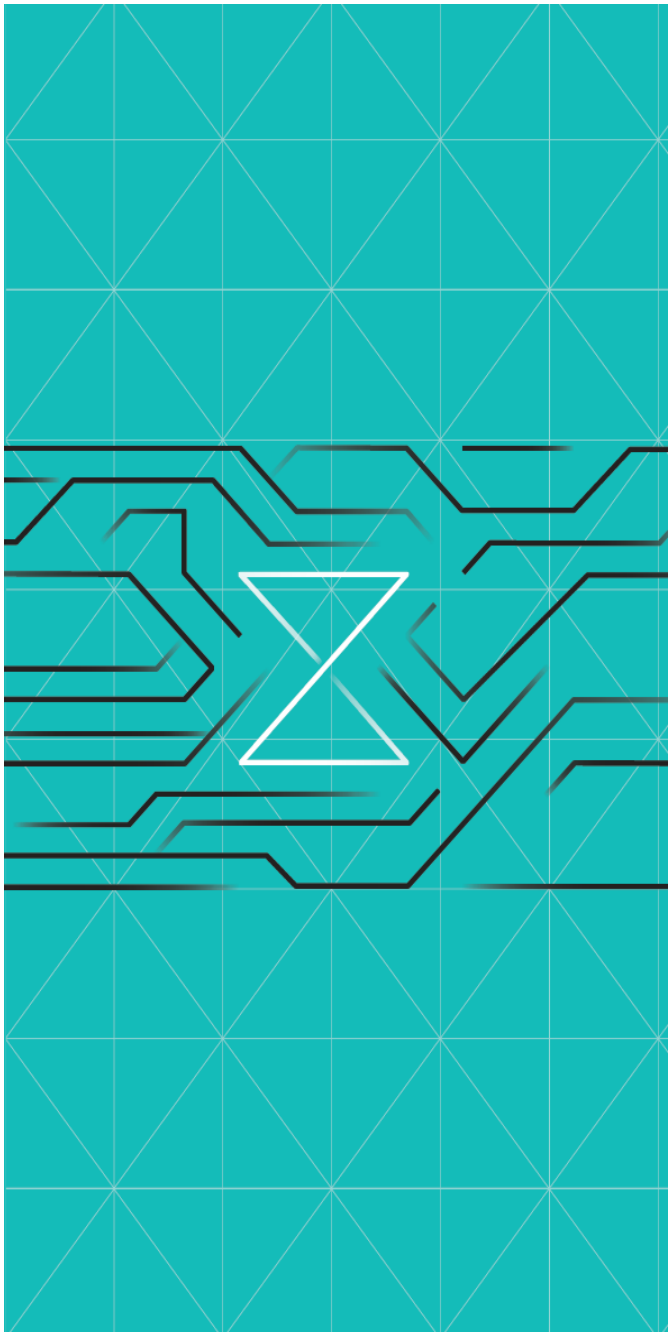


Avançado

250123-1547

ASM1

Meu primeiro programa em Assembler

Tradução automatizada usando o serviço IBM Globalization

250123-1547

AN INTRODUCTION TO ASSEMBLER PROGRAMMING

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

O Desafio

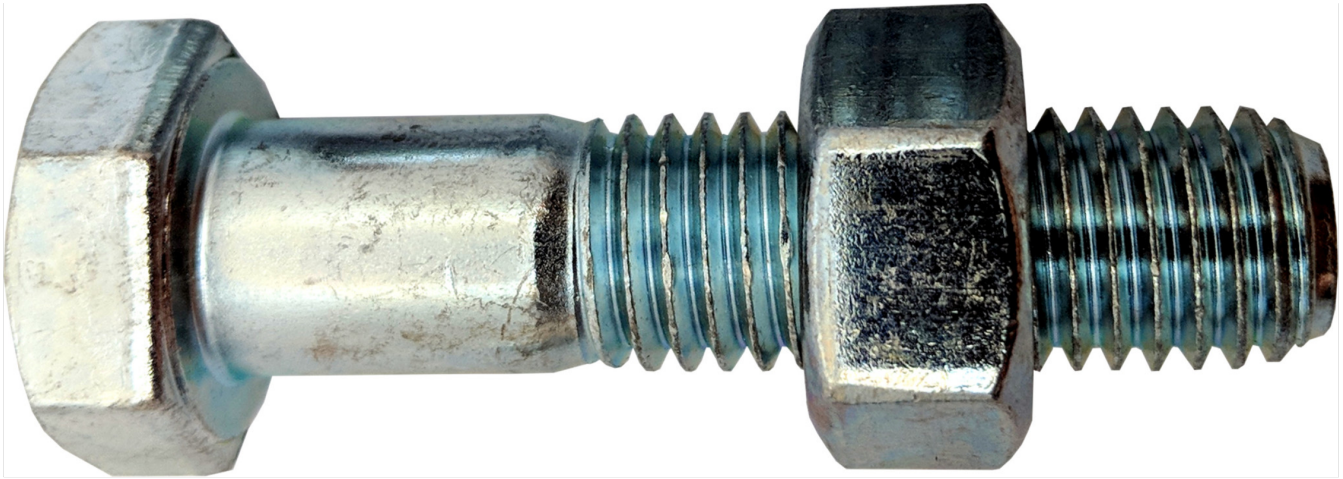
Este desafio lhe dará uma introdução à programação em Assembler. Ele explicará os conceitos básicos de como o código Assembler é compilado e executado.

Investimento

Etapas	Duração
5	20 minutos

ASM1250123-1547

1 AN INTRODUCTION TO ASSEMBLER PROGRAMMING



ALGUNS ANTECEDENTES

Um programa assembler relativamente simples será usado e explicado.

Cada arquitetura de computador tem instruções de máquina exclusivas para essa arquitetura. Todas as linguagens de computador compatíveis com a arquitetura devem ser traduzidas para as instruções de máquina exclusivas da arquitetura de computador subjacente.

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

Os compiladores e intérpretes traduzem as linguagens de computador suportadas em instruções de máquina exclusivas compreendidas pelo computador host.

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

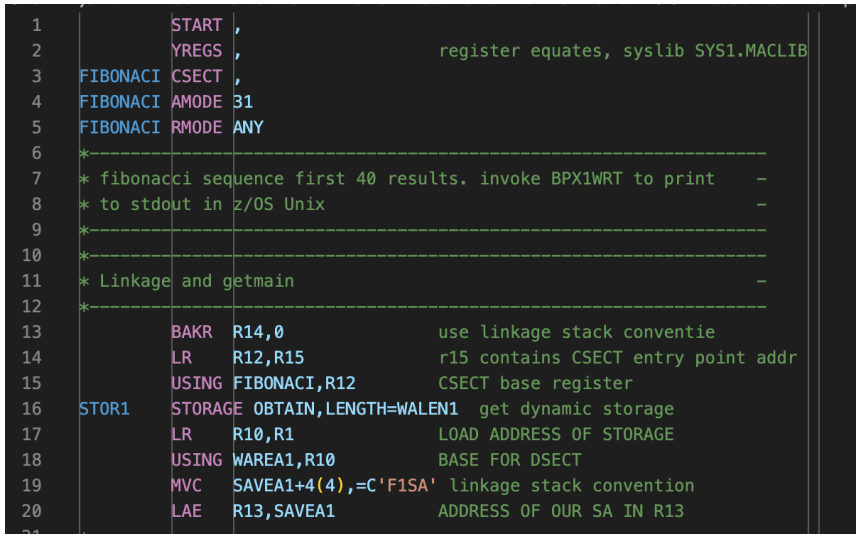
As linguagens de nível superior, como C/C++, Java, COBOL etc., foram criadas para facilitar a programação do computador, ocultando a complexidade das instruções subjacentes da máquina, da memória endereçável e dos registros.

Os registradores estão no topo da hierarquia da memória e oferecem a maneira mais rápida de acessar dados.

O Assembler foi e ainda é usado para permitir maior controle sobre o hardware por meio de manipulação direta. A linguagem de montagem geralmente é mais precisa e eficaz do que o código de alto nível, que não acessa diretamente as instruções de montagem da máquina. A manipulação direta do hardware é útil em situações de alta segurança e em programas em tempo real.

2 THE HEAD ...

Abaixo, você encontrará o código de exemplo que usará durante este desafio. O código foi criado para mostrar os primeiros 38 números da sequência de Fibonacci usando um modelo de cálculo fácil.

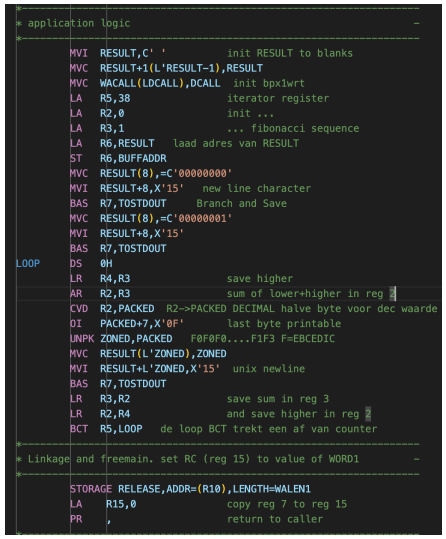


```
1         START ,
2         YREGS ,                  register equates, syslib SYS1.MACLIB
3   FIBONACI CSECT ,
4   FIBONACI AMODE 31
5   FIBONACI RMODE ANY
6   *-------------------------------------------------------------------
7   * fibonacci sequence first 40 results. invoke BPX1WRT to print    -
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 de partida para a entrada nos endereços de registro e o pool de armazenamento a ser usado. Esta seção estabelece a "vinculação padrão", uma convenção de longa data sobre como um programa (a linha de comando do Shell, por exemplo) pode passar o controle para outro programa e como esse programa pode devolver o controle, ao terminar, para a instrução correta no chamador.

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

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

Essa parte do código é a lógica real do aplicativo.

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

A arquitetura da CPU zSystems usa uma ampla variedade de instruções que podem ser encontradas no documento chamado "Principles of Operations" (Princípios de operações)

Nesse código, você pode ver algumas instruções básicas que "movem", "carregam" ou "armazenam" valores em locais de registro.

- **O MVI** mostra que um valor de caractere " " (em branco) é armazenado no endereço do local de início 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 uma função de serviço

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

Essa parte está sendo repetida até ser executada 38 vezes - você pode ver que o registro 5 (R5) foi inicializado anteriormente com o valor **38**.

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

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

Cerca de metade das instruções no loop está envolvida na formatação dos números em caracteres exibíveis e na colocação desses números no RESULT; o restante realiza os cálculos reais.

A instrução **BAS** é o que faz com que o programa chame a sub-rotina que imprime o valor atual do RESULT no *stdout*.

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

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  ,(0,0,0,0,0,0,0,0),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 serviço e uma área de armazenamento definida para manter vários valores de registro e quaisquer outras variáveis de trabalho necessárias ao programa.

Esse programa de serviço **BPX1WRT** está sendo usado para exibir a saída no arquivo de saída padrão (o "1" indica que o arquivo é "stdout"), pois a linguagem Assembler não tem uma instrução direta para exibir valores como a maioria das outras linguagens de programação.

5 MAKE THE NUMBERS

Navegue usando a seção USS do VSCode até /z/public/assembler/ e localize o arquivo de código-fonte de exemplo:

- **“fibonacci.s”**

Use o VSCode para copiá-lo em um subdiretório do seu diretório pessoal chamado **"assembly"** (crie o diretório, se ainda não tiver um).

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

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

Navegue até o diretório $HOME/assembly

Use o comando `ls` para garantir que você está na pasta correta e que o arquivo de origem é exibido.

Digite o seguinte comando para compilar a fonte (para esse tipo de fonte de código, a compilação também é conhecida como "montagem")

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

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

A próxima etapa é criar um arquivo executável no qual o arquivo de objeto é vinculado às bibliotecas necessárias (e a quaisquer outros módulos de objeto necessários).

Executar 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 arquivo *fibonacci.o* é o objeto de entrada a ser vinculado às bibliotecas/serviços do sistema para processamento.

Supondo que não haja erros no processo de vinculação, execute o programa simplesmente digitando `./fibonacci`

Os primeiros 38 números da sequência de Fibonacci devem ser exibidos na tela do terminal.

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

6 WRAP IT UP AND CLAIM THE POINTS

Você criou um comando a partir do código Assembler!

Vamos lhe dar algum crédito por isso; o procedimento usual se aplica:

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

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

ASM1250123-1547

Bom trabalho - vamos recapitular	A seguir .
Um exercício simples para que você crie e execute código Assembler. Você viu as etapas envolvidas • compilar o código-fonte para obter um arquivo "objeto" • vincular o objeto a quaisquer outros objetos e/ou bibliotecas necessários • executável o programa resultante	O próximo módulo de Assembler entrará em mais detalhes sobre o que está acontecendo com as instruções e o levará para dentro de um programa em execução