ASSEMBLER IN A FORTRAN ENVIRONMENT WITH A NEW DEBUGGING AID

Raymond Pavlak, Jr.
Adir Pridor

Rensselaer Polytechnic Institute
Troy, New York

Introduction

An Assembly Language course is generally taken as a second course in computer science. Many students, after finishing successfully the first (introductory) course, still find the Assembly Language course extremely difficult and time-consuming. It is felt that the course should be designed in a manner that allows a student to acquire the basic ideas and techniques, while practicing in actual Assembly programming, within a reasonable amount of effort and time to avoid adverse effects on his other studies. We suggest that two important contributions towards the above goal are (1) employing an approach that conceives any Assembly program as a subroutine called by FORTRAN (or another higher level language used in the first course), and (2) providing basic easy-to-use debugging facilities. In this paper we report on the experience obtained while teaching the Assembly course at Rensselaer Polytechnic Institute according to the above considerations.

Assembly in a FORTRAN Environment

The Assembly Language course at Rensselaer Polytechnic Institute is designed to introduce the student to the machine structure of the IBM 360, the techniques and ideas involved with programming at the Assembly level, the syntax of the IBM Assembler [1], interactions between the users' programs and the Operating System, macro instructions and Input/Output programming. It is felt that the student should be given the opportunity to run his own Assembler program as early as possible. If a student's first program is a main program written in Assembler, then he must be provided with Input/Output instructions, conversion procedures for numerical data types, and possibly some general information about macro instructions, and the referencing of files (data sets) by symbolic names in the Input/Output macro instructions. Trying to base the first program on non-numeric data has the demerit that most students at this stage are neither motivated nor reasonably experienced with alphanumeric processing.

An easier and more productive approach is to begin the course with the fixed-point arithmetic instructions and base the first computer projects on processing integers. The reading of input and, more important, printing of results are done in a main program that calls the Assembler program as a subroutine, and is written in a higher level language like FORTRAN. This way the student does not get into Input/Output procedures, conversions, and operating system considerations until later in the course, thus concentrating at the initial phase on the basics of register usage, addressing, and the elementary operations. Printing the results by a FORTRAN program makes it easy for the beginning student to label his output with explanatory titles, and to edit his results almost without effort. The only subject that may seem difficult to some students is the linkage conventions for subprograms, but since these conventions are used partly also by "main" programs, it appears helpful to introduce the conventions at this early stage, and explain that, to some extent, a "main" program can also be considered a subroutine called by the operating system. Owing to the fact that the students taking the Assembler course have had experience with FORTRAN in their first computer course, this approach was very instructive and effective.

In [3] the handling of Input/Output at the beginning is done by invoking calls to reading and printing subroutines. The above suggestion of using the FORTRAN environment seems more direct and natural. The approach in [2] employs special instructor-prepared macro instructions for reading data from cards and printing lines. This does not avoid the necessary conversions, and the student still has to be introduced to them quite early in the course.

As discussed below, running Assembly programs in a FORTRAN environment also helps in providing the students with debugging facilities, whose vitality at the initial stages cannot be overemphasized. In this respect, the approach is self-consistent: it puts students to actual program running at a very early stage and, at the same time, provides a tool without which debugging would be extremely difficult for the beginner.

Debugging

Perhaps the single most important debugging technique is that of observing intermediate results during program execution. In most high level programming languages this presents no difficulty, just the insertion of output (print) statements at strategic locations in the program. Assembler Language programming, however, presents some rather unique problems. Many Assembler Language programs are intimately related with a particular computer's hardware features and the normal resources used to print diagnostic information may already be in use by the program, thus precluding their use for debugging purposes. On the IBM System/360, in particular, we refer to the registers used for subroutine linkages and addressing purposes.

The basic requirements for an effective debugging system are ease of use and transparency to the program being debugged. By ease of use, it is meant that as few changes as possible be made to the source program and Job Control Language statements. In addition, simplicity, especially for the beginner, should be a major consideration. The novice programmer should not be required to learn the intricate details of the operating system or input/output processing. As far as transparency is concerned, the debugging routines should have a minimal effect on the user's program, and ideally should not require special resource allocations either.

Although OS/360 does provide debugging facilities through the TRACE and SNAP macro instructions, it is felt that their use is difficult for the beginning student. The major problems are their nontransparency and presupposed knowledge of the operating system (JCL) and input/output procedures. Following are some specific examples:

1. The macros may alter General Registers 0, 1, 14, and 15. This is a case of nontransparency to the program being debugged. The beginning student may be unaware of this fact, and might get confused if his program makes use of any of these registers.

2. Both macros assume the establishment of a Save Area and its associated address in General Register 13. This is a specialized resource allocation which the beginning student may very easily forget to do. In fact, this is one of the most common errors. Using either the SNAP or TRACE macro without a valid Save Area and address in Register 13 may cause unpredictable results or even an Addressing Exception.

3. Using the macros requires some knowledge of the operating system and input/output procedures, including the OPEN and DCB macros. This topic inevitably causes the most trouble to Assembler students, and it would seem that trying to teach this to the student in the early stages of the course might serve to overpower the student's comprehension. After all it is rather absurd that debugging be more difficult and complex than the original program.

4. In order to use the macros the student has to modify his Control Cards (JCL). We have found from experience that many new students are initially confused by the Control Cards necessary to run their program. Unfortunately the JCL cards must be modified in order to use the SNAP or TRACE macros. It would seem that this only adds to the complexity of the debugging process, and should be minimized if possible.

As a result of these problems with the currently available debugging software, we propose a macro instruction for the OS/360 Assembler Language, in which we have endeavored to simplify the debugging process as much as possible. This macro has proven to be very effective and useful in the course.

Regdump

Macro REGDUMP has been written in response to a need for a simple debugging aid for Assembler Language programs for new students. The function of REGDUMP is to provide the user with a snapshot of the contents of the 16 General Registers, the 4 Floating Point Registers and the value of the current Condition Code. As explained above, we have found a FORTRAN environment very helpful for teaching a first course in Assembler, and REGDUMP was written with this in mind. REGDUMP can thus be used without modifications to the JCL cards for the run, since the FORTRAN output routines called by REGDUMP are readily available for use. This, then, eliminates one of the sources of trouble to students.

Attacking the next problem, we have attempted to minimize the interference of REGDUMP with the normal operation of the Assembler program being debugged. As detailed below, it does not alter the contents of any registers or other information that the student is likely to ever use.

A typical example of using REGDUMP by a student would be as follows: the student has decided that it would be helpful to observe the contents of the registers at several points in his program. All he does is to insert at those points a card with 'REGDUMP' on it, and run his job. The program will be compiled and the code generated by REGDUMP will produce a register dump each time a REGDUMP call is encountered during execution.

The output from REGDUMP, illustrated by Fig. 1, provides the user with the following information at execution time:

1. General Registers 0-15 are printed in decimal and hexadecimal formats for convenience in interpreting both arithmetic and bit operations.

2. Floating Point Registers 0-6 are printed in both double precision decimal and hexadecimal formats. This facilitates checkout of floating point operations...

3. The current value of the Condition Code is printed in decimal in order to check conditional branches and follow the flow of logic in the program.

4. The current Program Mask is listed in hexadecimal for monitoring possible Program Interrupts and/or to determine what the exact results of some arithmetic operations will yield.

5. Some identification information (detailed below) of the REGDUMP call is printed to allow determining of which REGDUMP call in the course program is being executed, thus following program execution.

6. The address of the last instruction generated by the macro call is given for convenience in locating the macro coding when studying Loader Maps, dumps, or other object listings. This address aids the new student by giving him an address which lies between two consecutive source instructions thereby specifying precisely when the dump occurred.

All output is printed in an easy to read format which is blocked off from the program output by a border thus facilitating differentiation between diagnostic and program output.

In its final form, REGDUMP was chosen to be a macro call. The advantage of using this approach over that of standard subroutine linkage is that the beginning student need not concern himself with all of the linkage details. In REGDUMP, all housekeeping and linkage to auxiliary subroutines is done internally. Thus it is possible to use REGDUMP without the knowledge of linkage conventions.

In order to avoid the pitfall of a student forgetting to establish a Save Area and its address in General Register 13, REGDUMP creates its own internal Save Area. This way, REGDUMP performs normally and the student might detect the trouble by checking the REGDUMP output that contains the contents of register 13. Unreliable results and a possible Addressing Exception are thus avoided.

One of the functions of REGDUMP is to preserve the current status of the program. This is done by copying all pertinent data to a temporary storage area and later restoring this information. In REGDUMP, this data must be saved before the general registers can be set up for subroutine linkage. For this purpose, an inline storage area is generated as part of the REGDUMP expansion.

To minimize the amount of inline code, it was decided to perform the output functions by calling a subroutine external to the macro. In our implementation, the external subroutine was written in FORTRAN because, as explained earlier, REGDUMP was used in a FORTRAN environment. Furthermore, it is much easier for the course instructor to control the output format in FORTRAN.

The inline area begins with the storage area, skipped over by a branch instruction, and followed by a sequence of instructions that stores the relevant information into the storage area, calls the printing subroutine, restores the registers and condition code, and resumes normal execution. The storage area is organized as follows:

WORDS	BYTES	CONTENTS
1-3	0-11	Linkage Data to FORTRAN output subroutine
	8	current program mask and condition code
4-19	12-75	general registers (0-15) when REGDUMP is called
20-27	76-107	floating point registers (0-6)
28-45	108-179	conventional save area for FORTRAN output subroutine
46	180-183	identification information

Since the inline storage area requires over 180 bytes, provisions were made to generate it only once in each control section. Therefore the macro creates a compilation-timetable of the control sections in which it has been called. In the present implementation, there is provision for up to 10 control sections in which the macro may be called. If more than 10 control sections are used (which is rare), the macro will still work, but the full storage area will be generated for each REGDUMP call from the 11th control section onward.

As discussed earlier, just using 'REGDUMP' in the operation-code field is sufficient to invoke the debugging routine. Even with such a simple call, each dump of information will be uniquely identified by an internally generated sequence number. Optionally, the user may specify his own identification number by entering it in the operand field. Other than misspelling 'REGDUMP', it is nearly impossible to commit an error in trying to call REGDUMP. However, should the user supply an invalid identification number, the macro will be sure to print a message to this effect and it will generate a number of its own and specify it to the user. If no identification number was supplied in the call, again the macro will note this and assign a unique integer for an ID and print it for the user's information. Examples of what REGDUMP calls might look like, with the macro reaction shown on the following line appear in Table 1.

To summarize, REGDUMP is absolutely transparent to the student's program in that it keeps unchanged the contents of all registers and the condition code. It provides clear identification information so that the student can easily distinguish between different REGDUMP calls within the same program. Excellent legibility was made possible by blocking off the dump output from any program output that may occur between dumps. Thus by reading the output, the student may actually follow the dynamic flow of his program.

Concluding Remarks

The approach described above of teaching Assembler in a FORTRAN environment together with the REGDUMP macro has proved to be very effective and successful. A significant improvement has been noticed relative to previous years.

A natural addition to REGDUMP would be to enable dumping of storage areas. This was not done in the present implementation in order to make the calling sequence as simple as possible. This extension can be added in an obvious manner.

In order to implement REGDUMP, one should have the FORTRAN printing subroutine in a library consulted by the Loader. Such a standard library exists in almost every installation, and if not, can be appended through the use of a single JCL card.

Figure 2 contains a source listing of REGDUMP and Figure 3 shows the FORTRAN output subroutine.

References

1. IBM Corporation, OS Assembler Language, GC28-6514.

2. W. G. Rudd, Assembly Language Programming and the IBM 360 and 370 Computers, Prentice-Hall Inc., Englewood Cliffs (1976).

3. G. W. Struble, Assembler Language Programming: the IBM System/360 and 370, 2nd ed., Addison Wesley, Reading (1975).

a. REGDUMP (no id given)

```
a.      REGDUMP                                     (no id given)
                *, REGDUMP ID = 0005                (id assigned by REGDUMP)

b.      REGDUMP   25                                (id of 25 specified)
                                                    (no message given)

c.      REGDUMP   XYZ                               (invalid id given)
                *, ILLEGAL REGDUMP ID. 0008 USED.   (legal id assigned)
```

Table 1.

```
-----------------------------------------------------------------------------------------------------------------------
| REGDUMP  25 AT 1E61B6; CC IS 2, PGM MASK IS 2, REGS ARE:                                                            |
|          HEX    00000114     001E607E     00000014     40F423F0     001F5700     00800000     001F57B0     46000014|
|          DEC         276      1990782           20   1089741808      2053888      8388608      2054064   1174405140|
|          HEX    0000E1A0     001F5700     001F57F8     001E6000     421E6016     001E6214     421E6450     001E6010|
|          DEC       57760      2053888      2054136      1990656   1109286934      1991188   1109288016      1990672|
|            41DDC53 0A3C8165           3A2A740E 761D0000          41DDCE9E 00000000          397137A5 00000000   |
|         0.13815508880601630 02     0.99651787604559320-08     0.13862943649291990 02     0.16475294462026110-08|
-----------------------------------------------------------------------------------------------------------------------
| REGDUMP   2 AT 1E6208; CC IS 2, PGM MASK IS 2, REGS ARE:                                                            |
|          HEX    00000114     001E607E     00000014     40F423F0     001F5700     00800000     001F57B0     46000014|
|          DEC         276      1990782           20   1089741808      2053888      8388608      2054064   1174405140|
|          HEX    0000E1A0     001F5700     001F57F8     001E6000     421E6016     001E62C0     421E6450     001E6010|
|          DEC       57760      2053888      2054136      1990656   1109286934      1991360   1109288016      1990672|
|            41DD0C53 0A3C8165          3A2A740E 761D0000          41DDCE9E 00000000          397137A5 00000000   |
|         0.13815508880601630 02     0.98851787604559320-08     0.13862943649291990 02     0.16475294462026110-08|
-----------------------------------------------------------------------------------------------------------------------
          0.99999900E 06           0.13815511E 02           0.13815509E 02
-----------------------------------------------------------------------------------------------------------------------
| REGDUMP  25 AT 1E61B6; CC IS 2, PGM MASK IS 2, REGS ARE:                                                            |
|          HEX    000001FC     001E607E     000000FC     40FFFFFF     001F5700     00800000     001F57B0     460000FC|
|          DEC         508      1990782          252   1090519039      2053888      8388608      2054064   1174405372|
|          HEX    0000E1A0     001F5700     001F57F8     001E6000     421E6016     001E6214     421E6450     001E6010|
|          DEC       57760      2053888      2054136      1990656   1109286934      1991188   1109288016      1990672|
|            42AEAC4F 9AFFFFF1          35100000 00000000          42AEAC4F A0000000          34800000 00000000   |
|         0.17467308968305580 03     0.35527136788005010-14     0.17467308998107910 03     0.17763568394002500-14|
-----------------------------------------------------------------------------------------------------------------------
| REGDUMP   2 AT 1E6208; CC IS 2, PGM MASK IS 2, REGS ARE:                                                            |
|          HEX    000001FC     001E607E     000000FC     40FFFFFF     001F5700     00800000     001F57B0     460000FC|
|          DEC         508      1990782          252   1090519039      2053888      8388608      2054064   1174405372|
|          HEX    0000E1A0     001F5700     001F57F8     001E6000     421E6016     001E62C0     421E645D     001E6010|
|          DEC       57760      2053888      2054136      1990656   1109286934      1991360   1109288016      1990672|
|            42AEAC4F 9AFFFFF1          35100000 00000000          42AEAC4F A0000000          34800000 00000000   |
|         0.17467308968305580 03     0.35527136788005010-14     0.17467308998107910 03     0.17763568394002500-14|
-----------------------------------------------------------------------------------------------------------------------
          0.72370051E 76           0.17467310E 03           0.17467308E 03
```

Figure 1. Sample output from REGDUMP (numbers outside boxes are program output).

```
         MACRO
&NAM     REGDUMP   &ID
.*
.*   REGDUMP PRINTS A SNAP OF THE GENERAL PURPOSE AND FLOATING
.*   POINT REGS DURING EXECUTION.  IT ALSO GIVES THE CONDITION CODE.
.*   PROGRAM MASK. ADDR OF THE LAST INSTR. OF THE MACRO EXPANSION.
.*   AND AN ID NUMBER.  REGDUMP DOES NOT ALTER ANY OF THIS INFORMATION.
.*   REGDUMP CALLS FORTRAN SUBROUTINE REGDMP FOR OUTPUT
.*
.*   REGDUMP WRITTEN BY ADIR FRIDOR AND RAY PAVLAK - 11/14/76
.*
.*   &CS LIST OF CSECT NAMES. ONLY ONE STORAGE AREA FOR EACH CSECT.
.*
         GBLC  &CS(10)
.*
.*   &PC<=&PM, CURRENT NUMBER OF NAMES IN &CS.
.*
         GBLA  &PC,&PM
         LCLA  &P,&IDMOD
.*
.*   &A WILL BE NAME OF STORAGE AREA.
.*
         LCLC  &A,&N
         AIF   (T'&ID EQ 'O').CONT1        TO CHOOSE ID IF OMITTED
         AIF   (T'&ID NE 'N').MESS         CHECK IF ID VALID
         AIF   (&ID LT 0).MESS
&IDMOD   SETA  &ID-((&ID/1000)*1000)       REDUCE MOD 1000
         AGO   .CONT
.MESS    MNOTE *,'ILLEGAL REGDUMP ID. &SYSNDX USED.'
         AGO   .CONT2
.CONT1   MNOTE *,'REGDUMP ID = &SYSNDX'
.CONT2   ANOP
&IDMOD   SETA  &SYSNDX                     CHOOSE ID
.CONT    ANOP
.*
.*   &PM=10 IS LENGTH OF CSECT LIST ( &CS ).
.*
&PM      SETA  10
&P       SETA  1
.*
.*   LOOK FOR CSECT NAME IN TABLE.
.*
.CHKCS   AIF   (&P GT &PC).NEWCS
         AIF   ('&SYSECT' EQ '&CS(&P)').NODEF1
&P       SETA  &P+1
         AGO   .CHKCS
.NODEF1  ANOP
&A       SETC  'CS'.'&P'.'PREG'            CREATE AREA NAME
&N       SETC  '&NAM'                      ATTACH &NAM TO B INSTRUCTION
         AGO   .NODEF
.*
.*   IF &CS NOT FULL, INSERT CSECT NAME.
.*
.NEWCS   AIF   (&PC LT &PM).INSCS
         MNOTE *,'SEPERATE AREA ASSIGNED FOR THIS REGDUMP CALL.'


&A       SETC  'PREG'.'&SYSNDX'            CREATE AREA NAME
         AGO   .DEF
.INSCS   ANOP
&PC      SETA  &PC+1                       ADVANCE COUNTER
&CS(&PC) SETC  '&SYSECT'                   ADD NAME TO LIST
&A       SETC  'CS'.'&PC'.'PREG'           CREATE AREA NAME
.DEF     ANOP
         CNOP  0,8                         SAVE STORAGE AREA
&NAM     B     &A+184                      SKIP AREA
&A       DC    A(*+8)                      ADDRESS PARAM SENT TO REGDMP
         DC    V(REGDMP)
         DS    44F
.NODEF   ANOP
.*
.*   SAVE REGISTERS AND STATUS, GO TO OUTPUT ROUTINE, AND RESTORE DATA
.*
&N       STM   0,15,&A+12                  SAVE AND PASS GENERAL REGISTERS
         LA    1,&A
         STD   0,76(1)                     PASS FLOATING POINT REGISTERS
         STD   2,84(1)
         STD   4,92(1)
         STD   6,100(1)
         LA    13,108(1)                   ESTABLISH SAVE AREA
         L     15,4(1)
         LA    2,&IDMOD                    PASS ID
         ST    2,180(1)
         BALR  2,0                         TAKE CC, MASK, PROG COUNTER
         ST    2,8(1)
         BALR  14,15                       GO TO OUTPUT SUBROUTINE
         LD    0,76(1)                     RESTORE FLOATING POINT REGISTERS
         LD    2,84(1)
         LD    4,92(1)
         LD    6,100(1)
         SPM   2                           RESTORE CC
         LM    0,15,12(1)                  RESTORE GENERAL REGISTERS
         MEND
```

Figure 2. Source listing of REGDUMP.

```
0001              SUBROUTINE REGDMP(M)
            C
            C   PRINTING SUBROUTINE USED BY REGDUMP MACRO.
            C
0002              DATA K24 /Z1000000/
0003              DIMENSION M(44),MM(8)
0004              DOUBLE PRECISION F(4)
0005              EQUIVALENCE (F,MM)
0006              INTEGER ADDR,STATUS,CC
            C
0007                 DO 1 I=1,8
0008        1        MM(I) = M(I+17)
            C
0009              STATUS = M(1)/K24
0010              ADDR = M(1)-STATUS*K24+24
0011              ICC = STATUS/16
0012              MASK = STATUS-16*ICC
0013              CC = ICC-4
            C
            C     M(44) = REGDUMP ID NUMBER
            C     M(2) --> M(17) ARE GENERAL REGS 0-15
            C     M(18) --> M(25) ARE FLOATING POINT REGS
            C
0014              WRITE(6,100)M(44),ADDR,CC,MASK,(M(I),I=2,9),(M(I),I=2,9),
                 X          (M(I),I=10,17),(M(I),I=10,17),(M(I),I=18,25),F
            C
0015        100   FORMAT(
                 A          1X,132('-')
            C
                 B          /' | REGDUMP',I4,' AT ',Z6,'; CC IS',I2,', PGM MASK IS ',
                 C          Z1,', REGS ARE:',73X,'|',
            C
                 D          2(/' |',130X,'|'
                 E            /' |',15X,'HEX',8Z14,'|'
                 F            /' |',15X,'DEC',8I14,'|')
            C
                 G          /' |',130X,'|'
            C
                 I          /' |',15X,4(Z19,Z9),3X,'|'
                 J          /' |',18X,4D28.16,'|'
            C
                 K          /1H ,132('-')
                 L         )
            C
0016              RETURN
0017              END
```

Figure 3. Source listing of the FORTRAN output subroutine.