programming project. The main program can be viewed as a quickly written test for the subroutine.

Subroutine OUTSUB:

```
        TITLE   "SAMPLE SUBROUTINE"
        NAME    SUBSMOD
        GLOBAL  PORTN,OUTSUB
        SECTION SUBS1
; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER.
OUTSUB  STA     PORTN   ; OUTSUB STARTS HERE.
        RTS             ; RETURN TO PROGRAM.
        END
```

Main Program:

```
        GLOBAL  PORTN,OUTSUB
PORTN   EQU     80H     ; I/O ADDRESS = 80 (HEXADECIMAL)
START   LDA     #"?"    ; CHARACTER = "?"
        JSR     OUTSUB  ; SEND "?" TO I/O ADDRESS 80...
        JMP     $       ; ...AND LOOP FOREVER.
        END     START
```

2790-1

Fig. 12A-1. Source code for sample subroutine and main program.

Subroutine OUTSUB outputs a single ASCII character to an I/O address specified by PORTN. The main program specifies a I/O address and a character and calls OUTSUB to send the character to that address. The subroutine and main program are discussed in more detail later in this section.

Assembly Language Statements

An assembler source module is made up of assembly language statements. There are three types of assembly language statements:

- An **assembly language instruction** is translated by the assembler into a 6500/1 machine instruction.
- An **assembler directive** indicates a special action to be taken by the assembler. Assembler directives define data items, constants, and variables; provide information to the linker; control macros and conditional assembly; and specify options for the assembler and linker listings.
- A **macro invocation** is replaced by the statements of the macro it invokes. (Macro invocations are not discussed in this demonstration.)

Each assembly language statement has four fields. Each field may vary in width and certain fields may be blank. However, the fields always occur in the following order:

1. The **label** field. The label field always begins in column 1 of the statement. The label allows the statement to be referenced by other statements. The label usually represents the address of the instruction or data item represented by the statement.
2. The **operation** field. The word in the operation field indicates the type of action to be taken by the assembler. The word may be an assembler directive word, a 6500/1 mnemonic, or the name of a macro. If the word is a 6500/1 mnemonic, the assembler translates the statement into a machine instruction.
3. The **operand** field. The operand field completes the assembly language statement. Most assembler directives and 6500/1 instructions contain one or more operand expressions. The type and number of operands depend on the operation.

4. The **comment** field. Comments are used for program documentation only; they are ignored by the assembler. A semicolon (;) indicates that the remainder of the line is a comment. A comment may follow the operand field, or may begin with a semicolon in column 1 and take up an entire source line.

Explanation of the Subroutine Source Code

The following text explains each statement in the sample subroutine. (shown in Fig. 12A-1). The two statements preceding the END statement are 6500/1 instructions. The remaining statements are assembler directives.

```
        TITLE   "SAMPLE SUBROUTINE"
```

The phrase "SAMPLE SUBROUTINE" will appear in the heading on each page of the assembler source listing.

```
        NAME    SUBSMOD
```

When the subroutine is assembled, the resulting object module will be named "SUBSMOD".

```
        GLOBAL  PORTN,OUTSUB
```

PORTN and OUTSUB are declared as global symbols, since each symbol is given a value in one module and referred to in another module. For example, OUTSUB is defined in the subroutine and referred to in the main program. PORTN is called an **unbound global** because it is not defined anywhere in this module. OUTSUB is a **bound global**.

```
        SECTION SUBS1
```

Each object module is composed of one or more sections. The linker treats each section as a separate unit: sections from the same module may be placed in different ends of memory. The one section in object module SUBSMOD will be called SUBS1. (If you were to add more sections to this source module, they might be called SUBS2, SUBS3, and so on.)

The assembler directives SECTION, COMMON, and RESERVE each declare a different type of section, and may also specify restrictions on the relocatability of the section. When no restriction is specified, the section is **byte-relocatable**; that is, the section may begin at any byte in memory. The Linker section of this manual contains an explanation of the five attributes of a section: name, section type, relocation type, size, and memory location.

```
; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER.
```

This is a comment.

```
 OUTSUB STA     PORTN   ; OUTSUB STARTS HERE.
```

This 6500/1 instruction outputs the contents of the accumulator to an I/O address specified by PORTN. The symbol OUTSUB becomes defined as the address of this instruction, which is the first instruction in the subroutine. A program that contains the instruction JSR OUTSUB can execute this subroutine.

```
        RTS             ; RETURN TO PROGRAM.
```

This 6500/1 instruction returns control to the calling program.

```
        END
```

This assembler directive marks the end of the source module.

Explanation of the Main Program Source Code

The following text explains each statement in the sample main program. The program contains three assembler directives (GLOBAL, EQU, and END) and three 6500/1 instructions (LDA, JSR, and JMP).

```
        GLOBAL  PORTN,OUTSUB
```

As in the subroutine, PORTN and OUTSUB are global symbols. However, in this module, PORTN is a bound (defined) global while OUTSUB is an unbound (undefined) global. The GLOBAL statements allow the two modules to share the I/O address and the address of the subroutine.

```
PORTN   EQU     80H     ; I/O ADDRESS = 80 (HEXADECIMAL)
```

This assembler directive assigns the hexadecimal value 80 to the symbol PORTN. "PORTN" becomes synonymous with the constant "80H".

```
START   LDA     #"?"    ; CHARACTER = "?"
```

This 6500/1 instruction loads the hexadecimal value 3F (the ASCII code for question mark) into the accumulator. This statement is given a label, "START", so that the END statement may refer to it. The "#" specifies immediate addressing: the operand contains the value to be operated on.

```
        JSR     OUTSUB ; SEND "?" TO I/O ADDRESS 80...
```

This 6500/1 instruction transfers control to the instruction labeled OUTSUB in the subroutine module. The subroutine sends the question mark to the I/O address 80H.

```
        JMP     $       ;...AND LOOP FOREVER.
```

Control returns from the subroutine to this 6500/1 instruction. The $ represents the current value of the program counter: this instruction directs the 6500/1 to continue to reexecute this instruction indefinitely.

```
        END     START
```

This assembler directive terminates the source module and indicates that program execution should begin with the instruction labeled "START". START is called the **transfer address**. The transfer address is passed through the assembler and linker to the TEKDOS commands LOAD and GO.

Notice that this program source module does not contain a TITLE, NAME, or SECTION directive. The following default conditions result:

- There will be no special title in the page heading of the source listing.
- The object module will be called *NONAME*.
- The one section in *NONAME* will be given a default name, section type, and relocation type.

Naming Files

This Demonstration Run produces several files. To give each file a name that reflects its contents and importance, we will use the file naming standards described in the File Management section of the 8002A System Users Manual:

- The last character of the file name is a letter that represents the type of file.
- The next-to-last character is either a percent sign (%) or a semicolon (;). The percent sign signifies a file that cannot be readily replaced. The semicolon signifies a temporary or readily replaced file.
- The first part of the file name is a descriptive name.

The following files will be produced:

File Name	Explanation	How Created
DEMSUB%S	DEMonstration SUBroutine Source file	by you
DEMSUB;O	DEMonstration SUBroutine Object file	by assembler
DEMSUB;A	DEMonstration SUBroutine Assembler listing	by assembler
DEMPRO%S	DEMonstration PROgram Source file	by you
DEMPRO;O	DEMonstration PROgram Object file	by assembler
DEMPRO;A	DEMonstration PROgram Assembler listing	by assembler
DEMPRO;L	DEMonstration PROgram Load file	by linker
DEMPRO;K	DEMonstration PROgram linKer listing	by linker

Create the Subroutine Source File

The TEKDOS Editor helps create and modify source files. The Editor Dictionary section of the 8002A System Users Manual contains a complete explanation of the editor.

How to Correct Typing Mistakes in the Editor

To delete the last character typed, use the BACKSPACE, RUBOUT, or DELETE key:

- The BACKSPACE key on a CRT terminal cancels the character and backs the cursor over it.
- The RUBOUT key on a CRT terminal cancels the character and echoes the cancelled character on the terminal.
- The DELETE key on a hard-copy terminal cancels the character and echoes the cancelled character on the terminal.

To delete the entire line being typed:

- Press the ESC (escape) key once. You may then re-enter the line.

Start Editing

The EDIT command invokes the TEKDOS Editor. The file name in the EDIT command indicates the file to be edited. Enter the following line to begin the editing session that creates DEMSUB%S, the subroutine source file:

```
> EDIT DEMSUB%S <CR>

** EDIT VER x.x **
**NEW FILE**
*
```

The asterisk (*) is the editor prompt character. When you see the asterisk, you may enter the next editor command.

An assembly language program is easier to read if the statement fields are aligned as in Fig. 12A-1. The editor has tab stops at columns 8, 16, 24, 32, 40, 48, 56, and 64, which are convenient for aligning assembly language text. For example, in Fig. 12A-1, the operation field begins in column 8, the operand field begins in column 16, and the comment field begins in column 24.

Enter the following command to declare the backslash (\) as the editor tab character:

```
*TAB \ <CR>
```

The editor will interpret every backslash you enter as a skip to the next tab stop.

Enter input mode and type in the subroutine. Be sure to misspell "GLOBAL" in the third line of text. This deliberate typing error will be used to illustrate features of the assembler and editor.

```
*INPUT <CR>
INPUT:
\TITLE\"SAMPLE SUBROUTINE" <CR>
\NAME\SUBSMOD <CR>
\GLOABL\PORTN,OUTSUB <CR>
\SECTION\SUBS1 <CR>
; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER.  <CR>
OUTSUB\STA\PORTN\; OUTSUB STARTS HERE.  <CR>
\RTS\\; RETURN TO PROGRAM.  <CR>
\END <CR>
<CR>
*
```

When you enter the carriage return on the empty line, input mode is terminated and the editor prompt (*) appears.

The text you entered is stored in the editor workspace. Each backslash you entered has been replaced by spaces up to the next tab stop. To display the workspace contents from beginning to end, enter the following command:

```
*TYPE B-E <CR>
        TITLE   "SAMPLE SUBROUTINE"
        NAME    SUBSMOD
        GLOABL  PORTN,OUTSUB
        SECTION SUBS1
; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER.
OUTSUB  STA     PORTN   ; OUTSUB STARTS HERE.
        RTS             ; RETURN TO PROGRAM.
        END
*
```

Now enter the FILE command to copy the text in the workspace out to the new source file and end the editing session:

```
*FILE <CR>
*DOS* EOJ

>
```

The TEKDOS prompt (>) indicates that you are out of the editor and may enter another TEKDOS command.

Assemble the Subroutine and Examine Any Errors

The TEKDOS command ASM invokes the assembler and specifies the source file(s) to be assembled and the object and listing files to be produced. The ASM command has the following format:

```
ASM,objfile,lisfile,soufile[,soufile]...
```

objfile—name of object file to be produced
lisfile—name of listing file to be produced
soufile—name(s) of source file(s) to be assembled

To scan source file DEMSUB%S for errors, enter the following command:

```
> ASM,,,DEMSUB%S <CR>
```

Omitting the names of the object and listing files has two advantages:

1. The assembler runs faster because it produces no object code or listing.
2. The ASM command line is shorter.

You may want to omit the object and listing files from your ASM command line whenever you suspect that your source code contains errors.

The assembler responds as follows on your system terminal:

```
Tektronix  6500/1 ASM Vx.x
**** Pass 2
00003 0000 000000            GLOABL  PORTN,OUTSUB
***** ERROR 039: Invalid operation code
00006 0000 8D0000     OUTSUB STA     PORTN    ; OUTSUB STARTS HERE.
***** ERROR 074: Undefined symbol
    8 Source Lines       8 Assembled Lines   47038 Bytes available
    2 ERRORS             2 UNDEFINED SYMBOLS
*ASM* EOJ

>
```

The assembler's response can be interpreted as follows:

```
Tektronix  6500/1 ASM Vx.x
```

The assembler announces itself as it begins executing. The assembler reads through your source file twice. The first time through (Pass 1), the assembler makes a list of symbols that appear in the source code and tries to assign an address or value to each symbol.

```
**** Pass 2
```

The assembler begins its second pass through your source file. During Pass 2 the assembler produces the object and listing files and displays error messages and statistics.

```
00003 0000 000000            GLOABL  PORTN,OUTSUB
***** ERROR 039: Invalid operation code
```

The assembler cannot translate the above statement because "GLOABL" is not a 6500/1 mnemonic, an assembler directive word, or the name of a macro. The erroneous source line and the error message would appear in the listing (if any) just as they appear on the system terminal. The three numbers to the left of the statement will be explained when you examine an assembler listing later in this Demonstration Run.

```
00006 0000 8D0000     OUTSUB STA     PORTN    ; OUTSUB STARTS HERE.
***** ERROR 074: Undefined symbol
```

Because the assembler did not understand the GLOBAL statement, it does not know that PORTN is a global symbol. The assembler expects PORTN to be defined in this module.

```
    8 Source Lines       8 Assembled Lines   47038 Bytes available
    2 ERRORS             2 UNDEFINED SYMBOLS
```

These lines summarize the assembler's activities. There are eight lines of code in your source file. The number of assembled lines differs from the number of source lines only in programs that contain macros or conditional assembly.

The "Bytes available" message indicates the amount of Program Memory not used by the assembler. If the "Bytes available" figure is ever less than 1000 or so, you may need to divide your source module into smaller modules before you add any more statements.

The two errors, already discussed, produced the two undefined symbols GLOABL and PORTN.

Correct the Error in the Subroutine Source Code

Both errors detected by the assembler arose from the misspelling of "GLOBAL" in line 3 of the source file, DEMSUB%S. Invoke the editor so that you may correct the misspelling:

```
> EDIT DEMSUB%S <CR>

** EDIT VER x.x **
*
```

The editor command GET brings text into the workspace from the file being edited. Specify a large number of lines (99) to assure that the entire file is brought into the workspace:

```
*GET 99 <CR>
**EOF**
```

The message **EOF** indicates that the end of the input file has been reached.

Enter the following command line. The BEGIN command moves the workspace pointer to line 1. Starting at that line, the FIND command searches for the character string "GLO" and moves the pointer to the first line that contains the string.

```
*BEGIN.:FIND /GLO/ <CR>
        GLOABL  PORTN,OUTSUB
```

Now the workspace pointer is at the line you want to modify. Use the SUBSTITUTE command to reverse the letters "A" and "B" in "GLOABL":

```
*SUB /AB/BA/ <CR>
        GLOBAL  PORTN,OUTSUB
```

The modified line is displayed.

As before, the FILE command copies the edited source code to the source file and closes the editing session:

```
*FILE <CR>
**EOF**
*DOS* EOJ

>
```

Re-Assemble the Subroutine

Enter the following command to create an object file (DEMSUB;O) and an assembler listing file (DEMSUB;A) from the subroutine source file:

```
> ASM DEMSUB;O DEMSUB;A DEMSUB%S <CR>

Tektronix  6500/1 ASM Vx.x
**** Pass 2
    8 Source Lines       8 Assembled Lines   47038 Bytes available
             >>> No assembly errors detected <<<
*ASM* EOJ

>
```

This time the assembler finds no errors.

Examine the Subroutine Listing

In order to examine the assembler listing stored on file DEMSUB;A, copy the file to your line printer:

```
> COPY DEMSUB;A LPT1 <CR>
```

If you have no line printer, enter the following command to list the file on your system terminal. (Remember that you may use the space bar to suspend or resume display on a CRT terminal.)

```
> COPY DEMSUB;A <CR>
```

Figure 12A-2 shows the listing of the sample subroutine.

```
Tektronix       6500 ASM Vx.x  SAMPLE SUBROUTINE                Page     1


00002                           NAME    SUBSMOD
00003                           GLOBAL  PORTN,OUTSUB
00004                           SECTION SUBS1
00005                    ; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER.
00006 0000 8D0000  >   OUTSUB STA     PORTN    ; OUTSUB STARTS HERE.
00007 0003 60                  RTS              ; RETURN TO PROGRAM.
00008                           END




Tektronix       6500 ASM Vx.x  Symbol Table                     Page     2


SUBS1 Section (0004)

   OUTSUB - 0000 G

PORTN Unbound Global


    8 Source Lines        8 Assembled Lines    47038 Bytes available

          >>> No assembly errors detected <<<                       2790-2
```

Fig. 12A-2. Assembler listing for the sample subroutine.

The command ASM DEMSUB;O DEMSUB;A DEMSUB%S produces this listing file from the subroutine source file. The command COPY DEMSUB;A LPT1 copies the listing file to the line printer.

Every assembler listing has two parts: the source listing and the symbol table. Each page of the listing begins with a standard page heading.

The Source Listing

Page 1 of your listing contains the source listing. The heading includes the words "SAMPLE SUBROUTINE", which you supplied with the TITLE directive.

Each line of the source listing contains the following information:

1. the line number (decimal);
2. the memory location (hexadecimal) of the object code generated (if any);
3. the assembled object code (hexadecimal);
4. a relocation indicator (>) if the object code may be adjusted by the linker;
5. a text substitution indicator (+) if the assembler has modified the source statement (this demonstration gives no examples of text substitution);
6. the source statement.

If any statement contains an error, the appropriate error message appears directly after the statement.

Examine each line of your source listing:

- Line 1 (the TITLE directive) is not printed because it is a listing control directive.
- Lines 2, 3, 4, and 8 are assembler directives that produce no object code. The information they provide is stored in special areas of the object module.
- Line 5 is a comment.
- Lines 6 and 7 are 6500/1 assembly language instructions:

  The 6500/1 instruction STA PORTN produces the three-byte machine instruction 8D0000. 8D is the hexadecimal operation code for the STA instruction. The dummy value 0000 will be used for the I/O address until the linker supplies a value for PORTN.

  The machine instruction 8D0000 is stored in bytes 0000 through 0002 of section SUBS1.

  The 6500/1 instruction RTS produces the one-byte machine instruction 60, which is stored in byte 0003 of section SUBS1.

The Symbol Table

Page 2 of your listing contains the symbol table, which indicates the value and type of each symbol in your source code.

The assembler symbol table is divided into the following categories:

1. Strings and macros
2. Scalars (numeric values other than addresses; includes undefined symbols)
3. Sections (and addresses within each section)
4. Unbound globals

Examine the symbol table in your listing:

1. The strings and macros table is omitted, since the sample subroutine uses neither string variables nor macros.
2. The scalars table is omitted since the sample subroutine defines no scalars.
3. SUBS1 is the only section in the sample subroutine. The line

   ```
   SUBS1 Section (0004)
   ```

   tells you the following attributes of section SUBS1:

   - its name: SBUS1;
   - its section type: SECTION (as opposed to COMMON or RESERVE);
   - its relocation type: byte-relocatable (the default relocation type is implied when no other relocation type is specified);
   - its length: 4 bytes.

   OUTSUB has the value 0000 because OUTSUB is the address of the first byte in section SUBS1. The letter "G" indicates that OUTSUB is a global symbol.
4. PORTN is the only unbound (undefined) global symbol in the subroutine.

The statistics at the bottom of the symbol table are the same statistics that appeared on the system terminal when the assembler finished execution.

When there are errors in your source code, the two most useful parts of your listing are the source listing and the scalars table. The source listing contains the error messages and shows the erroneous lines in context with the rest of the program. The scalars table flags undefined symbols with the value "****".

Create the Main Program Source File

Now that you have created, corrected, and assembled the sample subroutine, it is time to create the main program that uses the subroutine. Enter the following command to begin the editing session that creates the main program source file, DEMPRO%S:

```
> EDIT DEMPRO%S <CR>

** EDIT VER x.x **
**NEW FILE**
*
```

Declare the editor tab character and type in the source code, just as you did for the subroutine: (This time, however, don't include any typing errors.)

```
*TAB \ <CR>
*INPUT <CR>
INPUT:
\GLOBAL\PORTN,OUTSUB <CR>
PORTN\EQU\80H\; I/O ADDRESS = 80 (HEXADECIMAL) <CR>
START\LDA\#"?"\; CHARACTER = "?" <CR>
\JSR\OUTSUB\; SEND "?" TO I/O ADDRESS 80... <CR>
\JMP\$\; ... AND LOOP FOREVER. <CR>
\END\START <CR>
<CR>
*
```

Inspect the text you have entered to be sure there are no errors:

```
*TYPE B-E <CR>
        GLOBAL  PORTN,OUTSUB
PORTN   EQU     80H      ; I/O ADDRESS = 80 (HEXADECIMAL)
START   LDA     #"?"     ; CHARACTER = "?"
        JSR     OUTSUB   ; SEND "?"  TO I/O ADDRESS 80 ...
        JMP     $        ; ... AND LOOP FOREVER.
        END     START
*
```

Enter the FILE command to save the text onto the source file and return to TEKDOS:

```
*FILE <CR>
*DOS* EOJ

>
```

Assemble the Main Program

Enter the following command line to create an object file (DEMPRO;O) and a listing file (DEMPRO;A) from the main program source file:

```
> ASM DEMPRO;O DEMPRO;A DEMPRO%S <CR>

Tektronix  6500/1 ASM Vx.x
**** Pass 2
    6 Source Lines       6 Assembled Lines   47061 Bytes available
              >>> No assembly errors detected <<<
*ASM* EOJ

>
```

The main program contains no errors.

Examine the Main Program Listing

Copy the assembler listing to the line printer or to the system terminal:

```
  > COPY DEMPRO;A LPT1 <CR>
```

or

```
  > COPY DEMPRO;A <CR>
```

```
Tektronix       6500 ASM Vx.x                               Page      1


00001                        GLOBAL  PORTN,OUTSUB
00002       0080       PORTN  EQU     80H       ; I/O ADDRESS = 80 (HEXADECIMAL)
00003 0000  A93F       START  LDA     #"?"      ; CHARACTER = "?"
00004 0002  200000  >         JSR     OUTSUB    ; SEND "?" TO I/O ADDRESS 80...
00005 0005  4C0500  >         JMP     $         ; ...AND LOOP FOREVER.
00006       0000    >         END     START

Tektronix       6500 ASM Vx.x  Symbol Table                 Page      2

Scalars

   PORTN -- 0080 G

%DEMPRO (default) Section (0008)

   START -- 0000

OUTSUB Unbound Global

    6 Source Lines      6 Assembled Lines    47061 Bytes available

            >>> No assembly errors detected <<<                       2790-3
```

Fig. 12A-3. Assembler listing for the sample main program.

The command ASM DEMPRO;O DEMPRO;A DEMPRO%S produces this listing file from the main program source file. The command COPY DEMPRO;A LPT1 copies the listing file to the line printer.

Compare the listing of the sample main program (Fig. 12A-3) with the listing of the sample subroutine (Fig. 12A-2).

The Source Listing

Page 1 of your assembler listing contains the source listing. Notice that there is no user-defined title for the program listing: the source code contains no TITLE directive.

Examine each line of the program source listing:

1. As in the subroutine, the GLOBAL statement produces no object code.
2. The EQU statement assigns the value 80 (hexadecimal) to the symbol PORTN. The symbol PORTN and its value are stored in the global symbol block of the program object module. At link time the value of PORTN will be substituted into the STA instruction in the subroutine.

3. The 6500/1 assembly language instruction LDA #"?" generates the machine instruction A93F. A9 is the operation code for "LDA immediate" and 3F is the ASCII code for the question mark. The machine instruction A93F is stored in bytes 0000 and 0001 of the main program.
4. The 6500/1 assembly language instruction JSR OUTSUB generates the machine instruction 200000 in bytes 0002 through 0004. 20 is the operation code for the JSR instruction. 0000 is a dummy value: the address of OUTSUB will be provided at link time.
5. The 6500/1 assembly language instruction JMP $ produces the three-byte machine instruction 4C0500 in bytes 0005 through 0007 of the main program. 4C is the operation code for the JMP instruction. 0500 is the address of the JMP instruction (low-byte first).
6. The END statement specifies that the transfer address is 0000, the address of the LDA instruction. The transfer address will be adjusted if this section of object code is not loaded at the beginning of memory.

The Symbol Table

1. The strings and macros table is again omitted because it is empty.
2. The scalars table lists the usual pre-defined scalars, plus the symbol PORTN. The value of PORTN is 0080 hexadecimal. The "G" indicates that PORTN is a global symbol.
3. Because the main program source code contains no SECTION directive, the section produced by this assembler run is given the following default attributes:
   - name: %DEMPRO (derived from the name of the object file);
   - section type: SECTION;
   - relocation type: byte-relocatable.

   Section %DEMPRO contains eight bytes of code. START is the address of the first byte of the section.
4. OUTSUB is the only unbound (undefined) global symbol in the main program.

The statistics at the bottom of the symbol table are the same statistics that appeared on the system terminal when the assembler finished execution.

Link the Object Modules

Now both the subroutine and the main program have been translated into machine language. In order for the subroutine and main program object modules to communicate with each other, they must be linked. The linker performs the following tasks in creating a load file of executable object code:

- It finds a block of memory for each section in the specified object files.
- It adjusts addresses to reflect relocation of sections.
- It provides values for unbound globals.

Enter the following command to create a load file (DEMPRO;L) and a linker listing file (DEMPRO;K) from your two object files:

```
> LINK DEMPRO;L DEMPRO;K DEMPRO;O DEMSUB;O <CR>
```

The linker responds as follows:

```
      NO ERRORS     NO UNDEFINED SYMBOLS
      2  MODULES    2 SECTIONS
      TRANSFER ADDRESS IS 0000
  *LINK* EOJ

  >
```

Examine the Linker Listing

Copy the linker listing file to the line printer or system terminal:

```
> COPY DEMPRO;K LPT1 <CR>
```

or

```
> COPY DEMPRO;K <CR>
```

Figure 12A-4 shows the linker listing.

```
Tektronix       6500 LINKER Vx.x        GLOBAL SYMBOL LIST      Page     1

  %DEMPRO    0000   OUTSUB    0008   PORTN      0080   SUBS1     0008


Tektronix       6500 LINKER Vx.x        MODULE MAP              Page     2


FILE:  DEMPRO;0

MODULE:  *NONAME*
   %DEMPRO    SECTION BYTE 0000-0007


FILE:  DEMSUB;0

MODULE:  SUBSMOD
   SUBS1      SECTION BYTE 0008-000B
   OUTSUB     0008

Tektronix       6500 LINKER Vx.x        MEMORY MAP              Page     3

   0000-0007  %DEMPRO   SECTION BYTE
   0008-000B  SUBS1     SECTION BYTE

    NO ERRORS     NO UNDEFINED SYMBOLS
    2  MODULES    2  SECTIONS
    TRANSFER ADDRESS IS 0000                                          2790-4
```

Fig. 12A-4. Linker listing.

The command LINK DEMPRO;L DEMPRO;K DEMPRO;O DEMSUB;O produces this linker listing file. The command COPY DEMPRO;K LPT1 copies the listing file to the line printer.

The standard linker listing contains three parts:

1. The **global symbol list** (page 1 of your listing) lists the value assigned to each global symbol. The name and starting address of each section are included. Undefined globals are flagged with the value "****".
2. The **module map** (page 2) provides the following information for each object module being linked:
   - the name of the object file or library file supplying the object module;
   - the name and attributes of each section in the module. Any entry points (addresses declared as global symbols) for each section are also listed.

   The module map allows you to verify that each section of your program has been assigned a place in memory.
3. The **memory map** (page 3) lists the sections in the order they occur in memory. Conflicting (overlapping) memory allocations are indicated with an asterisk (*).

   Linker statistics appear at the bottom of the memory map.

An optional feature of the linker listing, the internal symbol list, is useful for program debugging. The internal symbol list is not demonstrated here but is discussed in the Linker section.

The Memory Map

The memory map (page 3 of your listing) provides the most concise summary of the load file produced by the linker.

The memory map shows that bytes 0000 through 0007 of memory will contain section %DEMPRO (the main program) and that bytes 0008 through 000B will contain section SUBS1 (the subroutine).

The memory map also gives the section type (SECTION) and relocation type (byte-relocatable) for each section.

Notice that the transfer address remains unchanged because the section containing the transfer address is located at the beginning of memory.

The Module Map

The module map (page 2) shows much the same information as the memory map. The module map, however, reports the sections by module rather than by memory location.

The first object file, DEMPRO;O, contains the object module called *NONAME*. (Recall that the main program source code contains no NAME directive.) The main program consists of the single section %DEMPRO, whose attributes you already know from the memory map.

The second object file, DEMSUB;O, contains the subroutine object module, SUBSMOD. SUBSMOD consists of the single section SUBS1. The single entry point to SUBS1 is OUTSUB, whose adjusted address (after relocation) is 0008.

The Global Symbols List

The global symbols list (page 1) shows the two symbols declared in GLOBAL statements (OUTSUB and PORTN) and the two section names (%DEMPRO and SUBS1).

Load the Executable Object Code into Memory

Use the FILL command to fill program memory with zeros before you load your object program. Later, when you examine memory, the zeros make it easy to identify the beginning and end of your code. Fill memory from addresses 0000 through 000F with zeros using the following command line:

```
> FILL 0 F 00 <CR>
```

Enter the following command to copy the executable object code from the load file into program memory:

```
> LOAD DEMPRO;L <CR>
TRANSFER ADDRESS: 0000
*LOAD* EOJ

>
```

Bytes 0000 through 000B of program memory now contain the 12 bytes of machine language that form the executable program.

The TEKDOS command DUMP displays the contents of a specified section of memory. Each byte is displayed as a two-digit hexadecimal number and as the ASCII character it represents (if any). Enter the following command to display the contents of memory locations 0000 through 000F: 9

```
> DUMP 0 F <CR>
```

$$\underbrace{\texttt{0000}}_{\text{address of first byte displayed}}\texttt{=}\underbrace{\texttt{A9 3F 20 08 00 4C 05 00}}_{\text{main program}}\ \underbrace{\texttt{8D 80 00 60}}_{\text{subroutine}}\ \underbrace{\texttt{00 00 00 00}}_{\text{memory not affected by LOAD command}}\quad\underbrace{\texttt{.? ..L.....`....}}_{\text{corresponding ASCII characters}}$$

Compare the relocatable object code produced by the assembler with the executable object code produced by the linker. (The addresses and object bytes adjusted by the linker are underlined.)

Relocatable Object Code (from assembler listings)			**Executable Object Code (from DUMP output)**		
address	**object code**	**source code**	**address**	**object code**	**source code**
0000	A93F	LDA #"?"	0000	A93F	LDA #"?"
0002	200000	JSR OUTSUB	0002	20$\underline{0800}$	JSR OUTSUB
0005	4C0500	JMP $	0005	4C0500	JMP $
0000	8D0000	STA PORTN	$\underline{0008}$	8D$\underline{8000}$	STA PORTN
0003	60	RTS	$\underline{000B}$	60	RTS

Note the adjustments made by the linker:

- The subroutine is relocated from byte 0000 to byte 0008.
- The address of the subroutine is substituted into the JSR instruction.
- The I/O address is substituted into the STA instruction.

Summary of Demonstration Run

Enter the following command to list the files on your system disc:

```
> LDIR <CR>

FILE NAME  BLKS

TEKDOS       16
.           173
COPYSYS       1
NEWDISC       1
DEMSUB;O      1
DEMSUB;A      1
DEMPRO%S      1
DEMPRO;O      1
DEMPRO;A      1
DEMPRO;L      1
DEMPRO;K      1
DEMSUB%S      1

TOTAL FILES         12
TOTAL BLOCKS USED  199
BLOCKS AVAILABLE   105
TOTAL BAD BLOCKS     0

>
```

Recall the eight files you have created in this Demonstration Run:

- the two source files (DEMSUB%S and DEMPRO%S) you created using the editor;
- the two object files (DEMSUB;O and DEMPRO;O) and the two listing files (DEMSUB;A and DEMPRO;A) generated by the assembler;
- the load file (DEMPRO;L) and the listing file (DEMPRO;K) generated by the linker.

You have now finished the Demonstration Run. It emphasized how to:

- create a source file using the editor;
- create an object file from a source file using the assembler;
- create a load file from object files using the linker;
- copy the load file into memory using the LOAD command;
- interpret listings generated by the assembler and linker.

Because the 6500/1 does not execute instructions from the first 800H bytes of its addressing space, the program you have loaded below 800H will not run on the 6500/1 Emulator Processor. However, the linker allows you to relocate the object code to a more suitable part of memory (for example, bytes 0800-080B) without changing your source code. This feature of the linker is beyond the scope of this Demonstration Run; see the linker command LOCATE in the Linker section of this manual.

The 6500/1 Demonstration Run in the Emulator Specifics section of the 8002A System Users Manual shows you how to execute and monitor a program you have loaded into memory.

6500/1 INSTRUCTION SET

Introduction

All 6500/1 instructions are summarized in this subsection. The following topics are covered in this subsection:

- Notational Conventions—Describes each symbol used in this subsection.
- Addressing Modes—The methods by which the 6500/1 microprocessor accesses and manipulates data.
- Summary of 6500/1 Instruction Set—Explains the syntax and effects of each instruction, including the operation code, addressing mode, status bits affected, and number of cycles.

Notational Conventions

The following notational conventions are used in the description of the Addressing Modes and 6500/1 Instruction Set.

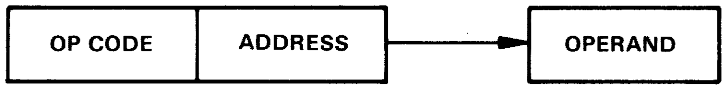
Symbol	Description
+	addition operator
-	subtraction operator
&	logical AND operator
!	logical inclusive OR operator
!!	logical exclusive OR operator
( )	contents of an address, register, or status bit
(( ))	contents of a location whose address is contained in the specified register (indirect addressing)
←	indicates "is transferred to"
#	specifies immediate addressing mode
@	specifies zero page addressing mode
0	reset to 0
1	set to 1
6	status bit is set to reflect bit 6 of memory location
7	status bit is set to reflect bit 7 of memory location
A	accumulator
addr	a 16-bit immediate address value (0 to 65535)
B	break status
C	carry bit
D	decimal mode bit
data8	one-byte data
disp	an 8-bit signed address displacement (-128 to +127)
FFFE	address FFFE (hexadecimal) in memory
FFFF	address FFFF (hexadecimal) in memory
I	interrupt disable bit

Symbol	Description
(IND,X)	indexed indirect addressing mode
(IND),Y	indirect indexed addressing mode
N	negative
PC	program counter
PCH	the high byte of the program counter
PCL	the low byte of the program counter
SP	stack pointer
SR	status register (includes the status bits N, Z, C, I, D, V)
u	unaffected by operation
V	overflow bit
X	index register X
x	set or reset, depending on operation result
Y	index register Y
Z	zero result bit
zasm	a zero page address at assembly time
zlink	a zero page address at link time
Zpage	zero page addressing mode

Addressing Modes

The following addressing modes are used by the 6500/1 Instructions.

Absolute Addressing. Instructions using absolute addressing have three bytes. The first byte contains the op code. The second and third bytes are the low-order and high-order bytes of the operand address.



Example:

```
          LDA   ADDR
           .
           .
           .
ADDR      BYTE  DATA
           .
```

Loads DATA from memory location ADDR into accumulator.

Accumulator Addressing. Instructions using accumulator addressing have one byte; that byte contains the op code. The operand of the instruction (the accumulator) is implicit in the op code. The following shift and rotate instructions use the accumulator addressing mode: ASL, LSR, and ROR.

OP CODE

Example:

```
            ROR    A
```

To specify the accumulator as opposed to a byte of memory enter "A" in the operand field instead of the address of the operand.

Implied Addressing. Instructions using implied addressing have one byte. The op code completely defines the operation to be done. Instructions utilizing this type of addressing include operations that clear or set bits in the processor status register, increment or decrement registers, or transfer contents of one register to another.

OP CODE

Example:

```
            TAX
```

The operands of TAX (index register A and X) are implied in the mnemonic.

Immediate Addressing. Instructions using immediate addressing have two bytes. The first byte contains the op code. The second byte contains the operand.

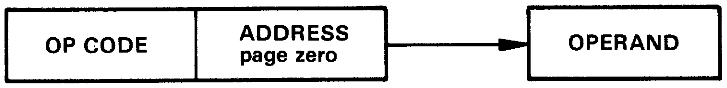
OP CODE	OPERAND

Example:

```
            LDA    #5
```

Loads the value 5 into the accumulator. The '#' specifies immediate addressing.

Zero Page Addressing. Instructions using zero page addressing have two bytes. The first byte contains the op code, while the second byte contains the address of the operand. The address must be within the range 0 to 255 (page zero).

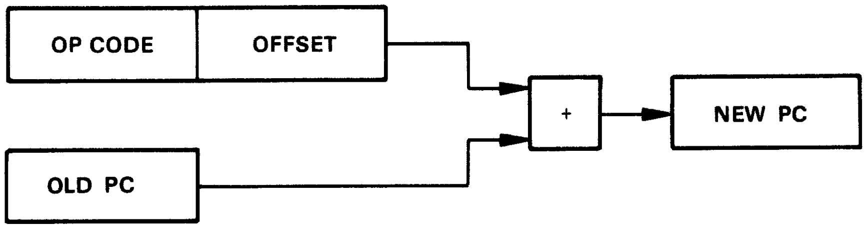


Example:

```
        LDA    @TEMP
```

TEMP is the address of the byte to be loaded into the accumulator. The @ sign forces zero page addressing. If TEMP is already defined in an absolute section, in range 0-255, the assembler will use zero page addressing even if the @ sign is omitted.

Relative Addressing. Instructions using relative address have two bytes. The first byte contains the op code, while the second byte contains the offset to be added to the program counter. All conditional branch instructions use relative addressing. The offset must be within the range -128 to +127.



If the operand is a scalar value, the assembler uses this value as the actual offset.

Example:

```
        BMI    100
```

Branches forward 100 bytes if negative-result bit is set.

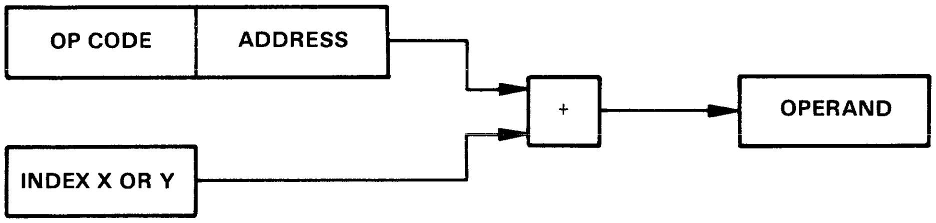
If the operand is an address, the assembler computes the offset necessary to branch to this address.

Example:

```
        BMI    ADDR
```

Branches (forward or backward, as necessary) to ADDR if negative-result bit is set. (ADDR must be within 128 bytes of current location.)

Absolute Indexed Addressing. Instructions using absolute indexed addressing have three bytes. The first byte contains the op code. The second and third bytes contain an address that is added to the index register X or Y to produce the address of the operand.

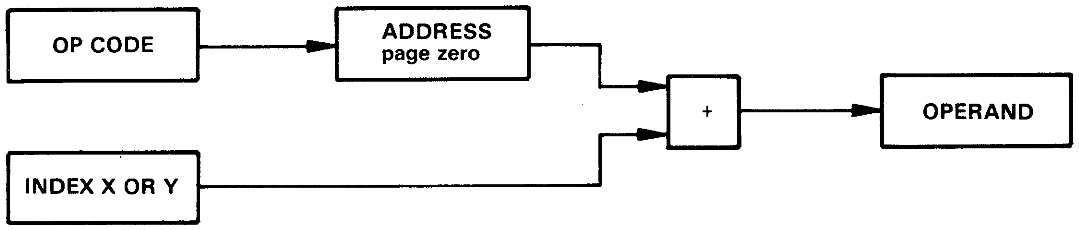


Example:

```
                LDA    TEMP,X
                 .
                 .
 TEMP + (X)     ADDR    OPERAND
```

If TEMP = 100 and index register X = 5, then the byte in memory location 105 is loaded into the accumulator.

Zero Page Indexed Addressing. Instructions using zero page indexed addressing have two bytes. The first byte contains the op code. The contents of index register X or Y is added to the second byte of the instruction to produce the address of the operand (a zero page address). Note that index register Y is allowed only in the instructions LDX and STX.



Example:

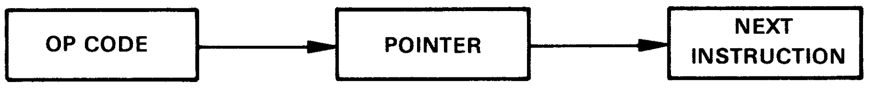
```
                LDA    TEMP,X
                 .
                 .
 TEMP + (X)     BYTE OPERAND
```

If index register X = 100 and TEMP = 50, then the byte in memory location 150 is loaded into the accumulator.

```
                LDX    @TEMP,X
```

The @ forces zero page addressing. If TEMP is already defined in an absolute section and in range 0–255, the assembler will use zero page addressing even if the @ sign is omitted.

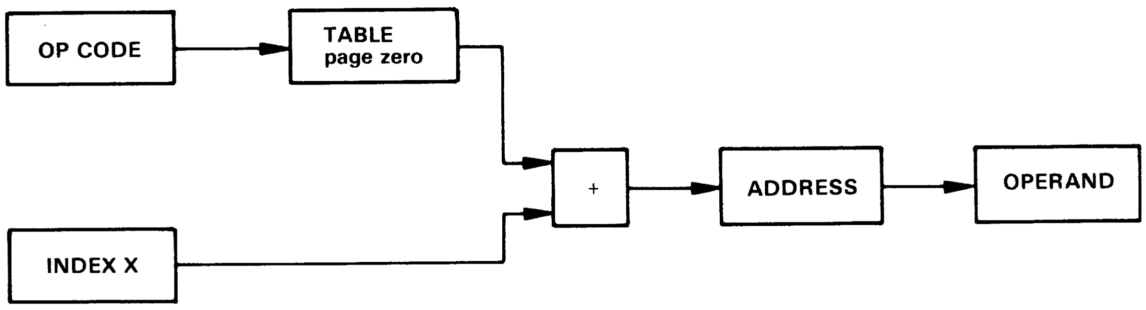
Indirect Addressing. Only the three-byte JMP instruction uses indirect addressing. The first byte contains the op code. The second and third bytes contain a pointer to the next instruction to be executed.



Example:

```
                JMP    (POINTER)
                 .
                 .
                 .
POINTER         WORD    ADDR
                 .
                 .
ADDR            [NEXT INSTRUCTION]
```

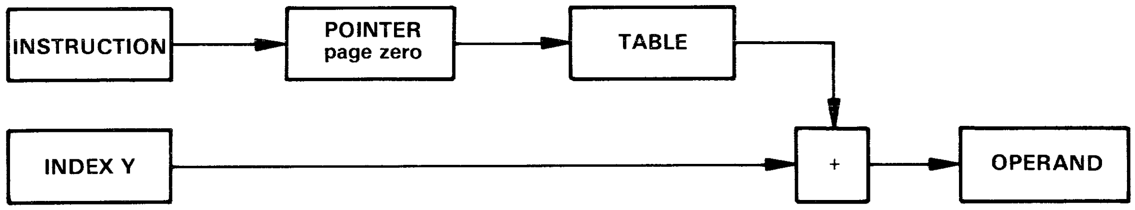
Indexed Indirect Addressing. Instructions using indexed indirect addressing have two bytes. The first byte contains the op code. The second byte contains the address of a table of pointers on page zero. Index register X selects a 16-bit address (pointer) from this table and that address points to the operand.



Example:

```
                    LDA    (TABLE,X)
                     .
                     .
                     .
TABLE + (X)         WORD    ADDR
                     .
                     .
ADDR                OPERAND
```

Indirect Indexed Addressing. Instructions using indirect indexed addressing have two bytes. The first byte contains the op code, while the second byte contains the address of a pointer on page zero. The pointer is the address of a table. Index register Y selects which byte in this table is the operand.



Example:

```
                    LDA   (POINTER),Y
                     .
                     .
                     .
POINTER             WORD   TABLE
                     .
                     .
                     .
TABLE + (Y)         OPERAND
```

SUMMARY OF 6500/1 INSTRUCTIONS

Mnemonic	Operand	Addr Mode	Operation	N	Z	C	I	D	V	OP Code	No. Bytes	No. Cycles
Arithmetic Instructions												
ADC	#data8	Immediate	(A)←(A)+data8+(C)	x	x	x	u	u	x	69	2	2
ADC	zasm	Z page	(A)←(A)+(zasm)+(C)	x	x	x	u	u	x	65	2	3
ADC	@zlink	Z page	(A)←(A)+(zlink)+(C)	x	x	x	u	u	x	65	2	3
ADC	zasm,X	Z page,X	(A)←(A)+(zasm+(X))+(C)	x	x	x	u	u	x	75	2	4
ADC	@zlink,X	Z page,X	(A)←(A)+(zlink+(X))+(C)	x	x	x	u	u	x	75	2	4
ADC	addr	Absolute	(A)←(A)+(addr)+(C)	x	x	x	u	u	x	6D	3	4
ADC	addr,X	Absolute,X	(A)←(A)+(addr+(X))+(C)	x	x	x	u	u	x	7D	3	4[a]
ADC	addr,Y	Absolute,Y	(A)←(A)+(addr+(Y))+(C)	x	x	x	u	u	x	79	3	4[a]
ADC	(zlink,X)	IND,X)	(A)←(A)+((zlink+(X)))+(C)	x	x	x	u	u	x	61	2	6
ADC	(zlink),Y	(IND),Y	(A)←(A)+((zlink)+(Y))+(C)	x	x	x	u	u	x	71	2	5[a]
Add memory to accumulator with carry. Zero bit is not valid in decimal mode.												
CLC		Implied	(C)←0	u	u	0	u	u	u	18	1	2
Clear carry bit.												
CLD		Implied	(D)←0	u	u	u	u	0	u	D8	1	2
Clear decimal mode.												
CLV		Implied	(V)←0	u	u	u	u	u	0	B8	1	2
Clear overflow bit.												
SBC	#data8	Immediate	(A)←(A)-data8-C	x	x	x	u	u	x	E9	2	2
SBC	zasm	Z page	(A)←(A)-(zasm)-C	x	x	x	u	u	x	E5	2	3
SBC	@zlink	Z page	(A)←(A)-(zlink)-C	x	x	x	u	u	x	E5	2	3
SBC	zasm,X	Z page,X	(A)←(A)-(zasm+(X))-C	x	x	x	u	u	x	F5	2	4
SBC	@zlink,X	Z page,X	(A)←(A)-(zlink+(X))-C	x	x	x	u	u	x	F5	2	4
SBC	addr	Absolute	(A)←(A)-(addr)-C	x	x	x	u	u	x	ED	3	4
SBC	addr,X	Absolute,X	(A)←(A)-(addr+(X))-C	x	x	x	u	u	x	FD	3	4[a]
SBC	addr,Y	Absolute,Y	(A)←(A)-(addr+(Y))-C	x	x	x	u	u	x	F9	3	4[a]
SBC	(zlink,X)	(IND,X)	(A)←(A)-((zlink+(X)))-C	x	x	x	u	u	x	E1	2	6
SBC	(zlink),Y	(IND),Y	(A)←(A)-((zlink)+(Y))-C	x	x	x	u	u	x	F1	2	5[a]
Subtract memory from accumulator with borrow. The carry bit reflects the complement of the borrow.												
SEC		Implied	(C)←1	u	u	1	u	u	u	38	1	2
Set carry bit.												
SED		Implied	(D)←1	u	u	u	u	1	u	F8	1	2
Set decimal mode.												

Mnemonic	Operand	Addr Mode	Operation	N	Z	C	I	D	V	OP Code	No. Bytes	No. Cycles
Branch Instructions												
BCC	disp	Relative	if C=0 then(PC)←(PC)+disp	u	u	u	u	u	u	90	2	2^b
BCC	addr	Relative	if C=0 then(PC)←addr	u	u	u	u	u	u	90	2	2^b
Branch if carry bit is clear.												
BCS	disp	Relative	if C=1 then(PC)←(PC)+disp	u	u	u	u	u	u	B0	2	2^b
BCS	addr	Relative	if C=1 then (PC)←addr	u	u	u	u	u	u	B0	2	2^b
Branch if carry bit is set.												
BEQ	disp	Relative	if Z=1 then(PC)←(PC)+disp	u	u	u	u	u	u	F0	2	2^b
BEQ	addr	Relative	if Z=1 then(PC)←addr	u	u	u	u	u	u	F0	2	2^b
Branch on zero result.												
BMI	disp	Relative	if N=1 then(PC)←(PC)+disp	u	u	u	u	u	u	30	2	2^b
BMI	addr	Relative	if N=1 then(PC)←addr	u	u	u	u	u	u	30	2	2^b
Branch on negative result.												
BNE	disp	Relative	if Z=0 then(PC)←(PC)+disp	u	u	u	u	u	u	D0	2	2^b
BNE	addr	Relative	if Z=0 then(PC)←addr	u	u	u	u	u	u	D0	2	2^b
Branch on non-zero result.												
BPL	disp	Relative	if N=0 then(PC)←(PC)+disp	u	u	u	u	u	u	10	2	2^b
BPL	addr	Relative	if N=0 then(PC)←addr	u	u	u	u	u	u	10	2	2^b
Branch on non-negative result.												
BVC	disp	Relative	if V=0 then(PC)←(PC)+disp	u	u	u	u	u	u	50	2	2^b
BVC	addr	Relative	if V=0 then(PC)←addr	u	u	u	u	u	u	50	2	2^b
Branch if overflow bit is clear.												
BVS	disp	Relative	if V=1 then(PC)←(PC)+disp	u	u	u	u	u	u	70	2	2^b
BVS	addr	Relative	if V=1 then(PC)←addr	u	u	u	u	u	u	70	2	2^b
Branch if overflow bit is set.												
JMP	addr	Absolute	(PC)←addr	u	u	u	u	u	u	4C	3	3
JMP	(addr)	Indirect	(PC)←(addr)	u	u	u	u	u	u	6C	3	5
Jump.												

Mnemonic	Operand	Addr Mode	Operation	N	Z	C	I	D	V	OP Code	No. Bytes	No. Cycles
Comparison Instruction												
BIT	zasm	Z page	(A)&(zasm)	7	x	u	u	u	6	24	2	4
BIT	@zlink	Z page	(A)&(zlink)	7	x	u	u	u	6	24	2	4
BIT	addr	Absolute	(A)&(addr)	7	x	u	u	u	6	2C	3	4
Compare accumulator bits with memory. Only the status bits are affected.												
CMP	#data8	Immediate	(A)-data8	x	x	x	u	u	u	C9	2	2
CMP	zasm	Z page	(A)-(zasm)	x	x	x	u	u	u	C5	2	3
CMP	@zlink	Z page	(A)-(zlink)	x	x	x	u	u	u	C5	2	3
CMP	zasm,X	Z page,X	(A)-(zasm+(X))	x	x	x	u	u	u	D5	2	4
CMP	@zlink,X	Z page,X	(A)-(zlink+(X))	x	x	x	u	u	u	D5	2	4
CMP	addr	Absolute	(A)-(addr)	x	x	x	u	u	u	CD	3	4
CMP	addr,X	Absolute,X	(A)-(addr+(X))	x	x	x	u	u	u	DD	3	4[a]
CMP	addr,Y	Absolute,Y	(A)-(addr+(Y))	x	x	x	u	u	u	D9	3	4[a]
CMP	(zlink,X)	(IND,X)	(A)-((zlink+(X)))	x	x	x	u	u	u	C1	2	6
CMP	(zlink),Y	(IND),Y	(A)-((zlink)+(Y))	x	x	x	u	u	u	D1	2	5[a]
Compare memory and accumulator. Only the status bits are affected.												
CPX	#data8	Immediate	(X)-data8	x	x	x	u	u	u	E0	2	2
CPX	zasm	Z page	(X)-(zasm)	x	x	x	u	u	u	E4	2	3
CPX	@zlink	Z page	(X)-(zlink)	x	x	x	u	u	u	E4	2	3
CPX	addr	Absolute	(X)-(addr)	x	x	x	u	u	u	EC	3	4
Compare memory and index register X. Only the status bits are affected.												
CPY	#data8	Immediate	(Y)-data8	x	x	x	u	u	u	C0	2	2
CPY	zasm	Z page	(Y)-(zasm)	x	x	x	u	u	u	C4	2	3
CPY	@zlink	Z page	(Y)-(zlink)	x	x	x	u	u	u	C4	2	3
CPY	addr	Absolute	(Y)-(addr)	x	x	x	u	u	u	CC	3	4
Compare memory and index register Y. Only status bits are affected.												

Mnemonic	Operand	Addr Mode	Operation	N	Z	C	I	D	V	OP Code	No. Bytes	No. Cycles
Increment and Decrement Instructions												
DEC	zasm	Z page	(zasm)←(zasm)−1	x	x	u	u	u	u	C6	2	5
DEC	@zlink	Z page	(zlink)←(zlink)−1	x	x	u	u	u	u	C6	2	5
DEC	zasm,X	Z page,X	(zasm+(X))←(zasm+(X))−1	x	x	u	u	u	u	D6	2	6
DEC	@zlink,X	Z page,X	(zlink+(X))←(zlink+(X))−1	x	x	u	u	u	u	D6	2	6
DEC	addr	Absolute	(addr)←(addr)−1	x	x	u	u	u	u	CE	3	6
DEC	addr,X	Absolute,X	(addr+(X))←(addr+(X))−1	x	x	u	u	u	u	DE	3	7
Decrement memory.												
DEX		Implied	(X)←(X)−1	x	x	u	u	u	u	CA	1	2
Decrement index register X.												
DEY		Implied	(Y)←(Y)−1	x	x	u	u	u	u	88	1	2
Decrement index register Y.												
INC	zasm	Z page	(zams)←(zasm)+1	x	x	u	u	u	u	E6	2	5
INC	@zlink	Z page	(zlink)←(zlink)+1	x	x	u	u	u	u	E6	2	5
INC	zasm,X	Z page,X	(zasm+(X))←(zasm+(X))+1	x	x	u	u	u	u	F6	2	6
INC	@zlink,X	Z page,X	(zlink+(X))←(zlink+(X))+1	x	x	u	u	u	u	F6	2	6
INC	addr	Absolute	(addr)←(addr)+1	x	x	u	u	u	u	EE	3	6
INC	addr,X	Absolute,X	(addr+(X))←(addr+(X))+1	x	x	u	u	u	u	FE	3	7
Increment memory.												
INX		Implied	(X)←(X)+1	x	x	u	u	u	u	E8	1	2
Increment index register X.												
INY		Implied	(Y)←(Y)+1	x	x	u	u	u	u	C8	1	2
Increment index register Y.												

Mnemonic	Operand	Addr Mode	Operation	N Z C I D V	Op Code	No. Bytes	No. Cycles
Interrupt and Control Instructions							
BRK		Implied	((SP))←(PCH) ((SP)-1)←(PCL) ((SP)-2)←(SR) (SP)←(SP)-3 (PCL)←(FFFE) (PCH)←(FFFF) I←1 B←1	u u u 1 u u	00	1	7
Break, interrupt program: push PC and SR on stack, and jump indirect through addresses FFFE and FFFF.							
CLI		Implied	(I)←0	u u u 0 u u	58	1	2
Clear interrupt disable bit.							
NOP		Implied	no operation	u u u u u u	EA	1	2
No operation.							
RTI		Implied	(SR)←((SP)+1) (PCL)←((SP)+2) (PCH)←((SP)+3) (SP)←(SP)+3 (PC)←(PC)+1	x x x x x x	40	1	6
Return from interrupt: pull SR and return address from stack.							
SEI		Implied	(I)←1	u u u 1 u u	78	1	2
Set interrupt disable status.							

Mnemonic	Operand	Addr Mode	Operation	N	Z	C	I	D	V	Op Code	No. Bytes	No. Cycle
Load, Store, and Transfer Instructions												
LDA	#data8	Immediate	(A)←data8	x	x	u	u	u	u	A9	2	2
LDA	zasm	Z page	(A)←(zasm)	x	x	u	u	u	u	A5	2	3
LDA	@zlink	Z page	(A)←(zlink)	x	x	u	u	u	u	A5	2	3
LDA	zasm,X	Z page,X	(A)←(zasm+(X))	x	x	u	u	u	u	B5	2	4
LDA	@zlink,X	Z page,X	(A)←(zlink+(X))	x	x	u	u	u	u	B5	2	4
LDA	addr	Absolute	(A)←(addr)	x	x	u	u	u	u	AD	3	4
LDA	addr,X	Absolute,X	(A)←(addr+(X))	x	x	u	u	u	u	BD	3	4[a]
LDA	addr,Y	Absolute,Y	(A)←(addr+(Y))	x	x	u	u	u	u	B9	3	4[a]
LDA	(zlink,X)	(IND,X)	(A)←((zlink+(X)))	x	x	u	u	u	u	A1	2	6
LDA	(zlink),Y	(IND),Y	(A)←((zlink)+(X))	x	x	u	u	u	u	B1	2	5[a]
Load accumulator with value.												
LDX	#data8	Immediate	(X)←data8	x	x	u	u	u	u	A2	2	2
LDX	zasm	Z page	(X)←(zasm)	x	x	u	u	u	u	A6	2	3
LDX	@xlink	Z page	(X)←(zlink)	x	x	u	u	u	u	A6	2	3
LDX	zasm,Y	Z page,Y	(X)←(zasm+(Y))	x	x	u	u	u	u	B6	2	4
LDX	@zlink,Y	Z page,Y	(X)←(zlink+(Y))	x	x	u	u	u	u	B6	2	4
LDX	addr	Absolute	(X)←(addr)	x	x	u	u	u	u	AE	3	4
LDX	addr,Y	Absolute,Y	(X)←(addr+(Y))	x	x	u	u	u	u	BE	3	4[a]
Load index register X with value.												
LDY	#data8	Immediate	(Y)←data8	x	x	u	u	u	u	A0	2	2
LDY	zasm	Z page	(Y)←(zasm)	x	x	u	u	u	u	A4	2	3
LDY	@zlink	Z page	(Y)←(zlink)	x	x	u	u	u	u	A4	2	3
LDY	zasm,X	Z page,X	(Y)←(zasm+(X))	x	x	u	u	u	u	B4	2	4
LDY	@zlink,X	Z page,X	(y)←(zlink+(X))	x	x	u	u	u	u	B4	2	4
LDY	addr	Absolute	(Y)←(addr)	x	x	u	u	u	u	AC	3	4
LDY	addr,X	Absolute,X	(Y)←(addr+(X))	x	x	u	u	u	u	BC	3	4[a]
Load index register Y with value.												
STA	zasm	Z page	(zasm)←(A)	u	u	u	u	u	u	85	2	3
STA	@zlink	Z page	(zlink)←(A)	u	u	u	u	u	u	85	2	3
STA	zasm,X	Z page,X	(zasm+(X))←(A)	u	u	u	u	u	u	95	2	4
STA	@zlink,X	Z page,X	(zlink+(X))←(A)	u	u	u	u	u	u	95	2	4
STA	addr	Absolute	(addr)←(A)	u	u	u	u	u	u	8D	3	4
STA	addr,X	Absolute,X	(addr+(X))←(A)	u	u	u	u	u	u	9D	3	5
STA	addr,Y	Absolute,Y	(addr+(Y))←(A)	u	u	u	u	u	u	99	3	5
STA	(zlink,X)	(IND,X)	((zlink+(X)))←(A)	u	u	u	u	u	u	81	2	6
STA	(zlink),Y	(IND),Y	((zlink)+(Y))←(A)	u	u	u	u	u	u	91	2	6
Store accumulator in memory.												

Mnemonic	Operand	Addr Mode	Operation	N	Z	C	I	D	V	Op Code	No. Bytes	No. Cycles
Load, Store, and Transfer Instructions												
STX	zasm	Z page	(zasm)←(X)	u	u	u	u	u	u	86	2	3
STX	@zlink	Z page	(zlink)←(X)	u	u	u	u	u	u	86	2	3
STX	zasm,Y	Z page,Y	(zasm+(Y))←(X)	u	u	u	u	u	u	96	2	4
STX	@zlink,Y	Z page,Y	(zlink+(Y))←(X)	u	u	u	u	u	u	96	2	4
STX	addr	Absolute	(addr)←(X)	u	u	u	u	u	u	8E	3	4
Store index register X in memory.												
STY	zasm	Z page	(zasm)←(Y)	u	u	u	u	u	u	84	2	3
STY	@zlink	Z page	(zlink)←(Y)	u	u	u	u	u	u	84	2	3
STY	zasm,X	Z page,X	(zasm+(X))←(Y)	u	u	u	u	u	u	94	2	4
STY	zlink,X	Z page,X	(zlink+(X))←(Y)	u	u	u	u	u	u	94	2	4
STY	addr	Absolute	(addr)←(Y)	u	u	u	u	u	u	8C	3	4
Store index register Y in memory.												
TAX		Implied	(X)←(A)	x	x	u	u	u	u	AA	1	2
Transfer accumulator to index register X.												
TAY		Implied	(Y)←(A)	x	x	u	u	u	u	A8	1	2
Transfer accumulator to index register Y.												
TXA		Implied	(A)←(X)	x	x	u	u	u	u	8A	1	2
Transfer index register X to accumulator.												
TYA		Implied	(A)←(Y)	x	x	u	u	u	u	98	1	2
Transfer index register Y to accumulator.												

Mnemonic	Operand	Addr Mode	Operation	N	Z	C	I	D	V	OP Code	No. Bytes	No. Cycles
Logical Instructions												
AND	#data8	Immediate	(A)←(A)&data8	x	x	u	u	u	u	29	2	2
AND	zasm	Z page	(A)←(A)&(zasm)	x	x	u	u	u	u	25	2	3
AND	@zlink	Z page	(A)←(A)&(zlink)	x	x	u	u	u	u	25	2	3
AND	zasm,X	Z page,X	(A)←(A)&(zasm+(X))	x	x	u	u	u	u	35	2	4
AND	@zlink,X	Z page,X	(A)←(A)&(zlink+(X))	x	x	u	u	u	u	35	2	4
AND	addr	Absolute	(A)←(A)&(addr)	x	x	u	u	u	u	2D	3	4
AND	addr,X	Absolute,X	(A)←(A)&(addr+(X))	x	x	u	u	u	u	3D	3	4[a]
AND	addr,Y	Absolute,Y	(A)←(A)&(addr+(Y))	x	x	u	u	u	u	39	3	4[a]
AND	(zlink,X)	(IND,X)	(A)←(A)&((zlink+(X)))	x	x	u	u	u	u	21	2	6
AND	(zlink),Y	(IND),Y	(A)←(A)&((zlink)+(Y))	x	x	u	u	u	u	31	2	5
AND memory with accumulator.												
EOR	#data8	Immediate	(A)←(A)!!data8	x	x	u	u	u	u	49	2	2
EOR	zasm	Z page	(A)←(A)!!(zasm)	x	x	u	u	u	u	45	2	3
EOR	@zlink	Z page	(A)←(A)!!(zlink)	x	x	u	u	u	u	45	2	3
EOR	zasm,X	Z page,X	(A)←(A)!!(zasm+(X))	x	x	u	u	u	u	55	2	4
EOR	@zlink,X	Z page,X	(A)←(A)!!(zlink+(X))	x	x	u	u	u	u	55	2	4
EOR	addr	Absolute	(A)←(A)!!(addr)	x	x	u	u	u	u	4D	3	4
EOR	addr,X	Absolute,X	(A)←(A)!!(addr+(X))	x	x	u	u	u	u	5D	3	4[a]
EOR	addr,Y	Absolute,Y	(A)←(A)!!(addr+(Y))	x	x	u	u	u	u	59	3	4[a]
EOR	(zlink,X)	(IND,X)	(A)←(A)!!((zlink+(X)))	x	x	u	u	u	u	41	2	6
EOR	(zlink),Y	(IND),Y	(A)←(A)!!((zlink)+(Y))	x	x	u	u	u	u	51	2	5[a]
Exclusive OR memory with accumulator.												
ORA	#data8	Immediate	(A)←(A)!data8	x	x	u	u	u	u	09	2	2
ORA	zasm	Z page	(A)←(A)!(zasm)	x	x	u	u	u	u	05	2	3
ORA	@zlink	Z page	(A)←(A)!(zlink)	x	x	u	u	u	u	05	2	3
ORA	zasm,X	Z page,X	(A)←(A)!(zasm+(X))	x	x	u	u	u	u	15	2	4
ORA	@zlink,X	Z page,X	(A)←(A)!(zlink+(X))	x	x	u	u	u	u	15	2	4
ORA	addr	Absolute	(A)←(A)!(addr)	x	x	u	u	u	u	0D	3	4
ORA	addr,X	Absolute,X	(A)←(A)!(addr+(X))	x	x	u	u	u	u	1D	3	4[a]
ORA	addr,Y	Absolute,Y	(A)←(A)!(addr+(Y))	x	x	u	u	u	u	19	3	4[a]
ORA	(zlink,X)	(IND,X)	(A)←(A)!((zlink+(X)))	x	x	u	u	u	u	01	2	6
ORA	(zlink),Y	(IND),Y	(A)←(A)!((zlink)+(Y))	x	x	u	u	u	u	11	2	5
OR memory with accumulator.												

Mnemonic	Operand	Addr Mode	Operation	N	Z	C	I	D	V	OP Code	No. Bytes	No. Cycles
Shift and Rotate Instructions												
ASL			(C)←(data7)									
			(data7 to data1)←(data6 to data0)									
			(data0)←0									
			where **data** is:									
ASL		Accumulator	(A)	x	x	x	u	u	u	0A	1	2
ASL	zasm	Z page	(zasm)	x	x	x	u	u	u	06	2	5
ASL	@zlink	Z page	(zlink)	x	x	x	u	u	u	06	2	5
ASL	zasm,X	Z page,X	(zasm+(X))	x	x	x	u	u	u	16	2	6
ASL	@zlink,X	Z page,X	(zlink+(X))	x	x	x	u	u	u	16	2	6
ASL	addr	Absolute	(addr)	x	x	x	u	u	u	0E	3	6
ASL	addr,X	Absolute,X	(addr+(X))	x	x	x	u	u	u	1E	3	7
Arithmetic shift left.												
LSR			(c)←(data0)									
			(data6 to data0)←(data7 to data1)									
			(data7)←0									
			where **data** is:									
LSR		Accumulator	(A)	0	x	x	u	u	u	4A	1	2
LSR	zasm	Z page	(zasm)	0	x	x	u	u	u	46	2	5
LSR	@zlink	Z page	(zlink)	0	x	x	u	u	u	46	2	5
LSR	zasm,X	Z page,X	(zasm+(X))	0	x	x	u	u	u	56	2	6
LSR	@zlink,X	Z page,X	(zlink+(X))	0	x	x	u	u	u	56	2	6
LSR	addr	Absolute	(addr)	0	x	x	u	u	u	4E	3	6
LSR	addr,X	Absolute,X	(addr+(X))	0	x	x	u	u	u	5E	3	7
Logical shift right.												
ROL			(C)←(data7)									
			(data7 to data1)←(data6 to data0)									
			(data0)←(C)									
			where **data** is:									
ROL		Accumulator	(A)	x	x	x	u	u	u	2A	1	2
ROL	zasm	Z page	(zasm)	x	x	x	u	u	u	26	2	5
ROL	@zasm	Z page	(zlink)	x	x	x	u	u	u	26	2	5
ROL	zasm,X	Z page,X	(zasm+(X))	x	x	x	u	u	u	36	2	6
ROL	@zasm	Z page,X	(zlink+(X))	x	x	x	u	u	u	36	2	6
ROL	addr	Absolute	(addr)	x	x	x	u	u	u	2E	3	6
ROL	addr,X	Absolute,X	(addr+(X))	x	x	x	u	u	u	3E	3	7
Rotate left.												

Mnemonic	Operand	Addr Mode	Operation	N	Z	C	I	D	V	OP Code	No. Bytes	No. Cycles
Shift and Rotate Instruction												
ROR			(C)←(data0)									
			(data6 to data0)←(data7 to data1)									
			(data7)←(C)									
			where **data** is:									
ROR		Accumulator	(A)	x	x	x	u	u	u	6A	1	2
ROR	zasm	Z page	(zasm)	x	x	x	u	u	u	66	2	5
ROR	@zlink	Z page	(zlink)	x	x	x	u	u	u	66	2	5
ROR	zasm,X	Z page,X	(zasm+(X))	x	x	x	u	u	u	76	2	6
ROR	@zlink	Z page,X	(zlink+(X))	x	x	x	u	u	u	76	2	6
ROR	addr	Absolute	(addr)	x	x	x	u	u	u	6E	3	6
ROR	addr	Absolute,X	(addr+(X))	x	x	x	u	u	u	7E	3	7
Rotate right.												
Stack Instructions												
JSR	addr	Absolute	((SP))←(PCH)	u	u	u	u	u	u	20	3	6
			((SP)-1)←(PCL)									
			(SP)←(SP)-2									
			(PC)←addr									
Jump to subroutine: push current PC onto stack, and jump.												
PHA		Implied	((SP))←(A)	u	u	u	u	u	u	48	1	3
			(SP)←(SP)-1									
Push accumulator on stack.												
PHP		Implied	((SP))←(SR)	u	u	u	u	u	u	08	1	3
			(SP)←(SP)-1									
Push processor status on stack.												
PLA		Implied	(A)←((SP)+1)	x	x	u	u	u	u	68	1	4
			(SP)←(SP)+1									
Pull accumulator from stack.												
PLP		Implied	(SR)←((SP)+1)	x	x	x	x	x	x	28	1	4
			(SP)←(SP)+1									
Pull processor status from stack.												
RTS		Implied	(PCL)←((SP)+1)	u	u	u	u	u	u	60	1	6
			(PCH)←((SP)+2)									
			(SP)←(SP)+2									
			(PC)←(PC)+1									
Return from subroutine: pull return address from stack.												
TSX		Implied	(X)←(SP)	x	x	u	u	u	u	BA	1	2
Transfer stack pointer to index register X.												
TXS		Implied	(SP)←(X)	u	u	u	u	u	u	9A	1	2
Transfer index register X to stack pointer.												

[a]**Add 1 if page boundary is crossed when index value is added.**

[b]**Add 1 if branch to same page. Add 2 if branch to different page.**

RESERVED WORDS

The following names may not be used to represent an address, data item, or variable.

6500/1 Mnemonics

ADC	CLD	JSR	RTS
AND	CLI	LDA	SBC
ASL	CLV	LDX	SEC
BCC	CMP	LDY	SED
BCS	CPX	LSR	SEI
BEQ	CPY	NOP	STA
BIT	DEC	ORA	STX
BMI	DEX	PHA	STY
BNE	DEY	PHP	TAX
BPL	EOR	PLA	TAY
BRK	INC	PLP	TSX
BVC	INX	ROL	TXA
BVS	INY	ROR	TXS
CLC	JMP	RTI	TYA

6500/1 Register Names

A	X	Y

Tektronix Assembler Directives, Options and Operators

ABSOLUTE	END	INPAGE	PAGE	STITLE
ASCII	ENDIF	LIST	REPEAT	STRING
BASE	ENDM	LO	RESERVE	SYM
BLOCK	ENDOF	MACRO	RESUME	TITLE
BYTE	ENDR	ME	SCALAR	TRM
CND	EQU	MEG	SECTION	WARNING
COMMON	EXITM	MOD	SEG	WORD
CON	GLOBAL	NAME	SET	
DBG	HI	NCHR	SHL	
DEF	IF	NOLIST	SHR	
ELSE	INCLUDE	ORG	SPACE	

PAGE SIZE

Page size for the 6500/1 assembler is 256 bytes.

ERROR MESSAGES

The following error messages apply only to the 6500/1 assembler.

*****ERROR 247: **Branch out of range.** The address to be branched to is more than 126 bytes backward or 129 bytes foward from the current pointer location.

*****ERROR 248: **Operand too complex.** The value of the operand is too large, or a syntax error exists in the current operand. An operand that starts with a left parenthesis may only have 40 symbols and constants before the closing right parenthesis.

*****ERROR 249: **Invalid Branch address.** The address to be branched to is either out of range or in another section, or the address expression contains a HI, LO, or ENDOF function.

*****ERROR 250: **Address not page zero.** The value is absolute and not in the range 0-255, but has been used in a context requiring a page zero address.

*****ERROR 251: **Invalid operand.** The operand is invalid for the given instruction.

*****ERROR 252: **Missing zero page address.** The value folowing the "@" is missing, too large, or not an address.

*****ERROR 253: **Invalid index register.** The index register specified is invalid for the given instruction.

*****ERROR 254: **Invalid immediate value.** The value following the # sign is either missing or not in the range 0-255.