Paper 607

An Assembler Written in SAS®

Ed Heaton, Westat, Rockville, MD

ABSTRACT

Is SAS a programming language, i.e. can it handle problems typically in the domain of a real language like C? To prove the point, when I was asked to write an assembler for a hypothetical machine (Leland Beck's **SIC/XE**) as a class project for a Systems Programming class at Johns Hopkins University, I chose to write the program in SAS.

This paper will present the programming strategy, techniques, and problems encountered in carrying out the task. Structured programming methods will be emphasized. Macro is used. Numerous functions and formats will be encountered along the road to success. In short, a lot of basic SAS is used and can be learned in this presentation.

THE HYPOTHETICAL MACHINE

This assembler was written to produce object code for the hypothetical machine described in Leland L. Beck's book, "*An Introduction to Systems Programming*." That text describes two machines: a basic machine called **SIC** (Simplified Instructional Computer) and an extended-feature machine called **SIC/XE**. This program will assemble code for either machine.

THE SIC

The features of the **SIC** machine include:

1. Memory – 8-bit bytes in 3-byte words for a total of 2^{15} bytes.
2. Registers – 24-bits each consisting of:
   a. **A** (0) – accumulator (used for arithmetic operations);
   b. **X** (1) – index register (used for addressing);
   c. **L** (2) – linkage register (the Jump to Subroutine (**JSUB**) instruction stores the return address in this register);
   d. **PC** (8) – program counter (contains the address of the next instruction to be fetched for execution); and
   e. **SW** (9) – status word (contains a variety of information, including a Condition Code).
3. Data formats:
   a. Character – strings of 8-byte ASCII codes and
   b. Integer – 24-bit, 2's complement.
4. Instruction formats (24-bit) consisting of:
   a. first 8 bits contain the operation code (**OpCode**);
   b. 9th bit is a flag (**X**) to indicate indexed-addressing; and
   c. last 15 bits for the address.
5. Addressing modes (re. the **X** bit):
   a. direct – the address is the target address and
   b. indexed – the contents of the **X** register is added to the address to get the target address.
6. An instruction set consisting of 24 distinct instructions.

THE SIC/XE

The **XE** version of the hypothetical machine includes the following features:

1. Memory increased to 2^{20} bytes.
2. Registers – four additional registers:
   a. **B** (3) – base register (used for addressing);
   b. **S** (4) – general working register (no special use);
   c. **T** (5) – another general working register; and
   d. **F** (6) – floating-point accumulator (48 bits).
3. Data formats – allows 48-bit floating-point numbers:
   a. First bit indicates the sign;
   b. Next 11 bits hold the exponent; and
   c. Last 36 bits hold the mantissa.
4. Instruction formats:
   a. Format 1 (1 byte) is simply the **OpCode**;
   b. Format 2 (2 bytes) consists of
      i. 8 bits for the **OpCode**,
      ii. 4 bits for the address of the first register, and
      iii. 4 bits for the address of the second register;
   c. Format 3 (3 bytes) consists of
      i. 6 bits for the **OpCode**,
      ii. a bit-flag (**n**) requests indirect addressing,
      iii. a bit-flag (**i**) specifies immediate operands,
      iv. a bit-flag (**x**) to indicate indexed addressing,
      v. a bit-flag (**b**) for base-relative addressing,
      vi. a bit-flag (**p**) for program-counter-relative addressing,
      vii. a bit-flag (**e**) which must be 0 for this format, and
      viii. 12 bits for the displacement.
   d. Format 4 (4 bytes) consists of
      i. 6 bits for the **OpCode**,
      ii. the same six flags as in the 3-byte instruction, and
      iii. 20 bits for the address.
5. Addressing modes – the **XE** machine allows two additional addressing modes:
   a. Base-relative – the displacement ($0 \le$ displacement ≤ 4095) from the Format-3 instruction is added to the value in the **B** register to get the address in memory and
   b. Program-counter – the displacement ($-2048 \le$ displacement ≤ 2047) from the Format-3 instruction is added to the value in the **PC** register to get the address in memory.
6. Instruction set – the **XE** machine has an additional 35 operation codes.

ASSEMBLY CODE

The format for the assembly code is as follows.

- 1 A decimal point indicates a comment line.
- 1-6 Statement label.
- 8-14 Mnemonic operation code.
- 16-21 Operand.
- 23-60 Comment.
- 16-60 Byte Literal. Hexadecimal literals are of the form
  X'000030'
  and character literals should be of the form
  C'The prime numbers less than 1024.'
  Comments can follow the Byte Literal.

THE ASSEMBLER

Most assemblers make two passes of the source program. This assembler is no exception. The first pass collects label definitions and assigns addresses. The second pass performs most of the work. The top-level macro of the assembler (macro **MAIN** – called at the end of the code) consists mainly of two macro calls.

```
%Macro MAIN ;
   %PassOne()
   %PassTwo()
%Mend MAIN ;
```

We need to generate error messages; so start a macro to do just that. We will add WHEN statements as we go along.

```
%Macro GenerateErrorMessage ( errorNumber ) ;
   error = &errorNumber ;
   Put
      @01 "ERROR("
      @07 error
      @11 "): Line="
      @19 lineNumber
```

```
      @ ;
      Select ( error ) ;
         …
         Otherwise ;
      End ;
   %Mend GenerateErrorMessage ;
```

PASS 1

Ok, now to work. Start with pass one. Here we will define our symbols. The tasks are:

- Assign addresses to all statements in the program;
- Save the values (addresses) assigned to all labels for use in Pass 2; and
- Perform some processing of *assembler directives*. (This includes processing that affects address assignments, such as determining the length of data areas defined by **BYTE**, **RESW**, etc.).

We need to read the source code and create two tables. One is the *symbol table* and the other is simply an intermediate working file. The following DATA step will create the two tables and an ASCII text file of error messages. The **SymTab** table includes the name and the location counter for each statement label in the source program. The **IntermediateFile** contains each soruce statement with its assigned address, error indicators, etc. The error file gets written to by the **%GenerateErrorMessage** macro.

We need to create an array to hold the statement labels so that we can check to assure that they are unique. There can only be one label per statement, so set the length of the array is equal to the number of statements. Add the task of finding the number of statements to the macro before the DATA step.

```
   %Macro PassOne () ;
      %GetNumberOfProgramStatements()
      Data
         %DefineSymTab()
         %DefineIntermediateFile()
      ;
         File Errors ;
         %DefinePassOneColumns()
         InFile Program
            lRecL=60
            pad
            end=_lastRecord
         ;
         Input @ ;
         lineNumber = _n_ ;
         code = _inFile_ ;
         If ( subStr( code , 1 , 1 ) eq "." )
            then go to FINISH ;
            Else do ;
               %ProcessAStatement()
            End ;
   FINISH: Output library.IntermediateFile ;
      Run ;
   %Mend PassOne ;
```

The input to the DATA step is the assembly program. The record length is 60 characters; I like to use LRECL and PAD to ward off input errors. The **END** statement is not required in the assembly language, so we used END= to mark the last record.

Notice that we did not really input anything. We just read the record into the input buffer. The contents of the input buffer is simply a character string; we can process this character string like we could any character variable.

Lines beginning with "." contain comments only.

That's the end of the DATA step and the first pass.

PASS 2

In the second pass we will assemble the instructions and generate the object program. The tasks are:

- Assemble instructions (translating operation codes and looking up addresses).
- Generate data values defined by **BYTE**, **WORD**, etc.
- Perform additional processing of assembler directives.
- Write the object program and the assembly listing.

```
   %Macro PassTwo () ;
      %GetLabelAddresses()
      %SetBaseAddresses()
      %If &XEFlag %then %do ;
         %CalculateDisplacement()
      %End ;
      %CreateInstructionCodes()
      %CreateObjectCodes()
      %WriteProgramListing()
      %WriteObjectCodeFile()
   %Mend PassTwo ;
```

That's it. Done.

DETAILS

Oh, yeah. We need to define these macros that we used.

SUPPORTING MACROS

GET NUMBER OF PROGRAM STATEMENTS

We chose to store the labels in an array and in the **SymTab** table. The array is used to assure that the label is unique. To set the upper bound for the array, scan the assembler code, reading only the first byte of each record into the buffer. Then save the number of records in a macro variable. Here is a macro to get this upper bound. Comments start with a decimal point (**.**); let's not include them in the count.

```
   %Macro GetNumberOfProgramStatements () ;
      %Global statements ;
      Data _null_ ;
         InFile program lRecL=1 end=eof ;
         Input ;
         If ( _inFile_ ne "." ) then
            statements ++1
         ;
         If eof then call symPut(
            "statements"
          , left( put( statements , 8. ) )
         ) ;
      Run ;
   %Mend GetNumberOfProgramStatements ;
```

DEFINE SYMTAB

The symbol table includes the name and value (address) for each label in the source program.

```
   %Macro DefineSymTab () ;
      library.symTab (
         label= "Symbol Table"
         index= ( label )
         keep= label locCtr
         where= ( label not in ( " " , "." ) )
      )
   %Mend DefineSymTab ;
```

DEFINE INTERMEDIATE FILE

Pass 1 usually writes an intermediate file that contains each source statement together with its assigned address, error indicators, etc.

```
   %Macro DefineIntermediateFile () ;
      library.IntermediateFile (
         label= "Intermediate File"
         keep=
            lineNumber loc code locCtr label
            mnemonic operand literal byteString
            nFlag iFlag xFlag bFlag pFlag eFlag
            eFlagShift format2Flag error
      )
   %Mend DefineIntermediateFile ;
```

DEFINE PASS-ONE COLUMNS

Let's use an **ATTRIB** statement to define the columns of both tables output by the **%PassOne** macro.

```
   %Macro DefinePassOneColumns () ;
      Attrib
         lineNumber
            label="Line Number"
            format=z4.
         loc
```

```
            label="Address of Instruction"
            format=hex6.
         Code
            label="Code"
            length=$60
            format=$60.
         locCtr
            label="Location Counter"
            format=hex6.
         label
            label="Label"
            length=$6
            format=$6.
         mnemonic
            label="Mnemonic Operation Code"
            length=$6
            format=$6.
         operand
            label="Operand"
            length=$8
            format=$8.
         literal
            label="Literal Value"
            format=hex6.
         byteString
            label="Literal Value Byte String"
            length=$84
         nFlag
            label="Indirect Flag"
         iFlag
            label="Immediate Flag"
         xFlag
            label="Index Flag"
         bFlag
            label="Base-Relative Flag"
         pFlag
            label="Program-Counter Flag"
         eFlag
            label="Extended-Addressing Flag"
         eFlagShift
            label="Extended-Address Bit-Shifter"
            format=hex4.
         format2Flag
            label="Format 2 Flag"
         error
            label="Error Code"
            format=z4.
      ;
      Length _label1-_label&statements $6 ;
      Array symTab [*] $
         _label1-_label&statements
      ;
      Retain _firstRecord 1 ;
      Retain locCtr _label1-_label&statements ;
   %Mend DefinePassOneColumns ;
```

We retained **_firstRecord**; it will keep us informed on how to process the statements. We also retained the location counter (**locCtr**) and buckets for the statement labels (**_label1-_label&statements**).

PROCESS A STATEMENT

Read the characters in columns 1-6 of the assembler code; this is reserved for labels. Then compare the length of the label with all leading and trailing spaces removed with the length of the label with all spaces removed to determine if there are imbedded spaces.

```
   %Macro ProcessAStatement () ;
      Input label $ 1-6 @ ;
      label = upCase( label ) ;
      If (
         length( label ) ne
         length( compress( label ) )
      ) then do ;
         %GenerateErrorMessage( 1 )
      End ;
      %CheckForEFlag()
      %ReadMnemonic()
      Select ;
         When ( _firstRecord ) do ;
            %ProcessFirstStatement()
         End ;
         When ( mnemonic eq "END" ) do ;
            %ProcessEndStatement()
         End ;
         Otherwise do ;
            %ProcessMiddleStatement()
         End ;
      End ;
   %Mend ProcessAStatement ;
```

We checked to see if the **E** flag needed to be set because extended format instructions are four bytes long. We also read the value in the field for mnemonic operation codes; if it contains the **END** directive, we need to proceed differently.

The **%GenerateErrorMessage** macro will write error messages; let's put this message in the macro before we forget.

```
When ( 1 ) put @24
   "Label contains imbedded spaces."
;
```

GET LABEL ADDRESSES

Add the location counter values from the *Symbol Table* to the records that have a symbol in the *Operand Field*. First merge the *location counter* values into the table created by **%PassOne**.

Then we need to generate errors for those assembler statements where the *operation code* requires a valid label (**SIC** machine only).

We will use the **??** modifier to supress SAS error handling as we attempt to read the *operand* into a numeric field. If the conversion works, we will write that number to the *target address*.

We need a list of the *mnemonic operation codes* to be sure that the statement is valid. We will use a macro variable to hold this list. This macro variable will look like the following.

```
"ADD" "ADDF" "ADDR" "AND" ... "TIXR" "WD"
```

The *OpCodes* are stored in a SAS data set called **OpTab**. Let's look at the code to create this macro variable.

```
%Macro GetOpCodes () ;
   %Global opCodes ;
   Proc sql noPrint ;
      Select
         quote( trim(mnemonic) )
            into :opCodes separated by " "
         from library.OpTab
      ;
   Quit ;
%Mend GetOpCodes ;
```

We will call this macro from the **%PassOne** macro.

```
%Macro PassOne () ;
   %GetNumberOfProgramStatements()
   %GetOpCodes()
   Data
      …
   Run ;
%Mend PassOne ;
```

Now, we can code our **%GetLabelAddresses** macro.

```
%Macro GetLabelAddresses () ;
   Proc sql ;
      Create table PassTwo as
         select
            IntermediateFile.*
          , symTab.locCtr as targetAddress
         from library.IntermediateFile
            left join library.symTab
         on (
            IntermediateFile.operand
          = symTab.label
         )
         order by IntermediateFile.lineNumber
      ;
   Quit ;
   Data PassTwo ;
      File Errors mod ;
      Set PassTwo ;
      If (
         not &XEFlag
       & ( mnemonic in ( &opCodes ) )
```

```
         & ( mnemonic not in (
              &standAloneOpCodes
           ) )
         & missing( targetAddress )
        ) then do ;
           %GenerateErrorMessage( 2 )
        End ;
        If (
           missing( targetAddress )
         & not missing( input(operand,?? 9.) )
        ) then
           targetAddress = input(operand,?? 9.)
        ;
     Run ;
  %Mend GetLabelAddresses ;
```

Add the error message to the **%GenerateErrorMessage** macro.

```
  When ( 2 ) put @24
     "Your operand is not a defined label."
  ;
```

SET BASE ADDRESSES

Search for the **BASE** assembler directive. If found, store the address for computing *base-relative* addressing.

```
  %Macro SetBaseAddresses () ;
     Data PassTwo ;
        Set PassTwo ;
        Attrib
           baseAddr
              label= "Base Address"
              length= 4
              format= hex6.
        ;
        Retain baseAddr ;
        If ( mnemonic eq "BASE" ) then do ;
           baseAddr = targetAddress ;
           targetAddress = . ;
        End ;
     Run ;
  %Mend SetBaseAddresses ;
```

CALCULATE DISPLACEMENT

If we are assembling code for an **XE** machine, then we need to calculate the displacement for *relative* addressing.

If the *operand* field contains a number, we do not want to calculate a displacement. To test for the number, attempt to convert the value to a number with the INPUT function. Use the ?? format modifier to suppress error messages. If the field does not contain a number, the the INPUT function will return a missing value which can be tested with the MISSING function.

We will first test to see if we can use *program-counter-relative* addressing. If we cannot, we will test to see if we can use *base-relative* addressing. If we can't use either, then the **E** flag had better be set.

```
  %Macro CalculateDisplacement () ;
     Data PassTwo ;
        Set PassTwo ;
        File Errors mod ;
        If missing(
           input( operand , ?? 9. )
        ) then select ;
           When (
              -2048 le
              ( targetAddress - locCtr ) le
              2047
           ) do ;
              pFlag = 1 ;
              targetAddress =
                 targetAddress - locCtr
              ;
              If ( targetAddress lt 0 ) then
                 targetAddress =
                    input( subStr( put(
                       targetAddress , hex6.
                    ) , 4 )
                  , hex6.
                 )
              ;
           End ;
           When ( 0 le (
              targetAddress - baseAddr
           ) le 4095 ) do ;
              bFlag = 1 ;
              targetAddress =
                 targetAddress - baseAddr
               ;
           End ;
           Otherwise if (
              missing( eFlag )
            & ( upCase( mnemonic ) ne "END" )
            & not missing( targetAddress )
           ) then do ;
              %GenerateErrorMessage( 3 )
           End ;
        End ;
     Run ;
  %Mend CalculateDisplacement ;
```

Add the error message to the **%GenerateErrorMessage** macro.

```
  When ( 3 ) put @24
     "You must use extended format for direct"
     " addressing."
  ;
```

CREATE INSTRUCTION CODES

This macro will combine the *OpCode*, the address, and, if necessary, an index flag to create an *Instruction Code*.

- The **n** flag bit is used to indicate *indirect-addressing* mode. It is the 18th bit from the right. E.g.:
  0000 0010 0000 0000 0000 0000 -> X'020000'
- The **i** flag bit is used to indicate *immediate-addressing* mode. It is the 17th bit from the right. E.g.:
  0000 0001 0000 0000 0000 0000 -> X'010000'
- The **x** flag bit is used to indicate *indexed-addressing* mode. It is the 16th bit from the right. E.g.:
  0000 0000 1000 0000 0000 0000 -> X'008000'
- The **b** flag bit is used to indicate *base-relative-addressing* mode. It is the 15th bit from the right. E.g.:
  0000 0000 0100 0000 0000 0000 -> X'004000'
- The **p** flag bit is used to indicate *program-counter-addressing* mode. It is the 14th bit from the right. E.g.:
  0000 0000 0010 0000 0000 0000 -> X'002000'
- The **e** flag bit is used to indicate *extended-addressing* mode. It is the 21st bit from the right of this 4-byte instruction. E.g.:
  0000 0000 0001 0000 0000 0000 0000 0000 -> X'00100000'

```
  %Macro CreateInstructionCodes ;
     Proc sql ;
        Create table library.PassTwo as
           select
              PassTwo.*
            , OpTab.opCode
            , sum(
                 OpTab.opCode
                  * input( "010000" , hex6. )
                  * eFlagShift
               , PassTwo.nFlag
                  * input( "020000" , hex6. )
                  * eFlagShift
                  * &XEFlag
               , PassTwo.iFlag
                  * input( "010000" , hex6. )
                  * eFlagShift
                  * &XEFlag
               , PassTwo.xFlag
                  * input( "008000" , hex6. )
                  * eFlagShift
               , PassTwo.bFlag
                  * input( "004000" , hex6. )
                  * eFlagShift
                  * &XEFlag
               , PassTwo.pFlag
                  * input( "002000" , hex6. )
                  * eFlagShift
                  * &XEFlag
```

```
                , PassTwo.eFlag
                   * input( "00100000" , hex8. )
                   * &XEFlag
                , PassTwo.targetAddress
               ) as instructionCode format=hex6.
            from PassTwo left join library.OpTab
            on (
              PassTwo.mnemonic eq OpTab.mnemonic
            )
            order by PassTwo.lineNumber
         ;
      Quit ;
   %Mend CreateInstructionCodes ;
```

We have not created the variable **eFlagShift**. When we do, it will be used to left-shift the bits of the *Instruction Code* so that we have a *Format-4* instruction. **EFlagShift** will have values of **X'0001'** or **X'0100'**. If the instruction code is multiplied by the latter, it becomes a 4-byte instruction. Multiplying by the former does nothing.

The flags **nFlag**, **iFlag**, **xFlag**, **bFlag**, **pFlag**, and **eFlag** have values of zero or one. If zero, then zero is added to the sum.

CREATE OBJECT CODES

Let's create character-based, user-readable representations of the object code for our instructions.

```
   %Macro CreateObjectCodes () ;
      Data library.PassTwo ;
         Length objectCode $100 ;
         Set library.PassTwo ;
         If ( mnemonic not in (
            "START","BASE","EQU","ORG","END"
         ) ) then do ;
            If ( eFlag )
               then objectCode = put(
                  instructionCode , hex8.
               ) ;
               Else objectCode = put(
                  instructionCode , hex6.
               ) ;
         End ;
         If format2Flag then objectCode =
            subStr( objectCode , 1 , 4 )
         ;
         If not missing( literal ) then
            objectCode = put( literal , hex6. )
         ;
         If not missing( byteString ) then
            objectCode = byteString
         ;
      Run ;
   %Mend CreateObjectCodes ;
```

WRITE PROGRAM LISTING

Write a file that contains the program listing and the errors. Notice that I used the MOD option so that I could append the error file to the assembler listing. For that listing I simply read a record using the INPUT statement and wrote that record using the _INFILE_ automatic variable.

```
   %WriteProgramListing () ;
      Title2 "Assembler Listing" ;
         Proc printTo print=listing new ;
            Proc print data=library.PassTwo ;
               Id lineNumber ;
               Var loc code objectCode ;
            Run ;
         Proc printTo print=print ;
         Data _null_ ;
            InFile Errors ;
            File listing mod ;
            Input ;
            If ( _n_ eq 1 ) then put //
               "***** Error Messages *****"
            / ;
            Put _inFile_ ;
         Run ;
      Title2 ;
    %Mend WriteProgramListing ;
```

WRITE OBJECT CODE FILE

Create the *Object Code File*. Well, actually create a file of ASCII records of the hexidecimal representation of the object code using a header record, text records, and an end record. We will use two passes of the **library.PassTwo** data set to accomplish this: pass one will write the *Header* and *Text* records; pass two will write the *Modify* and *End* records.

To make the *Object Code File* easier for humans to read, we will insert a caret (**^**) between each instruction. We will use the COMPRESS function to find the length of the object code with the carets removed. This is the true length of the object code, which is limited to 30 bytes (60 half-bytes).

A character string input by a **BYTE** directive can contain more than 30 bytes, so we need to be able to partition that string across two text records.

First we will cycle through the object code for the program instructions, accumulate lines of object code, and write either a header record or text records to the *Object Code File*.

```
   %Macro WriteObjectCodeFile () ;
      Data _null_ ;
         Format
            startingAddress         hex6.
            lengthOfObjectCodeField hex2.
         ;
         Length objectCodeField $90 ;
         Retain startingAddress objectCodeField;
         Retain splitFlag 0 ;
         Set library.PassTwo ( where=(
            subStr( code , 1 , 1 ) ne "."
         ) ) ;
         File objCode ;
         Select ( mnemonic ) ;
            When ( "START" ) do ;
               %WriteHeaderRecord()
            End ;
            When ( "END" ) do ;
               %WriteTextRecord()
            End ;
            Otherwise do ;
               If (
                  mnemonic in ("RESW","RESB")
               ) then do ;
                  %WriteTextRecord()
                  %ProcessTextRecord()
               End ;
               If missing( objectCodeField )
                  then startingAddress = loc
               ;
               If ( (
                  length( compress(
                     objectCodeField , "^"
                  ) ) + length( objectCode )
               ) le 60 )
                  then do ;
                     %AccumulateObjectCode()
                  End ;
                  Else do ;
                     If ( length( compress(
                        objectCodeField , "^"
                     ) ) gt 60 )
                        then do ;
                          %SplitLongByteString()
                          %ProcessTextRecord()
                        End ;
                        Else do ;
                          %WriteTextRecord()
                          %ProcessTextRecord()
                        End ;
                  End ;
               End ;
         End ;
      Run ;
```

Next, cycle through the object code again, searching for *extended-format* flags. When found, write **Modify** records. Write an **End** record when done.

```
      Data _null_ ;
         Length objectCodeField $90 ;
```

```
      Format startingAddress hex6. ;
      Set library.PassTwo ( where=
         ( subStr( code , 1 , 1 ) ne "." )
      ) ;
      File objCode mod ;
      Select ;
         When (
            ( eFlag eq 1 )
          & missing( input(operand, ?? 9.) )
         ) do ;
            %WriteModifyRecord()
         End ;
         When ( mnemonic eq "END" ) do ;
            %WriteEndRecord()
         End ;
         Otherwise ;
      End ;
   Run ;
%Mend WriteObjectCodeFile ;
```

CHECK FOR E FLAG

If the 4-byte extended format (Format 4) is used, the format must be specified with the prefix **+** added to the operation code in the source statement. It is the programmer's responsibility to specify this form of addressing when it is required.

```
%Macro CheckForEFlag () ;
   Input @7 _eFlagField $char1. @ ;
   Select (_eFlagField) ;
      When (" ") eFlagShift = 1 ;
      When ("+") do ;
        eFlag = 1 ;
        eFlagShift = input("0100",hex4.) ;
      End ;
      Otherwise do ;
        %GenerateErrorMessage( 4 )
      End ;
   End ;
%Mend CheckForEFlag ;
```

Of course we don't want to forget to describe error 2 in our **%GenerateErrorMessage** macro.

```
When ( 4 ) put @24
   "Column 7 must be"
   %If (&XEFlag eq 1) %then " a + or" ;
   " blank."
;
```

READ MNEMONIC

The *mnemonic operation code* is restricted to columns 8 through 14. This macro will read the mnemonic code and test to assure that it contains no blanks.

```
%Macro ReadMnemonic () ;
   Input mnemonic $ 8-14 @ ;
   mnemonic = upCase( mnemonic ) ;
   If (
      length(mnemonic) ne
      length( compress(mnemonic) )
   ) then do ;
      %GenerateErrorMessage( 5 )
   End ;
%Mend ReadMnemonic ;
```

Now let's go back to our **%GenerateErrorMessage** macro and describe error 3.

```
When ( 5 ) put @24
   "The Mnemonic contains imbedded spaces."
;
```

PROCESS FIRST STATEMENT

The processing of the first record is unique enough that we have a macro just to read it.

```
%Macro ProcessFirstStatement () ;
   If (mnemonic eq "START")
      then do ;
         %ProcessNOriFlag()
         %ReadOperand()
         locCtr = input( operand , hex6. ) ;
         Call symPut(
            "StrtAddr"
          , trim( left( put(locCtr,hex6.) ) )
         ) ;
         End ;
         Else locCtr = 0 ;
      loc = locCtr ;
      _firstRecord = 0 ;
%Mend ProcessFirstStatement ;
```

PROCESS END STATEMENT

The processing of the last record is unique enough that we have a macro just to read it.

```
%Macro ProcessEndStatement () ;
   loc = locCtr ;
   Call symPut(
      "EndAddr"
    , trim( left( put(locCtr,hex6.) ) )
   ) ;
   % ReadOperand()
%Mend ProcessEndStatement ;
```

PROCESS MIDDLE STATEMENT

First we want to store the address of the instruction. If we have a value in the space reserved for statement labels we need to make sure it is a valid label. Finally, we need to determine if the *OpCode* is a *Format-2* instruction and proceed accordingly.

```
%Macro ProcessMiddleStatement () ;
   loc = locCtr ;
   If ( upCase( label ) ne " " ) then do ;
      %VerifyLabel
   End ;
   If ( mnemonic in ( &opCodes ) )
      then do ;
         %ProcessOpCodeInstruction()
      End ;
      Else do ;
         %ProcessAssemblerDirective()
      End ;
%Mend ProcessMiddleStatement ;
```

WRITE HEADER RECORD

This macro will write a header record. The format of the header record is as follows; the numbers are the columns.

- 1 **H**
- 2 field separator (**^**)
- 3-8 program name
- 9 field separator (**^**)
- 10-15 starting address of object program (hexadecimal)
- 16 field separator (**^**)
- 17-22 length of object probram in bytes (hexadecimal)

```
%Macro WriteHeaderRecord () ;
   _ProgramLength = put(
      input( "&EndAddr" , hex6. ) -
      input( "&StrtAddr" , hex6. )
   , hex6.
   ) ;
   Put
      @01 "H^"
      @03 label
      @09 "^&StrtAddr"
      @16 "^"
      @17 _ProgramLength
   ;
%Mend WriteHeaderRecord ;
```

WRITE TEXT RECORD

This macro will write a text record. The format of the text record is as follows; the numbers are the columns.

- 1 **T**
- 2 field separator (**^**)
- 3-8 starting address for object code in this record (hexadecimal)
- 9 field separator (**^**)
- 10-11 length of object code in this record in bytes (hexadecimal)
- 12 field separator (**^**)
- 13-?? object code, represented in hexadecimal (2 columns

per byte of object code -- This section of the record will have field separators between each section of object code. The object code is limited to 60 half-bytes, but the field separators will lengthen this.)

```
%Macro WriteTextRecord () ;
   lengthOfObjectCodeField = length(
      compress( objectCodeField , "^" )
   ) / 2 ;
   If (
      lengthOfObjectCodeField gt 1
   ) then put
      @01 "T^"
      @03 startingAddress
      @09 "^"
      @10 lengthOfObjectCodeField
      @12 objectCodeField
   ;
%Mend WriteTextRecord ;
```

PROCESS TEXT RECORD

Start a text record by initializing the object code field to a caret followed by the first object code. If the split flag is set, then there is already some object code, so simply append to it.

```
%Macro ProcessTextRecord () ;
   If splitFlag
      then do ;
         objectCodeField =
            trim( objectCodeField )
            || "^" || objectCode
         ;
         splitFlag = 0 ;
      End ;
      Else do ;
        If not missing( objectCode )
           then do ;
              objectCodeField =
                 "^" || objectCode
              ;
              startingAddress = loc ;
           End ;
           Else objectCodeField = " " ;
      End ;
%Mend ProcessTextRecord ;
```

ACCUMULATE OBJECT CODE

Add the new object code to the text record, and increment the **lengthOfObjectCode** variable.

```
%Macro AccumulateObjectCode () ;
   If ( not missing( objectCode ) ) then do ;
      objectCodeField =
         trim(objectCodeField)
         || "^" || objectCode
      ;
      lengthOfObjectCodeField =
         lengthOfObjectCodeField
       + length( objectCode ) / 2
      ;
   End ;
%Mend AccumulateObjectCode ;
```

SPLIT LONG BYTE STRING

First set the split-code flag. Then write the extra object code to the **RemainingObjectCode** variable. Now write the text record with the first 60 half-bytes of object code. Finally, initialize the object code field of the new text record with the bytes of object code that we chopped off from the too-long object code.

```
%Macro SplitLongByteString () ;
   splitFlag = 1 ;
   remainingObjectCode = subStr( compress(
      objectCodeField , "^"
   ) , 61 ) ;
   objectCodeField = subStr(
      objectCodeField , 1 , 61
   ) ;
   %WriteTextRecord
   startingAddress = startingAddress + 30 ;
   objectCodeField =
      "^" || remainingObjectCode
   ;
%Mend SplitLongByteString ;
```

%WRITEMODIFYRECORD

This macro will write a **MODIFY** record. The format of the modify record is as follows.

- 1 **M**
- 2 field separator (**^**)
- 3-8 starting location of the address field to be modified, relative to the beginning of the program (hexadecimal)
- 9 field separator (**^**)
- 10-11 length of the address field to be modified, in half-bytes (hexadecimal)

```
%Macro WriteModifyRecord () ;
   startingAddress = loc + 1 ;
   Put
      @01 "M^"
      @03 startingAddress
      @09 "^"
      @10 "05"
   ;
%Mend WriteModifyRecord ;
```

%WRITEENDRECORD

This macro will write an **END** record.

- 1 **E**
- 2 field separator (**^**)
- 3-8 address of first executable instruction in object program (hexadecimal)

```
%Macro WriteEndRecord () ;
   Put @01 "E^" @03 instructionCode ;
%Mend WriteEndRecord ;
```

PROCESS N OR I FLAG

In our assembler language, *indirect* addressing is indicated by adding the prefix @ to the operand. The **n** bit is set to indicate that the contents stored at this location represent the address of the operand, not the operand itself. A # prefix indicates *immediate* addressing where the target address (not the contents stored at that address) becomes the operand; so we will set the **i** bit.

```
%Macro ProcessNOrIFlag () ;
   Input @15 _NIFlagField $char1. @ ;
   If ( (nFlag + iFlag) ne 0 )
      then select ( _NIFlagField )
   ;
      When ( " " ) do ;
         nFlag = 1 ;
         iFlag = 1 ;
      End ;
      When ( "@" ) do ;
         nFlag = 1 ;
         iFlag = 0 ;
      End ;
      When ( "#" ) do ;
         nFlag = 0 ;
         iFlag = 1 ;
      End ;
      Otherwise do ;
         %GenerateErrorMessage( 6 )
      End ;
   End ;
%Mend ProcessNOrIFlag ;
```

Finnally describe error 6 in the **%GenerateErrorMessage** macro.

```
When ( 6 ) put @24
   "Invalid character in OpCode previx."
 / @30 "Specify @ for Indirect addressing"
 / @30 "or # for Immediate addressing "
;
```

READ OPERAND

This macro will read the operand and set the **x** flag. Things can get a bit tricky here; the operand can be:

- a statement label followed by a comma and an **X** (for *indexed* addressing),
- an integer if the *mnemonic* field contains the **RESB**, **RESW**, or **WORD** directives, or
- a character or hex string if the *mnemonic* field contains the **BYTE** directive.

Not all of these are limited to eight characters, so let's just read the entire remainder of the record and then throw away all that follows the first blank space.

```
%Macro ReadOperand () ;
   Input _opPlus $ 16-60 @ ;
   If not missing( _opPlus ) then
      _opPlus = subStr(
         _opPlus
       , 1
       , index( _opPlus , " " ) - 1
      )
   ;
   If (
      length( _opPlus ) gt 8
   ) then do ;
      %GenerateErrorMessage( 7 )
   End ;
   operand = upCase( _opPlus ) ;
   If index( operand , ",X" )
      then do ;
         operand = subStr(
            operand
          , 1
          , index( operand , "," ) - 1
         ) ;
         xFlag = 1 ;
      End ;
      Else xFlag = 0 ;
   If (
      subStr( left(operand) , 1 , 1 ) eq "="
   ) then do ;
      %GenerateErrorMessage( 8 )
   End ;
   If (
      indexC(
         operand
       , "'"
       , '"'
       , "!@#$%^&*()_=|\:;<>?./~`"
      )
    | ( indexc(operand,"+-") gt 1 )
   ) then do ;
      %GenerateErrorMessage( 9 )
   End ;
   If (
      length( operand ) ne
      length( compress( operand ) )
   ) then do ;
      %GenerateErrorMessage( 10 )
   End ;
%Mend ReadOperand ;
```

So let's add the error descriptions to the **%GenerateErrorMessage** macro.

```
When ( 7 ) put @24
   "Comments must be blank-separated from"
   " the operand."
;
When ( 8 ) put @24
   "This assembler does not support"
   " literals."
;
 When ( 9 ) put @24
   "You have an illegal character in your"
   " operand."
;
When ( 10 ) put @24
   "The operand contains imbedded spaces."
;
```

VERIFY LABEL

The label field should contain only labels. So use the array of previously stored labels to determine that we have no duplication. If the label already exists in **symTab**, there is an error. If we have searched all the previously stored labels and have not found the current label, then we will add the current label to the stored labels and cease looking.

```
%Macro VerifyLabel () ;
   Do _i=1 to dim( symTab ) ;
      If ( label eq symTab[_i] )
         then do ;
            %GenerateErrorMessage( 11 )
         End ;
         Else if missing( symTab[_i] )
         then do;
            symTab[_i] = label ;
            Go to WRITETOSYMTAB ;
         End ;
   End ;
WRITETOSYMTAB: Output library.symTab ;
%Mend VerifyLabel ;
```

Now let's write the error message.

```
When ( 11 ) put @24
   "The label has already been used."
;
```

PROCESS OPCODE INSTRUCTION

Some of the operation codes are 2-byte instructions. We need to know which ones. So let's go back to the beginning of our assembler create a list of the *mnemonics* for these *OpCodes*.

```
%Let Format2OpCodes =
   "CLEAR"
 , "COMPR"
 , "DIVR"
 , "MULR"
 , "RMO"
 , "SHIFTL"
 , "SHIFTR"
 , "SUBR"
 , "SVCR"
 , "TIXR"
;
```

Compare the *mnemonic* of the *OpCode* against this list and, if this is a two-byte instruction, set the appropriate flags and increment the *location counter*.

```
%Macro ProcessOpCodeInstruction () ;
   If ( mnemonic in ( &Format2OpCodes ) )
      then do ;
         %ProcessFormat2Instruction()
      End ;
      Else do ;
         format2Flag = 0 ;
         locCtr = sum( locCtr , 3 , eFlag ) ;
         %ProcessNOrIFlag()
         %ReadOperand()
         If (
            ( mnemonic not in (
               &standAloneOpCodes
            ) )
          & missing( operand )
         ) then do ;
            %GenerateErrorMessage( 12 )
         End ;
         If (
            not &XEFlag
          & not missing(
               input( operand , ?? 9. )
            )
         ) then do ;
            %GenerateErrorMessage( 13 )
         End ;
      End ;
%Mend ProcessOpCodeInstruction ;
```

Only the **XE** machine allows numbers in the operand field. So, if we assembled on the basic **SIC** machine, we tested the operand by attempting to write the characters to a number. Use the ?? format modifier to suppress messages to the log.

We used a macro variable called **&standAloneOpCodes** that we need to define. Let's define it at the beginning of our assembler where we defined **&Format2OpCodes**.

```
%Let standAloneOpCodes =
   "FIX"
 , "FLOAT"
 , "HIO"
 , "NORM"
 , "RSUB"
 , "SIO"
 , "TIO"
;
```

Again, we need to add the error descriptions to the **%GenerateErrorMessage** macro.

```
When ( 12 ) put @24 "Operand required." ;
When ( 13 ) put @24
   "Your basic SIC computer cannot use"
   " numbers for operands."
;
```

PROCESS ASSEMBLER DIRECTIVE

Assembler directives tell the assembler how to reserve memory and how to preload that memory. They optionally control addressing modes.

- Let's start with the **WORD** directive. This directive tells the assembler to generate a one-word integer constant.
- Next we have the **RESW** directive. It tells the assembler to reserve the indicated number of words for a data area.
- Now lets process the **RESB** directive to tell the assembler to reserve the indicated number of bytes for the data area.
- Next comes the **BYTE** directive. It is used to generate a character or hexadecimal constant, occupying as many bytes as necessary.
- The **BASE** directive is only for the **SIC/XE** machine which allows *base-relative* addressing. The programmer must tell the assembler what the *base* register will contain during execution of the program so that the assembler can compute displacements.

```
%Macro ProcessAssemblerDirective () ;
   format2Flag = 0 ;
   Select ( mnemonic ) ;
      When ( "WORD" ) do ;
         locCtr ++3 ;
         %ReadOperand()
        If missing(
            input( operand , ?? 9.)
         ) then do ;
            %GenerateErrorMessage( 14 )
         End ;
         Else literal = operand ;
      End ;
      When ( "RESW" ) do;
         %FindLengthOfRESW()
         locCtr = locCtr + _length ;
      End ;
      When ( "RESB" ) do ;
         %FindLengthOfRESB()
         locCtr = locCtr + _length ;
      End ;
      When ( "BYTE" ) do ;
         %ProcessBYTE()
         locCtr = locCtr + _length ;
      End ;
      When ( "BASE" ) do ;
         %ReadOperand()
      End ;
      Otherwise do ;
         %GenerateErrorMessage( 15 )
      End ;
   End ;
```

Again, we need to add the error descriptions to the **%GenerateErrorMessage** macro.

```
When ( 14 ) put @24
   "Undefined mnemonic."
;
When ( 15 ) put @24
   "WORD must be a number."
;
```

PROCESS FORMAT-2 INSTRUCTION

If this is a two-byte instruction, set the appropriate flags and increment the *location counter*. Two-byte instructions allow register addresses as the *operands*; read these addresses.

```
%Macro ProcessFormat2Instruction () ;
   Format2Flag = 1 ;
   NFlag = 0 ;
   IFlag = 0 ;
   LocCtr = LocCtr + 2 ;
   %ReadRegisters()
%Mend ProcessFormat2Instruction ;
```

FIND LENGTH OF RESW

Read the number for the **WORD** directive directly from the assembler statement. Use the ?? input modifier, it will suppress error handling so that anything read that is not the character representation of a number will cause a missing value but no error message will be written to the SAS log.

```
%Macro FindLengthOfRESW () ;
   Input @16 _words ?? @ ;
   If missing( _words )
      then do ;
         %GenerateErrorMessage( 16 )
      End ;
      Else _length = 3 * _words ;
%Mend FindLengthOfRESW ;
```

And add the error message to the **%GenerateErrorMessage** macro.

```
When ( 16 ) put @24
   "Invalid number of reserved words."
;
```

FIND LENGTH OF RESB

Read the number of requested reserved words directly from the assembler statement. Again, use the ?? input modifier.

```
%Macro FindLengthOfRESB () ;
   Input @16 _length ?? @ ;
   If missing( _length ) then do ;
      %GenerateErrorMessage( 17 )
   End ;
%Mend FindLengthOfRESB ;
```

Add the error message to the **%GenerateErrorMessage** macro.

```
When ( 17 ) put @24
   "Invalid number of reserved bytes."
;
```

PROCESS BYTE

To process the **BYTE** directive, we first need to determine if we are reading a character string or a hexadecimal string. Since the operands must start in the 16th column, and since the assembler code is limited to 60 byte statements, we are limited to 43-byte strings.

If the string in the assembler statement is a hexadecimal string, then the length of the memory needed to store the string is half the length of the byte string (It takes two characters to represent a byte.).

If the string in the assembler statement is a character string, then we must convert the character string to a hexadecimal string.

```
%Macro ProcessBYTE () ;
   Length _byteString $42 ;
   Input _hexOrChar $ 16-16 @ ;
   Select ( upCase( _hexOrChar ) ) ;
      When ( "X" ) do ;
         %GetByteString()
         _length = length(_byteString) / 2 ;
         byteString = _byteString ;
      End ;
      When ( "C" ) do ;
         %GetByteString()
         _length = length( _byteString ) ;
         byteString = put(
            trim( _byteString ) , $hex.
         ) ;
      End ;
      Otherwise do ;
```

```
            %GenerateErrorMessage( 18 )
         End ;
      End ;
   %Mend ProcessBYTE ;
```

Add the error message to the **%GenerateErrorMessage** macro.

```
   When ( 18 ) put @24
      "Byte string must be hexadecimal (X) or"
      " character (C)"
   ;
```

READ REGISTERS

This macro will read the registers for the two-byte instructions. It will read the first character of **operand** as the first register and, if the second character of **operand** is a comma, it will then read the third character as the second register.

```
%Macro ReadRegisters () ;
   Input operand $ 16-18 @ ;
   operand = upCase( operand ) ;
   Select ( subStr( operand , 1 , 1 ) ) ;
      %AssignRegisterAddress( r1 )
   End ;
   If (subStr(operand,3,3) eq ",") then do ;
      Select ( subStr(operand,3,3) ) ;
         %AssignRegisterAddress( r2 )
      End ;
   End ;
   operand = left( put( sum(
      ( r1 * input( "1000" , hex4. ) )
    , ( r2 * input( "0100" , hex4. ) )
   ) , 6. ) ) ;
%Mend ReadRegisters ;
```

GET BYTE STRING

Read the remainder of the assembler statement starting with the first character after the **X** or **C** character which indicates the type of byte string. Check to ascertain that the string is delimited by single quote marks. Then strip the quote marks and store the string in **_byteString**.

```
%Macro GetByteString () ;
   Input _byteString $ 17-60 @ ;
   If ( subStr(_byteString,1,1) ne "'" )
      then do ;
         %GenerateErrorMessage( 19 )
      End ;
      Else do ;
         _byteString = subStr(
            _byteString , 2
         ) ;
         If not indexC( _byteString , "'" )
            then do ;
               %GenerateErrorMessage( 19 )
            End ;
            Else _byteString = subStr(
               _byteString
             , 1
             , index( _byteString , "'" ) - 1
            ) ;
         End ;
%Mend GetByteString ;
```

Add the error message to the **%GenerateErrorMessage** macro.

```
   When ( 19 ) put @24
      "Byte string must be enclosed in single"
      " quotes."
   ;
```

ASSIGN REGISTER ADDRESS

This macro will convert the *mnemonic* register name to its address.

```
%Macro AssignRegisterAddress ( register ) ;
   When ( "A" ) &register = 0 ;
   When ( "X" ) &register = 1 ;
   When ( "L" ) &register = 2 ;
   When ( "B" ) &register = 3 ;
   When ( "S" ) &register = 4 ;
   When ( "T" ) &register = 5 ;
   When ( "F" ) &register = 6 ;
   Otherwise do ;
      %GenerateErrorMessage( 20 )
   End ;
%Mend AssignRegisterAddress ;
```

Don't forget the error message.

```
When ( 20 ) put @24
   "Register must be one of"
   " A, X, L, B, S, T, or F."
;
```

CONCLUSION

This assembler operates from a **%MAIN** macro that is called on the last line of the code. Some preliminary initialization code precedes the **%MAIN** macro. This includes definition of global macro variables, creation of a window to get user-supplied information, and creation of references to working libraries and files.

The **%MAIN** macro called the **%PassOne** and **%PassTwo** macros. **%PassOne** generated a list of the mnemonic *operation codes* and stored them in an array. It created another array to store the *statement labels*. **%PassOne** then generated a SAS data set (**IntermediateFile**) containing the *line number*, the location in memory of the code that it will generate for each statement, the *statement label*, the mnemonic *operation code*, the *operand* for the operation codes, *literal values*, and hexadecimal representations of *BYTE* directives. It also produced flags for the *n*, *i*, *x*, *b*, *p*, and *e* bits, and for 2-byte instructions. Finally, **%PassOne** created a symbol table (**SymTab**).

%PassTwo merged the addresses from **SymTab** onto **IntermediateFile** from **%PassOne** using the **%GetLabelAddresses** macro. It assigned base addressees to each record and calculated displacements. It then created hexadecimal instructions and the object code for each assembly statement. Finally, **%PassTwo** wrote the assembler listing, including any error messages, and the object code file.

While this was an exercise for a college class, I hope that you found something to take home with you. We used several data manipulation techniques, and the overall scheme is an example of top-down programming in SAS.

REFERENCES

Beck, Leland L., An Introduction to Systems Programming, 3d ed. (Reading, Massachusetts: Addison Wesley Longman, Inc., 1997).

ACKNOWLEDGMENTS

I want to thank Ian Whitlock of Westat for his continual support and encouragement in my career growth. He was an inspiration before I met him, and has proven to be a wonderful mentor and friend since.

I also want to thank Dianne Rhodes of Westat, who directed my focus toward more career-enhancing facilities such as SAS Users Groups and the SAS-L list-server when we both worked elsewhere.

Finally, I want to thank all the wonderful and insightful contributors to SAS-L for their selfless contributions. They have proven to be my most valuable teaching aid.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Ed Heaton
Westat: An Employee-Owned Research Corporation
1650 Research Boulevard
Rockville, MD 20850
Work Phone: (301) 610-4818
Fax: (301) 294-3992
Email: EdwardHeaton@Westat.com