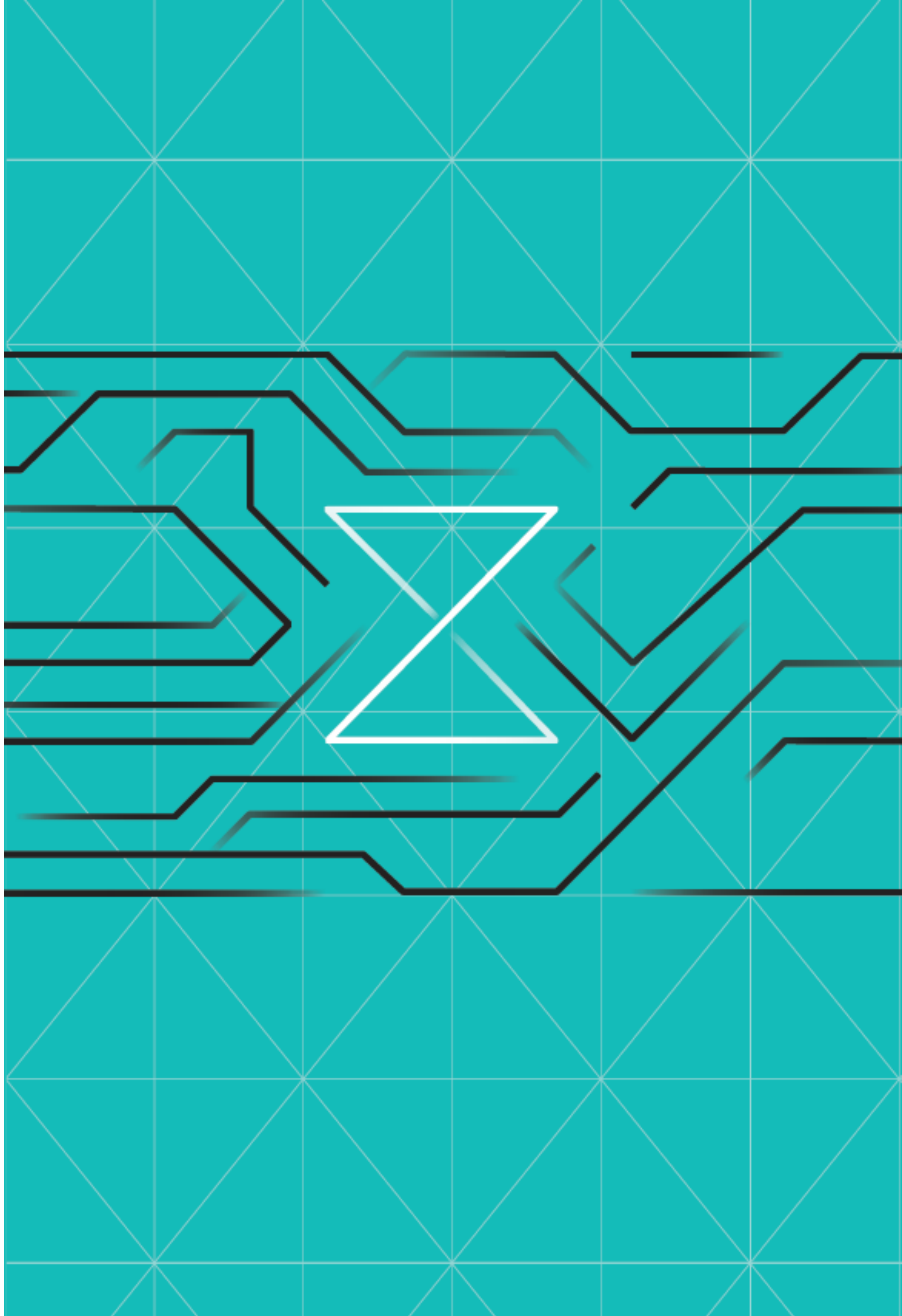
Advanced

230329-1718

ASM1

My first Assembler program



AN INTRODUCTION TO ASSEMBLER PROGRAMMING

A simple example to show that building and running Assembler programs is very similar to the way COBOL, PL/1 and GO programs are produced.

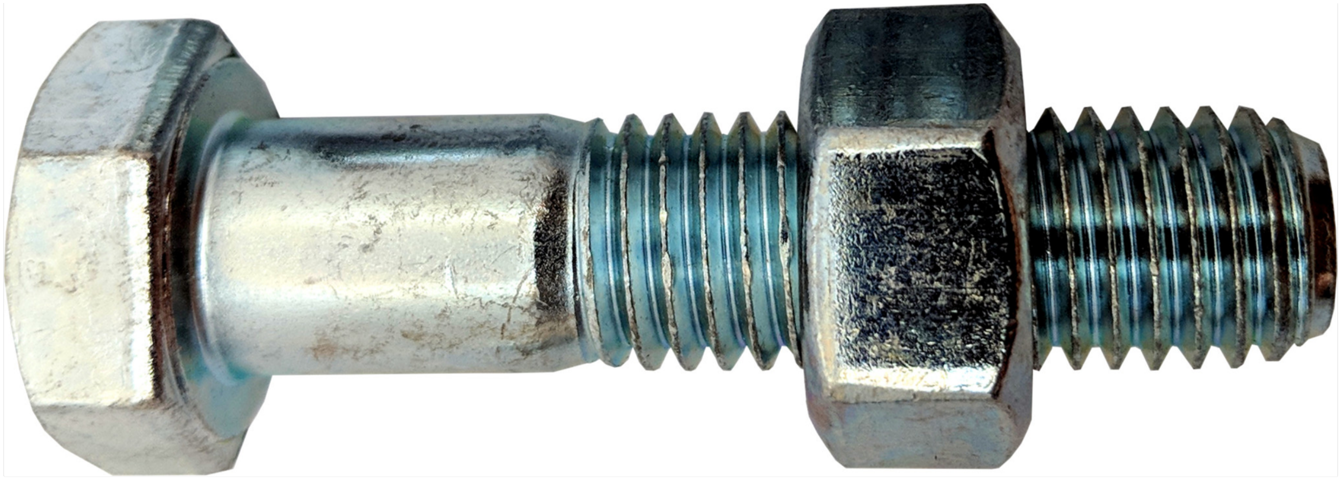
The Challenge

This challenge will give you an introduction to Assembler programming. It will explain the basics of how Assembler Code is compiled and executed.

Investment

Steps	Duration
5	20 minutes

1 AN INTRODUCTION TO ASSEMBLER PROGRAMMING



ASM1|230329-1718

SOME BACKGROUND

A relatively simple assembler program will be used and explained.

Each computer architecture has machine instructions unique to the architecture. All computer languages supported by the architecture must be translated into the unique machine instructions of the underlying computer architecture.

Each computer architecture has an assembly language which includes “mnemonics” that are assembled into machine instructions understood by the computer.

Compilers and interpreters translate supported computer languages into the unique machine instructions understood by the hosting computer.

Computer processing memory is used to load and store the machine instructions and data for processing. The operating system keeps track of processing memory locations using addresses where some of the memory is free, some of the memory has machine instructions, and some of the memory has data.

Higher level languages such as C/C++, Java, COBOL, etc. were created to make programming the computer easier by hiding the complexity of the underlying machine instructions, addressable memory, and registers.

Registers are at the top of the memory hierarchy, and provide the fastest way to access data.

2 THE HEAD …

Below you will find the example code which you will use during this challenge. The code is built to show the first 40 numbers of the Fibonnaci sequence using an easy calculating model.

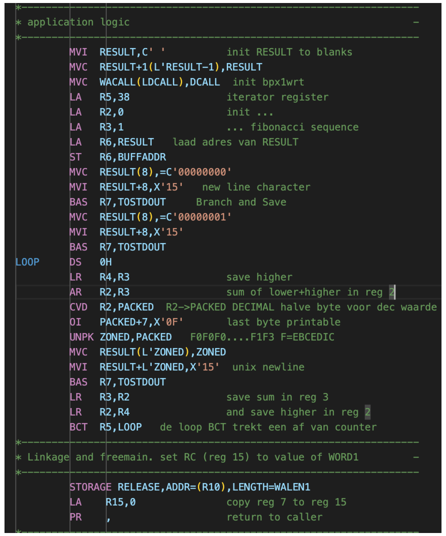
```
         START ,
         YREGS ,                   register equates, syslib SYS1.MACLIB
FIBONACI CSECT ,
FIBONACI AMODE 31
FIBONACI RMODE ANY
*-------------------------------------------------------------------
* fibonacci sequence first 40 results. invoke BPX1WRT to print     -
* to stdout in z/OS Unix                                           -
*-------------------------------------------------------------------
*-------------------------------------------------------------------
* Linkage and getmain                                              -
*-------------------------------------------------------------------
         BAKR  R14,0               use linkage stack conventie
         LR    R12,R15             r15 contains CSECT entry point addr
         USING FIBONACI,R12        CSECT base register
STOR1    STORAGE OBTAIN,LENGTH=WALEN1  get dynamic storage
         LR    R10,R1              LOAD ADDRESS OF STORAGE
         USING WAREA1,R10          BASE FOR DSECT
         MVC   SAVEA1+4(4),=C'F1SA' linkage stack convention
         LAE   R13,SAVEA1          ADDRESS OF OUR SA IN R13
```

The first part of the code example demonstrates basic initialisation to have the program being identified correctly. It determines the start point for entry in register addresses and the storage pool to use. This section establishes “standard linkage” - a long-standing convention for how one program (the Shell command line, for example) can hand control over to another program, and for that program to be able to return control when it finishes back to the right instruction in the caller.

Much more detail available at https://www.ibm.com/docs/en/zos/2.4.0?topic=guide-linkage-conventions

ASM1|230329-1718

3 .. THE BODY …



```
*-----------------------------------------------------------------
* application logic                                              -
*-----------------------------------------------------------------
         MVI  RESULT,C' '           init RESULT to blanks
         MVC  RESULT+1(L'RESULT-1),RESULT
         MVC  WACALL(LDCALL),DCALL  init bpx1wrt
         LA   R5,38                 iterator register
         LA   R2,0                  init ...
         LA   R3,1                  ... fibonacci sequence
         LA   R6,RESULT   laad adres van RESULT
         ST   R6,BUFFADDR
         MVC  RESULT(8),=C'00000000'
         MVI  RESULT+8,X'15'   new line character
         BAS  R7,TOSTDOUT     Branch and Save
         MVC  RESULT(8),=C'00000001'
         MVI  RESULT+8,X'15'
         BAS  R7,TOSTDOUT
LOOP     DS   0H
         LR   R4,R3                 save higher
         AR   R2,R3                 sum of lower+higher in reg 2
         CVD  R2,PACKED  R2->PACKED DECIMAL halve byte voor dec waarde
         OI   PACKED+7,X'0F'        last byte printable
         UNPK ZONED,PACKED   F0F0F0....F1F3 F=EBCEDIC
         MVC  RESULT(L'ZONED),ZONED
         MVI  RESULT+L'ZONED,X'15'  unix newline
         BAS  R7,TOSTDOUT
         LR   R3,R2                 save sum in reg 3
         LR   R2,R4                 and save higher in reg 2
         BCT  R5,LOOP   de loop BCT trekt een af van counter
*-----------------------------------------------------------------
* Linkage and freemain. set RC (reg 15) to value of WORD1        -
*-----------------------------------------------------------------
         STORAGE RELEASE,ADDR=(R10),LENGTH=WALEN1
         LA    R15,0                copy reg 7 to reg 15
         PR    ,                    return to caller
```

This part of the code is the actual application logic.

The first (purple) columns are the instructions which will be executed.

The zSystems CPU architecture uses a wide variety of instructions which can be found in the document called "Principles of Operations"

In this code you can see some basic instructions which will "move", "load" or "store" values in register locations.

- **MVI** shows that a character value of " " (blank) is stored in the predefined register start location address (RESULT).
- the next statement **MVC** moves a numeric value of "+1" to a new location
- the next statement prepares an area of memory that will be used to call a program or service function

ASM1|230329-1718

Within the loop (from **LOOP** label to the **BCT** statement) you can see the calculations used to calculate the next Fibonacci number.

This part is being looped until it has executed 38 times - you can see that register 5 (R5) was earlier initialised with the value **38**.

The **BCT** instruction subtracts 1 from the current R5 value, and if the result is not 0, the program branches to the labeled location (**LOOP**)

The first 2 values (0 and 1) for the Fibonacci sequence calculations were given as base starting points in R2 and R3.

About half of the instructions in the loop are involved with formatting the numbers into displayable characters, and placing these numbers into the RESULT; these rest perform the actual calculations.

The **BAS** instruction is what causes the program to call the subroutine that prints the current RESULT value to *stdout*.

Before the loop ends, the current numeric values for the last number added, and the current total, are moved into the base numbers for the next iteration.

4 … AND THE TAIL

```
*----------------------------------------------------------------------
* subroutine                                                          -
*----------------------------------------------------------------------
TOSTDOUT DS    0H
         CALL  BPX1WRT,(FILEDESC,                                       X
               BUFFADDR,                                                X
               ALET,                                                    X
               WRITECNT,                                                X
               WARETVAL,                                                X
               WARC,                                                    X
               WARSN),MF=(E,WACALL)
         BR    R7
*----------------------------------------------------------------------
* constants and literal pool                                          -
*----------------------------------------------------------------------
DCALL   CALL    ,(0,0,0,0,0,0,0),MF=L
LDCALL   EQU    *-DCALL
ALET     DC     F'0'
FILEDESC DC     F'1'                stdout
WRITECNT DC     F'9'
         LTORG ,                    create literal pool
*
WAREA1   DSECT        convention
SAVEA1   DS    18F   beschrijft gealloceerde mem convention in 31 AMODE
PACKED   DS    CL8   Moet 8 bytes lang zijn
ZONED    DS    CL8
RESULT   DS    CL32    characterLength 32 bytes
WACALL   DS    CL(LDCALL)
         DS    0D
BUFFADDR DS    F
WARETVAL DS    F
WARC     DS    F
WARSN    DS    F
WALEN1   EQU   *-SAVEA1    *current location counter offset vanaf DSECT
         END   FIBONACI
```

In the last part of the code you will find a subroutine to produce the output using a service program and a defined storage area for keep several register values, and any other working variables needed by the program.

This service program **BPX1WRT** is being used to display the output to the standard output file (the “1” indicates the file is “stdout”) since the Assembler language does not have a direct instruction for displaying values like most other programming languages.

ASM1|230329-1718

5 MAKE THE NUMBERS

Navigate using the USS section of VSCode to /z/public/assembler/ and find the example source file:

- **“fibonacci.s”**

Use VSCode to copy this to a subdirectory of your home directory called **“assembly”**.

The first thing you need to do is compile the source code fibonacci.s to a binary file.

Use the terminal function from VSCode to make a SSH connection to the IBM Z Xplore system.

Navigate to the $HOME/assembly directory

Use the `ls` command to assure yourself that you are in the correct folder and the source file is displayed.

Enter the following command to compile the source (for this type of code source, compilation is also known as “assembling”)

```
as -o fibonacci.o fibonacci.s
```

as is the command to “assemble source”, where *fibonacci.s* is the source file and *fibonacci.o* the binary output file (the “object” file).

The next step is to create an executable file where the object file is linked with required libraries (and any other required object modules).

Execute the command

```
ld -o fibonacci fibonacci.o
```

ld is the linker command (“link” was already taken for creating directory links to files) where **fibonacci** is the executable output filename and the *fibonacci.o* file is the input object to be linked to system libraries/services for processing.

Assuming no errors from the linkage process, execute the program by simply typing `./fibonacci`

The first 40 numbers of the Fibonacci sequence should be displayed on your terminal screen.

```
00000000
00000001
00000002
00000003
00000005
[ ... ]
09227465
14930352
24157817
39088169
63245986
```

6 WRAP IT UP AND CLAIM THE POINTS

You made a command from Assembler code!

Let’s get you some credit for that; the usual drill applies:

Submit **CHKASM1** using the Shell command-line:

```
tsocmd submit "'ZXP.PUBLIC.JCL(CHKASM1)'"
```

Nice job - let's recap	Next up …
A simple exercise to get you building and running Assembler code. You saw the steps involved • compile the source to get and "object" file • link the object with any other needed objects and/or libraries • executable the resulting program	The next Assembler module will go into more detail about what is going on with the instructions, and take you inside a running program

ASM1|230329-1718