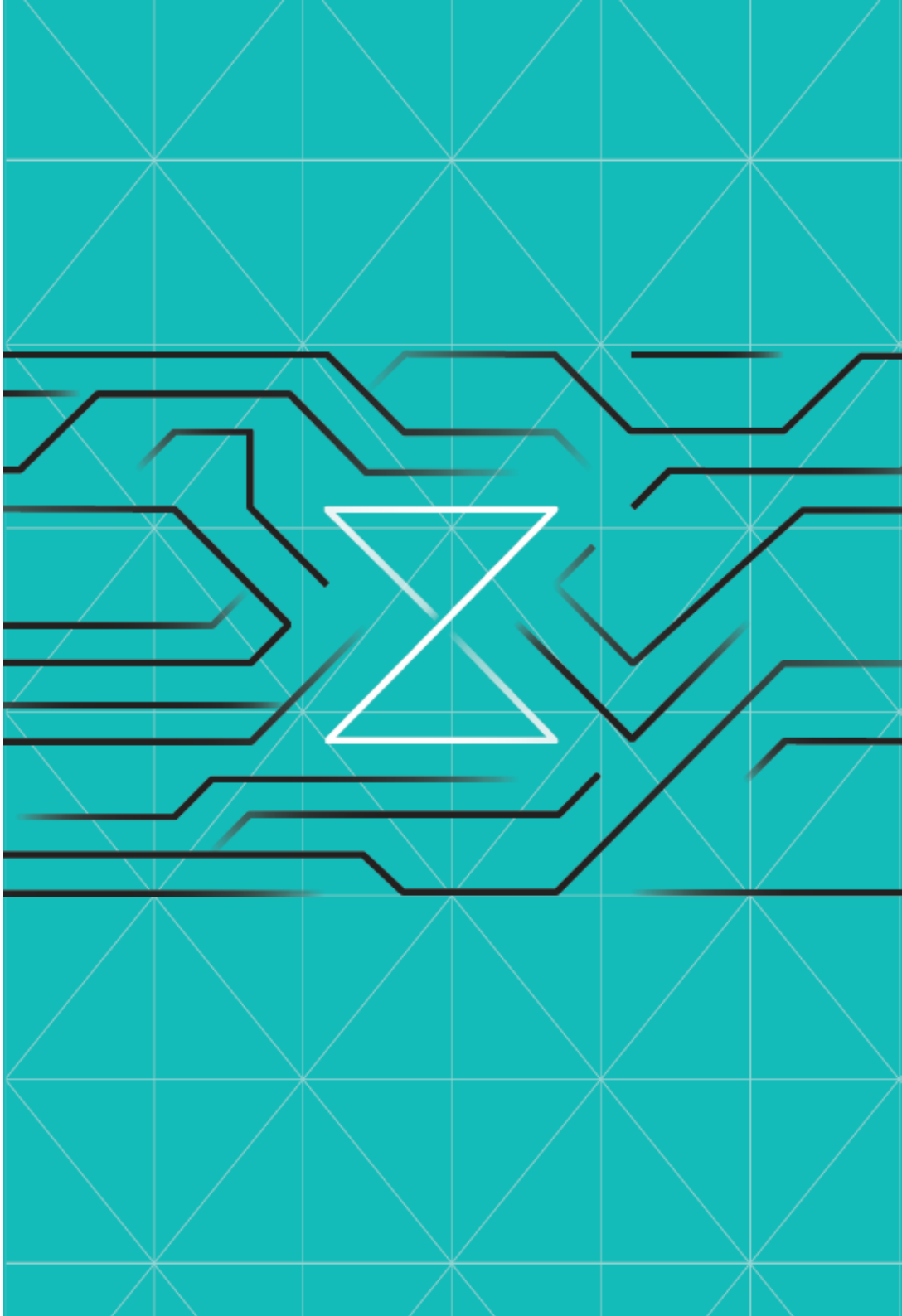
Advanced

240321-2022

ASM1

My first Assembler program

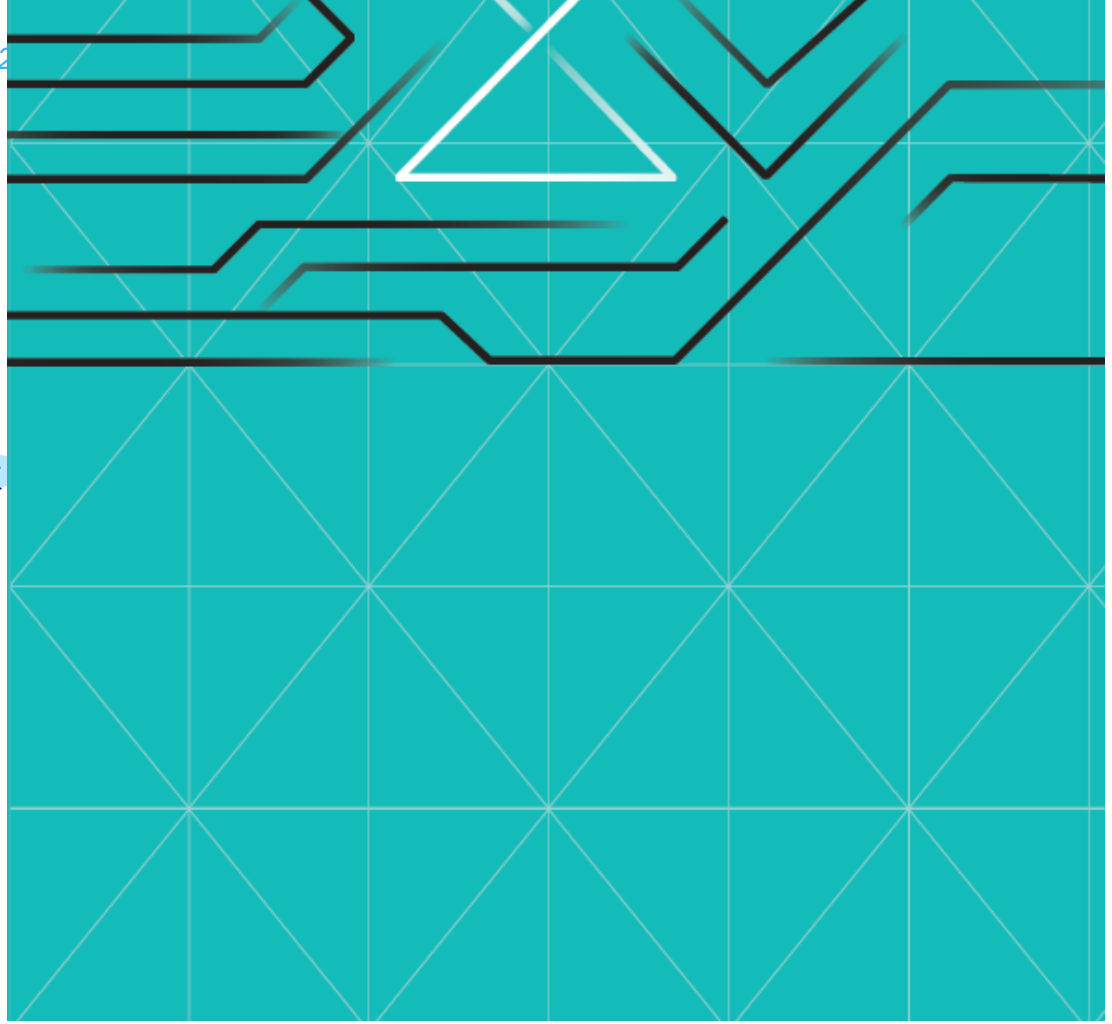
Automated Translation by Watson Language Translator



Z Xplore

ASM1

Meu primeiro programa Assembler



UMA INTRODUÇÃO AO ASSEMBLER PROGRAMMING

Um exemplo simples para mostrar que a construção e execução de programas Assembler é muito semelhante à maneira como os programas COBOL, PL/1 e GO são produzidos.

O Desafio

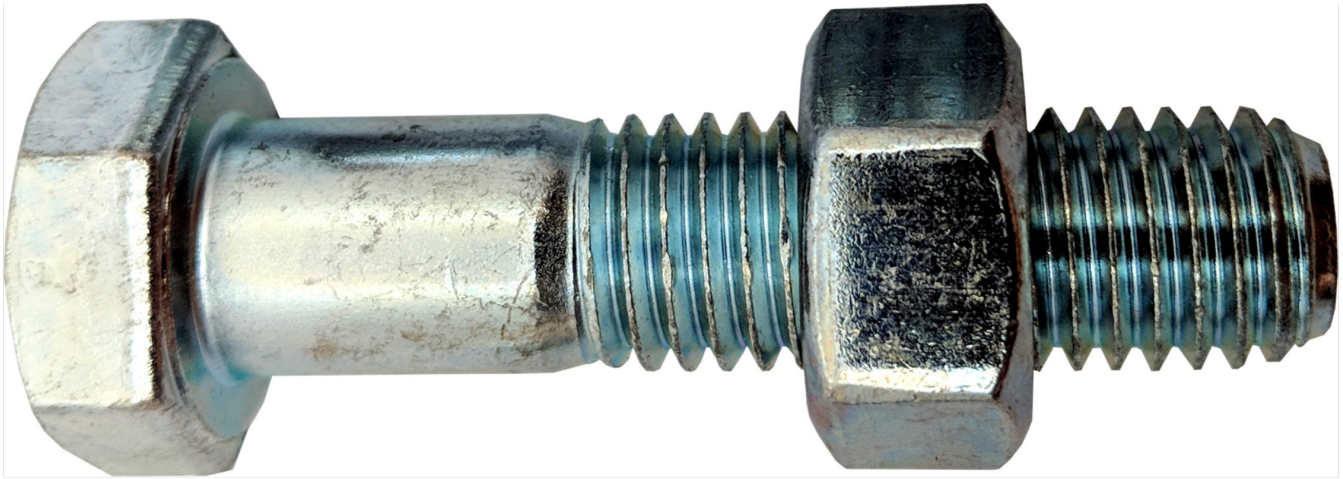
Este desafio lhe dará uma introdução à programação do Assembler. Ele explicará o básico de como o Código do Assembler é compilado e executado.

Investimento

Etapas	Duração
5	20 minutos

ASM1|240321-2022

1 UMA INTRODUÇÃO À PROGRAMAÇÃO DO ASSEMBLER



ALGUNS ANTECEDENTES

Um programa assembler relativamente simples será usado e explicado.

Cada arquitetura de computador tem instruções de máquina exclusivas da arquitetura. Todas as linguagens de computador suportadas pela arquitetura devem ser traduzidas nas instruções exclusivas da máquina da arquitetura do computador subjacente.

Cada arquitetura de computador tem uma linguagem de montagem que inclui "mnemônica" que são montadas em instruções de máquina compreendidas pelo computador.

Compiladores e intérpretes traduzem linguagens de computador suportadas nas instruções exclusivas da máquina compreendidas pelo computador de hospedagem.

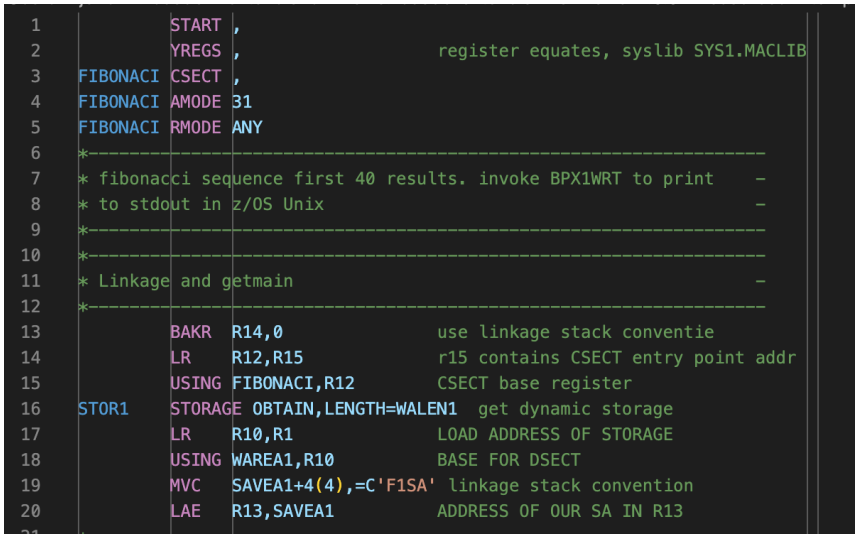
A memória de processamento de computador é usada para carregar e armazenar as instruções e dados da máquina para processamento. O sistema operacional mantém o controle de locais de memória de processamento usando endereços onde parte da memória está livre, parte da memória tem instruções da máquina e parte da memória tem dados.

Linguagens de nível superior, como C/C + +, Java, COBOL, etc., foram criadas para facilitar a programação do computador, ocultando a complexidade das instruções da máquina subjacente, memória endereçável e registros.

Os registros estão no topo da hierarquia de memória e fornecem a maneira mais rápida de acessar os dados.

2 A CABEÇA ...

Abaixo você encontrará o código de exemplo que você usará durante este desafio. O código é construído para mostrar os primeiros 40 números do Sequência de Fibonnaci. usando um modelo de cálculo fácil



```
1          START ,
2          YREGS ,                  register equates, syslib SYS1.MACLIB
3   FIBONACI CSECT ,
4   FIBONACI AMODE 31
5   FIBONACI RMODE ANY
6   *-------------------------------------------------------------------
7   * fibonacci sequence first 40 results. invoke BPX1WRT to print    -
8   * to stdout in z/OS Unix                                          -
9   *-------------------------------------------------------------------
10  *-------------------------------------------------------------------
11  * Linkage and getmain                                             -
12  *-------------------------------------------------------------------
13         BAKR  R14,0              use linkage stack conventie
14         LR    R12,R15            r15 contains CSECT entry point addr
15         USING FIBONACI,R12       CSECT base register
16  STOR1  STORAGE OBTAIN,LENGTH=WALEN1  get dynamic storage
17         LR    R10,R1             LOAD ADDRESS OF STORAGE
18         USING WAREA1,R10         BASE FOR DSECT
19         MVC   SAVEA1+4(4),=C'F1SA' linkage stack convention
20         LAE   R13,SAVEA1         ADDRESS OF OUR SA IN R13
```

A primeira parte do exemplo de código demonstra a inicialização básica para que o programa seja identificado corretamente Ele determina o ponto inicial para entrada em endereços de registro e o conjunto de armazenamentos a ser usado. Esta seção estabelece "ligação padrão"-uma convenção de longa data para como um programa (a linha de comandos Shell, por exemplo) pode transferir o controle para outro programa, e para que esse programa possa retornar o controle quando ele terminar de volta para o a instrução correta no responsável pela chamada.

Muito mais detalhes disponíveis em https://www.ibm.com/docs/en/zos/2.4.0?topic=guide-linkage-conventions

3 .. O CORPO ...

```
*-------------------------------------------------------------------*
* 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
```

Essa parte do código é a lógica de aplicativo real

As primeiras (roxas) colunas são as instruções que serão executadas.

A arquitetura de CPU do zSystems usa uma ampla variedade de instruções que podem ser encontradas no documento chamado "Princípios de Operações"

Neste código você pode ver algumas instruções básicas que irão "mover", "carregar" ou "armazenar" valores em locais de registro.

- **MVI** mostra que um valor de caractere de "" (em branco) é armazenado no endereço do local inicial do registro predefinido (RESULT).
- a próxima instrução **MVC** move um valor numérico de "+ 1" para um novo local
- a próxima instrução prepara uma área de memória que será usada para chamar um programa ou função de serviço

Dentro do loop (de **LOOP** rótulo para o **BCT** você pode ver os cálculos usados para calcular o próximo número de Fibonacci.

Esta parte está sendo looped até que tenha executado 38 vezes-você pode ver que o registro 5 (R5) foi inicializado anteriormente com o valor **38** .

O **BCT** subtrai 1 do valor R5 atual e, se o resultado não for 0, o programa ramifica para o local rotulado (**LOOP**)

Os primeiros 2 valores (0 e 1) para os cálculos de sequência de Fibonacci foram dados como pontos de partida base em R2 e R3.

Cerca de metade das instruções no loop estão envolvidas com a formatação dos números em caracteres exibíveis e a colocação desses números no RESULTADO; esses números restantes realizam os cálculos reais.

O **BAS** instrução é o que faz com que o programa chame a sub-rotina que imprime o valor RESULT atual para *stdout* .

Antes de o loop terminar, os valores numéricos atuais para o último número incluído e o total atual são movidos para os números base para a próxima iteração.

4 ... E A CAUDA

```
*-------------------------------------------------------------------*
* 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
```

Na última parte do código, você encontrará uma sub-rotina para produzir a saída usando um programa de serviços e uma área de armazenamento definida para manter vários valores de registro e quaisquer outras variáveis de trabalho necessárias ao programa.

Este programa de serviço **BPX1WRT** está sendo usado para exibir a saída para o arquivo de saída padrão (o "1" indica que o arquivo é "stdout") uma vez que a linguagem Assembler não tem uma instrução direta para exibir valores como a maioria das outras linguagens de programação.

5 FAÇA OS NÚMEROS

Navegue usando a seção USS do VSCode para /z/public/assembler/ e localize o arquivo de origem de exemplo:

- **"fibonacci.s"**

Use o VSCode para copiar isso para um subdiretório de seu diretório inicial chamado **"montagem"** .

A primeira coisa que você precisa fazer é compilar o código fonte fibonacci.s para um arquivo binário.

Use a função de terminal do VSCode para criar uma conexão SSH com o sistema IBM Z Xplore.

Navegue para o diretório $HOME/assembly

Use o `ls` para assegurar que você esteja na pasta correta e que o arquivo de origem seja exibido.

Insira o comando a seguir para compilar a origem (para esse tipo de origem de código, a compilação também é conhecida como "montagem")

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

como é o comando para "montar origem", em que *fibonacci.s* é o arquivo de origem e *fibonacci.o* o arquivo de saída binário (o arquivo "object").

A próxima etapa é criar um arquivo executável no qual o arquivo de objeto está vinculado com as bibliotecas necessárias (e quaisquer outros módulos de objeto necessários)

Execute o comando.

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

ld é o comando do vinculador ("link" já foi usado para criar links de diretório para arquivos) em que **fibonacci** é o nome do arquivo de saída executável e o *fibonacci.o* arquivo é o objeto de entrada a ser vinculado a bibliotecas / serviços do sistema para processamento

Supondo que não haja erros do processo de ligação, execute o programa simplesmente digitando `./fibonacci`

Os primeiros 40 números da sequência Fibonacci devem ser exibidos na tela do terminal.

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

6 EMBRULHE-O E REIVINDIQUE OS PONTOS

Você fez um comando a partir do código Assembler!

Vamos dar-lhe algum crédito por isso; o exercício habitual aplica-se:

Enviar **CHKASM1** usando a linha de comando Shell:

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

Bom trabalho-vamos recapitular	Em seguida ...
Um exercício simples para construir e executar o código do Assembler. Você viu os passos envolvidos • compilar a origem para obter e o arquivo "object" • vincular o objeto a quaisquer outros objetos e / ou bibliotecas necessários • executar o programa resultante	O próximo módulo Assembler irá entrar em mais detalhes sobre o que está acontecendo com as instruções, e levá-lo dentro de um programa em execução