eBook Gratuit

APPRENEZ

Assembly Language

eBook gratuit non affilié créé à partir des
contributeurs de Stack Overflow.

#assembly

Table des matières

À propos ... 1
Chapitre 1: Démarrer avec le langage d'assemblage ... 2
Remarques ... 2
Examples ... 2
introduction ... 2
Langage machine ... 4
Bonjour tout le monde pour Linux x86_64 (Intel 64 bit) ... 5
Hello World pour OS X (x86_64, gaz de syntaxe Intel) ... 6
Exécution d'un assembly x86 dans Visual Studio 2015 ... 7
Chapitre 2: Contrôle de flux ... 12
Introduction ... 12
Examples ... 12
Trivial IF-THEN-ELSE dans l'assemblage m68k ... 12
POUR ... SUIVANT dans l'assemblage Z80 ... 12
If-instruction dans l'assembly Intel-syntax ... 13
Boucle tant que la condition est vraie dans l'assembly de syntaxe Intel ... 14
Chapitre 3: Exemples Linux elf64 n'utilisant pas la glibc ... 15
Examples ... 15
Interface utilisateur ... 15
Subrtx.asm ... 15
Generic.asm ... 17
Makefile ... 18
Chapitre 4: La pile ... 20
Remarques ... 20
Examples ... 20
Zilog Z80 Stack ... 20
Chapitre 5: Les interruptions ... 22
Remarques ... 22
Examples ... 22
Travailler avec des interruptions sur le Z80: ... 22

Chapitre 6: Registres 23
Remarques 23
Examples 23
Zilog Z80 enregistre 23
x86 registres 25
x64 registres 26
Crédits 28

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: assembly-language

It is an unofficial and free Assembly Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Assembly Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec le langage d'assemblage

Remarques

Assembly est un nom général utilisé pour de nombreuses formes de code machine lisibles par l'homme. Il diffère naturellement beaucoup entre les différents processeurs (Central Processing Unit), mais sur un seul processeur, il peut exister plusieurs dialectes d'Assemblage incompatibles, chacun compilé par un assembleur différent, dans le code machine identique défini par le créateur du processeur.

Si vous souhaitez poser une question sur votre propre problème d'assemblage, indiquez toujours quel matériel vous utilisez et quel assembleur vous utilisez, sinon il sera difficile de répondre à votre question en détail.

L'apprentissage de l'assemblage d'un seul processeur particulier aidera à apprendre les bases de différents processeurs, mais chaque architecture matérielle peut présenter des différences considérables en termes de détails.

Liens:

Assemblage X86 Wikibook

Examples

introduction

Le langage d'assemblage est une forme lisible par l'homme du langage machine ou du code machine qui correspond à la séquence réelle de bits et d'octets sur laquelle la logique du processeur fonctionne. Il est généralement plus facile pour les humains de lire et de programmer des mnémoniques que binaires, octaux ou hexadécimaux. Les humains écrivent donc généralement du code en langage assembleur, puis utilisent un ou plusieurs programmes pour le convertir en langage machine.

EXEMPLE:

```
mov eax, 4
cmp eax, 5
je point
```

Un assembleur est un programme qui lit le programme en langage assembleur, l’analyse et produit le langage machine correspondant. Il est important de comprendre que contrairement à un langage comme C ++ qui est un langage unique défini dans un document standard, il existe de nombreux langages d'assemblage différents. Chaque architecture de processeur, ARM, MIPS, x86, etc. a un code machine différent et donc un langage d'assemblage différent. De plus, il existe

parfois plusieurs langages d'assemblage différents pour la même architecture de processeur. En particulier, la famille de processeurs x86 a deux formats populaires, souvent appelés **syntaxe de gaz** (le `gas` est le nom de l'exécutable pour GNU Assembler) et la **syntaxe Intel** (nommée d'après l'auteur de la famille de processeurs x86). Ils sont différents mais équivalents en ce sens que l'on peut généralement écrire n'importe quel programme dans l'une ou l'autre syntaxe.

Généralement, l'inventeur du processeur documente le processeur et son code machine et crée un langage d'assemblage. Il est courant que ce langage d'assemblage soit le seul utilisé, mais contrairement aux auteurs de compilateurs qui tentent de se conformer à un standard de langage, le langage d'assemblage défini par l'inventeur du processeur n'est généralement pas la version utilisée par ceux qui écrivent des assembleurs. .

Il existe deux types généraux de processeurs:

- CISC (ordinateur de jeu d'instructions complexes): possède de nombreuses instructions différentes et souvent complexes en langage machine
- RISC (Reduced Instruction set Computers): en revanche, les instructions sont moins nombreuses et plus simples

Pour un programmeur en langage assembleur, la différence réside dans le fait qu’un processeur CISC peut recevoir de nombreuses instructions à apprendre, mais il existe souvent des instructions adaptées à une tâche particulière, alors que les processeurs RISC ont des instructions moins nombreuses et complexes. pour écrire plus d'instructions pour faire la même chose.

Les autres compilateurs de langages de programmation produisent parfois des assembleurs en premier, qui sont ensuite compilés en code machine en appelant un assembleur. Par exemple, *gcc* utilise son propre assembleur de *gaz* en phase finale de compilation. Le code machine produit est souvent stocké dans des fichiers *objets* , qui peuvent être liés dans le fichier exécutable par le programme de l'éditeur de liens.

Une "chaîne d'outils" complète consiste souvent en un compilateur, un assembleur et un éditeur de liens. On peut alors utiliser cet assembleur et cet éditeur de liens directement pour écrire des programmes en langage assembleur. Dans le monde GNU, le paquet *binutils* contient l'assembleur et l'éditeur de liens et les outils associés; ceux qui sont uniquement intéressés par la programmation en langage d'assemblage n'ont pas besoin de *gcc* ou d'autres packages de compilateurs.

Les petits microcontrôleurs sont souvent programmés uniquement en langage d'assemblage ou dans une combinaison de langage d'assemblage et d'un ou plusieurs langages de niveau supérieur tels que C ou C ++. Ceci est fait car on peut souvent utiliser les aspects particuliers de l' **architecture du jeu d'instructions** pour que de tels dispositifs écrivent un code plus compact et efficace que ce qui serait possible dans un langage de niveau supérieur et ces dispositifs ont souvent une mémoire et des registres limités. De nombreux microprocesseurs sont utilisés dans **des systèmes embarqués** qui sont des périphériques autres que les ordinateurs à usage général qui ont un microprocesseur à l'intérieur. Des exemples de tels systèmes embarqués sont les téléviseurs, les fours à micro-ondes et l'unité de commande du moteur d'une automobile moderne.

Beaucoup de ces dispositifs n’ont ni clavier ni écran, de sorte qu’un programmeur écrit généralement le programme sur un ordinateur à usage général, exécute un **assembleur croisé** (ainsi appelé parce que ce type d’assembleur produit du code pour un type de processeur différent de celui sur lequel il est exécuté).) et / ou un **compilateur croisé** et un **cross-linker** pour produire du code machine.

Il existe de nombreux fournisseurs de tels outils, aussi variés que les processeurs pour lesquels ils produisent du code. Beaucoup de processeurs, mais pas tous, ont également une solution open source comme GNU, sdcc, llvm ou autre.

Langage machine

Le code machine désigne les données, en particulier le format de machine natif, directement traité par la machine, généralement par le processeur appelé *CPU* (Central Processing Unit).

L'architecture informatique commune (architecture *von Neumann*) comprend un processeur à usage général (CPU), une mémoire à usage général - stockant à la fois le programme (ROM / RAM) et les données traitées et les périphériques d'entrée / sortie (périphériques E / S).

Le principal avantage de cette architecture est la simplicité relative et l’universalité de chacun des composants - par rapport aux machines informatiques précédentes (avec un programme câblé dans la construction de la machine), ou des architectures concurrentes (par exemple *l’architecture Harvard* séparant la mémoire du programme de la mémoire). Les données). Le désavantage est un peu moins performant. À long terme, l’universalité permettait une utilisation flexible, qui dépassait généralement le coût des performances.

Comment cela se rapporte-t-il au code machine?

Le programme et les données sont stockés dans ces ordinateurs sous forme de nombres, dans la mémoire. Il n'y a pas de véritable moyen de distinguer le code des données, donc les systèmes d'exploitation et les opérateurs de machine donnent des indications sur le processeur, à quel point de la mémoire commence le programme, après avoir chargé tous les numéros en mémoire. Le CPU lit alors l'instruction (numéro) stockée au point d'entrée et la traite rigoureusement, en lisant séquentiellement les prochains numéros en tant qu'instructions supplémentaires, à moins que le programme lui-même n'indique à la CPU de continuer l'exécution ailleurs.

Par exemple, deux nombres de 8 bits (8 bits regroupés sont égaux à 1 octet, soit un nombre entier non signé compris entre 0 et 255): `60 201` , lorsqu'ils sont exécutés en tant que code sur la Zilog Z80, seront traités comme deux instructions: `INC a` (valeur incrémentée dans le registre `a` par un) et `RET` (retour de la sous-routine, pointant le processeur pour exécuter les instructions de différentes parties de la mémoire).

Pour définir ce programme, un humain peut entrer ces nombres par un éditeur de mémoire / fichier, par exemple dans hex-editor sous la forme de deux octets: `3C C9` (nombres décimaux 60 et 201 écrits en codage base 16). Ce serait la *programmation en **code machine*** .

Pour faciliter la tâche de la programmation CPU pour les humains, un programme **Assembler a** été créé, capable de lire un fichier texte contenant quelque chose comme:

```
subroutineIncrementA:
    INC   a
    RET

dataValueDefinedInAssemblerSource:
    DB    60          ; define byte with value 60 right after the ret
```

sortie de la séquence de nombres hexadécimaux d'octets `3C C9 3C` , entourée de nombres supplémentaires facultatifs spécifiques à la plate-forme cible: indiquer quelle partie de ce code est exécutable, où est le point d'entrée du programme (la première instruction), quelles parties sont codées données (non exécutables), etc.

Remarquez comment le programmeur a spécifié le dernier octet avec la valeur 60 comme "données", mais du point de vue de la CPU, il ne diffère en rien de `INC a` octet. Le programme d'exécution doit naviguer correctement sur les octets préparés en tant qu'instructions et traiter les octets de données uniquement en tant que données pour les instructions.

Une telle sortie est généralement stockée dans un fichier sur un périphérique de stockage, chargée ultérieurement par le système d'exploitation (*système d'exploitation - un code machine déjà exécuté sur l'ordinateur, permettant de manipuler l'ordinateur*) point d'entrée du programme.

Le processeur ne peut traiter et exécuter que du code machine - mais tout contenu de mémoire, même aléatoire, peut être traité en tant que tel, bien que le résultat puisse être aléatoire, allant de " *crash* " détecté O périphériques, ou endommagement des équipements sensibles connectés à l'ordinateur (pas un cas commun pour les ordinateurs domestiques :)).

Le processus similaire est suivi par de nombreux autres langages de programmation de haut niveau, en compilant la **source** (forme de texte lisible par un humain) en chiffres, représentant le code de la machine (instructions natives du CPU) ou code de machine virtuelle spécifique à la langue, qui est ensuite décodé en code machine natif pendant l'exécution par un interpréteur ou une machine virtuelle.

Certains compilateurs utilisent l'assembleur comme étape intermédiaire de la compilation, traduisant tout d'abord le source sous la forme Assembler, puis exécutant l'outil Assembleur pour en extraire le **code machine** final (exemple GCC: exécutez `gcc -S helloworld.c` pour obtenir une version assembleur du programme C) `helloworld.c`).

Bonjour tout le monde pour Linux x86_64 (Intel 64 bit)

```
section .data
    msg db "Hello world!",10      ; 10 is the ASCII code for a new line (LF)

section .text
    global _start

_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, 13
    syscall
```

```
    mov rax, 60
    mov rdi, 0
    syscall
```

Si vous voulez exécuter ce programme, vous devez d'abord installer Netwide Assembler , `nasm` , car ce code utilise sa syntaxe. Utilisez ensuite les commandes suivantes (en supposant que le code se trouve dans le fichier `helloworld.asm`). Ils sont nécessaires pour l'assemblage, la liaison et l'exécution, respectivement.

- `nasm -felf64 helloworld.asm`
- `ld helloworld.o -o helloworld`
- `./helloworld`

Le code utilise `sys_write` syscall de Linux. Ici vous pouvez voir une liste de tous syscalls pour l'architecture x86_64. Lorsque vous tenez également compte des pages de manuel de write et exit , vous pouvez traduire le programme ci-dessus en C qui fait la même chose et est beaucoup plus lisible:

```
#include <unistd.h>

#define STDOUT 1

int main()
{
    write(STDOUT, "Hello world!\n", 13);
    _exit(0);
}
```

Deux commandes sont nécessaires ici pour la compilation et la liaison (première) et pour l'exécution:

- `gcc helloworld_c.c -o helloworld_c` .
- `./helloworld_c`

Hello World pour OS X (x86_64, gaz de syntaxe Intel)

```
.intel_syntax noprefix

.data

.align 16
hello_msg:
    .asciz "Hello, World!"

.text

.global _main
_main:
    push rbp
    mov rbp, rsp

    lea rdi, [rip+hello_msg]
```

```
    call _puts

    xor rax, rax
    leave
    ret
```

Assembler:

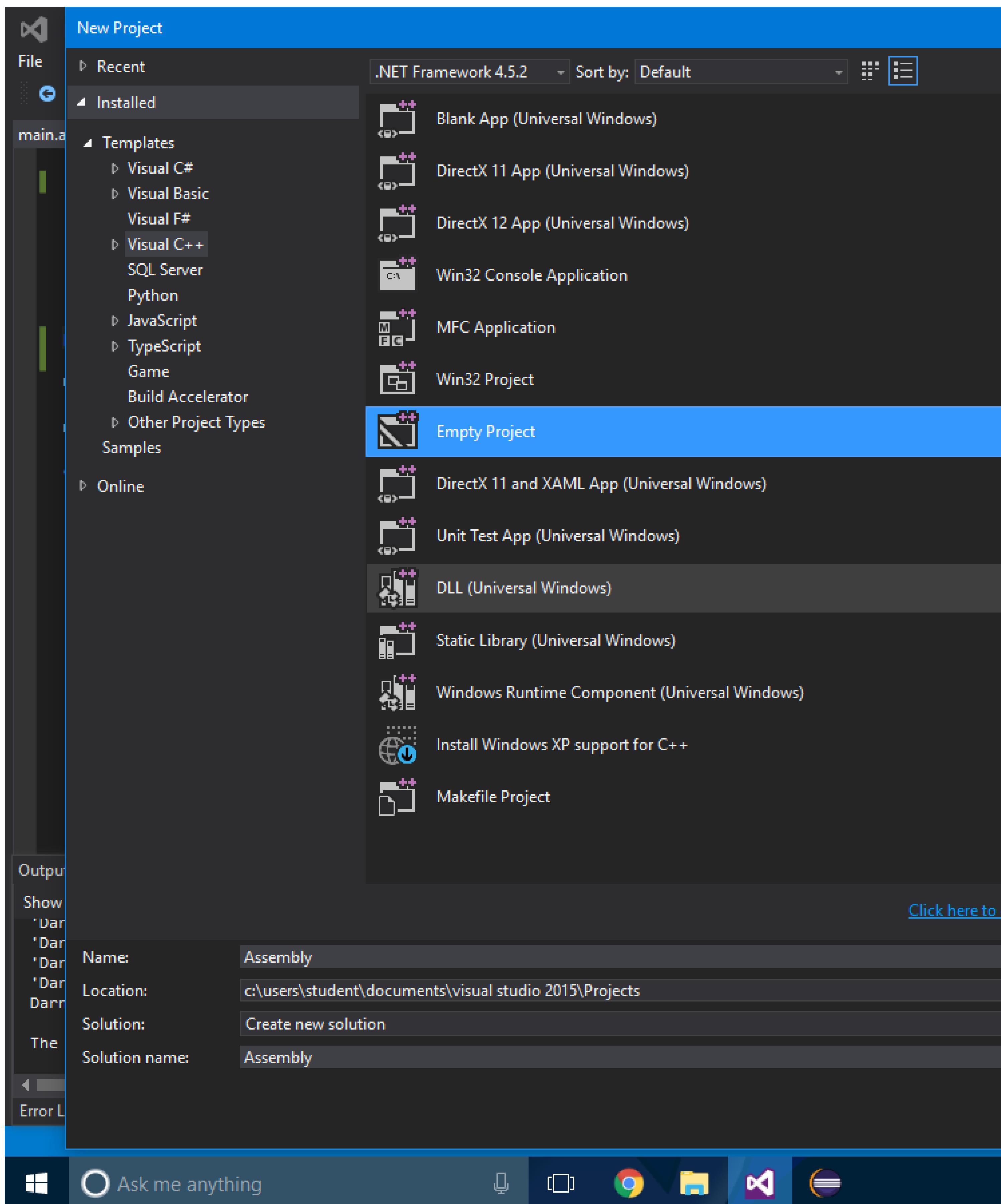
```
clang main.s -o hello
./hello
```

Remarques:

- L'utilisation d'appels système est déconseillée car l'API d'appel système d'OS X n'est pas considérée comme stable. Au lieu de cela, utilisez la bibliothèque C. (Référence à une question de débordement de pile)
- Intel recommande que les structures plus grandes qu'un mot commencent sur une limite de 16 octets. (Référence à la documentation Intel)
- Les données d'ordre sont passées dans les fonctions via les registres: rdi, rsi, rdx, rcx, r8 et r9. (Référence à System V ABI)

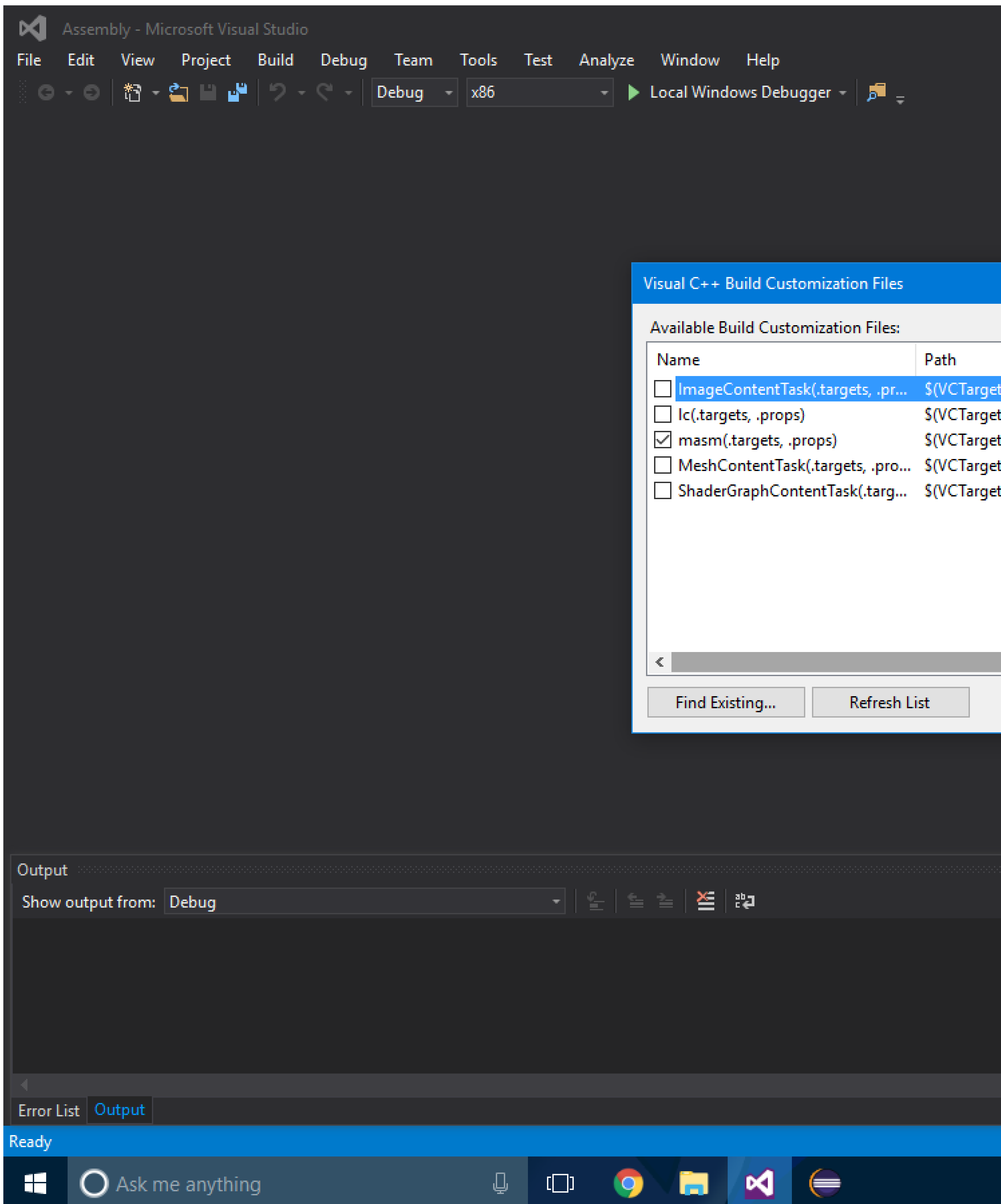
Exécution d'un assembly x86 dans Visual Studio 2015

Étape 1 : Créez un projet vide via **Fichier -> Nouveau projet** .



Étape 2 : Cliquez avec le bouton droit sur la solution du projet et sélectionnez **Build Dependencies-> Build Customizations** .

Étape 3 : Cochez la case *".masm"* .



Étape 4 : Appuyez sur le bouton *"ok"* .

Étape 5 : Créez votre fichier d'assemblage et saisissez-le:

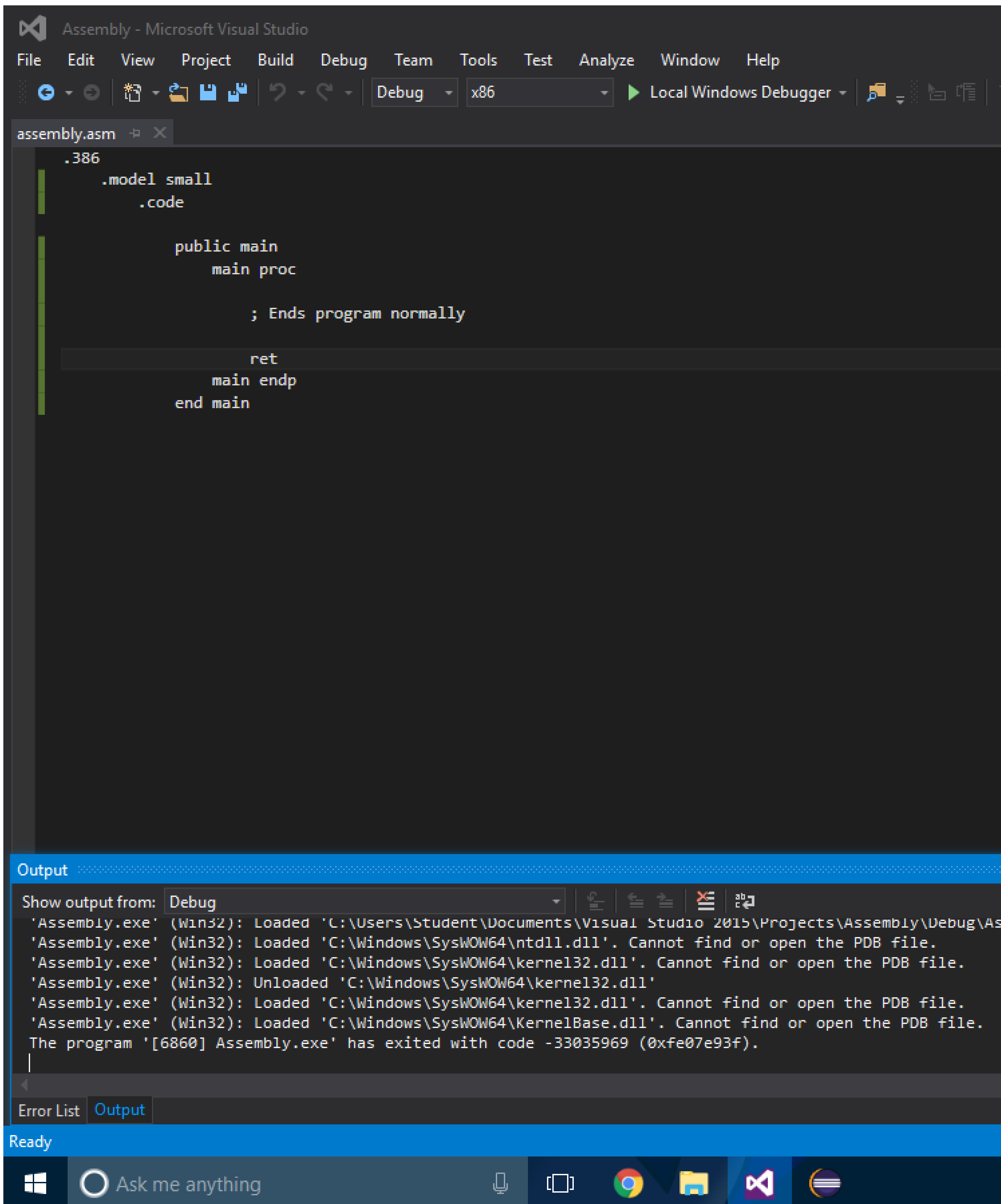
```
.386
    .model small
        .code

            public main
                main proc

                    ; Ends program normally

                    ret
                main endp
            end main
```

Étape 6 : Compiler!



Lire Démarrer avec le langage d'assemblage en ligne:
https://riptutorial.com/fr/assembly/topic/1358/demarrer-avec-le-langage-d-assemblage

Chapitre 2: Contrôle de flux

Introduction

Chaque morceau de logiciel non trivial a besoin de structures de contrôle de flux pour détourner le flux du programme en fonction des conditions. L'assemblage étant le langage de programmation de niveau le plus bas, il ne fournit que des *primitives* pour les structures de contrôle. En règle générale, les opérations de la machine affectent les *indicateurs* de la CPU et les *branches / sauts conditionnels* implémentent le contrôle de flux. Dans l'assemblage, toutes les structures de contrôle de niveau supérieur doivent être construites à partir de telles primitives.

Examples

Trivial IF-THEN-ELSE dans l'assemblage m68k

```
; IF d0 == 10 GO TO ten, ELSE GO TO other
    CMP    #10,d0         ; compare register contents to immediate value 10
                          ; instruction affects the zero flag
    BEQ    ten            ; branch if zero flag set
other:
    ; do whatever needs to be done for d0 != 10
    BRA    afterother     ; unconditionally jump across IF case
ten:
    ; do whatever needs to be done for d0 == 10
afterother:
    ; continue normal common program flow
```

Les instructions qui affectent les drapeaux et les branches conditionnelles (qui peuvent également être basées sur des *combinaisons* spécifiques *de drapeaux*) sont disponibles, dépendent beaucoup du processeur choisi et doivent être recherchées dans les manuels.

POUR ... SUIVANT dans l'assemblage Z80

Le Z80 a une instruction spécifique pour implémenter les comptages de boucle: `DJNZ` pour "décrémenter B enregistrer et sauter sinon zéro". Donc, B est le registre de choix pour implémenter des boucles sur ce processeur. FOR ... NEXT doit être implémenté "en arrière", car le registre est décompté à zéro. D'autres processeurs (comme le 8086, ce processeur utilise le registre CX comme compteur de boucle) peuvent avoir des registres et des instructions de compteur de boucle similaires, d'autres CPU permettent des commandes de boucle avec des registres arbitraires (m68k a une instruction DBRA compatible avec n'importe quel registre de données).

```
; Trivial multiplication (by repeated adding, ignores zero in factors, so
; not recommended for general use)
;
; inputs:     A = Factor 1
;             B = Factor 2
;
```

```
; output:    A = Factor 1 * Factor 2
;
; Pseudo code
; C = A : A = 0 : FOR B = Factor 2 DOWNTO 0 : A = A + C : NEXT B

mul:
     LD    C,A        ; Save Factor 1 in C register
     XOR   A          ; Clear accumulator
mLoop:
     ADD   A,C        ; Add Factor 1 to accumulator
     DJNZ  mLoop      ; Do this Factor 2 times
     RET              ; return to caller
```

If-instruction dans l'assembly Intel-syntax

```
section .data
    msg_eq db 'Equal', 10
    len_eq equ $ - msg_eq

    msg_le db 'Less than', 10
    len_le equ $ - msg_le

    msg_gr db 'Greater than', 10
    len_gr equ $ - msg_gr ; Length of msg_gr
section .text
    global _main ; Make the _main label global for linker
_main:
    cmp 4, 5 ; Compare 4 and 5
    je _equal ; je = jump if equal
    jl _less ; jl = jump if less
    jg _greater ; jg = jump if greater
exit:
    ret ; Return
_equal:
    ; Whatever code here
    mov rax, 0x2000004 ; sys_write, 4 for linux
    mov rdi, 1 ; STDOUT
    mov rsi, msg_eq
    mov rdi, len_eq

    syscall

    jmp exit ; Exit
_less:
    ; Whatever code here
    mov rax, 0x2000004
    mov rdi, 1
    mov rsi, msg_le
    mov rdi, len_le

    syscall

    jmp exit
_greater:
    ; Whatever code here

    mov rax, 0x2000004
    mov rdi, 1
    mov rsi, msg_gr
```

```
    mov rdi, len_gr

    syscall
    jmp exit
```

Boucle tant que la condition est vraie dans l'assembly de syntaxe Intel

```
section .data
    msg db 'Hello, world!', 0xA
    len equ $ - msg
section .text
global _main
_main:
    mov rax, 0 ; This will be the current number
    mov rcx, 10 ; This will be the last number

_loop:
    cmp rax, rcx
    jl .loopbody ; Jump to .loopbody if rax < rcx
    jge _exit ; Jump to _exit if rax ≥ rcx
.loopbody:
    push rax ; Store the rax value for later use

    mov rax, 0x2000004 ; 4 for Linux
    mov rdi, 1 ; STDOUT
    mov rsi, msg
    mov rdx, len

    syscall

    pop rax ; Take it back to rax

    inc rax ; Add 1 to rax. This is required since the loop must have an ending.

    jmp _loop ; Back to loop
_exit:
    ret    ; Return
```

Cela exécutera `.loopbody` aussi longtemps que `rax < rcx` .

Lire Contrôle de flux en ligne: https://riptutorial.com/fr/assembly/topic/8172/controle-de-flux

Chapitre 3: Exemples Linux elf64 n'utilisant pas la glibc

Examples

Interface utilisateur

Je dirais que 80% du traitement des systèmes informatiques modernes ne nécessite pas d'interaction avec l'utilisateur, comme le code du noyau pour Linux, OSX et Windows. Pour ceux qui le font, il y a deux principes fondamentaux qui sont l'interactivité via le clavier (*dispositifs de pointage*) et la console. Cet exemple et d'autres dans ma série sont orientés autour de la console à base de texte (émulation VT100) et du clavier.

En soi, cet exemple est très simple, mais il s’agit d’une composante essentielle vers des algorithmes plus complexes.

Subrtx.asm

```
        STDIN    equ        0
       STDOUT    equ        1

     SYS_READ    equ        0
    SYS_WRITE    equ        1

    global  gets, strlen, print, atoq

            section .text
```

Comme il est destiné exclusivement au clavier, la probabilité d'erreurs est quasi nulle. J'imagine que le plus souvent, le programme sera capable d'envisager la taille de la mémoire tampon pour contourner le dépassement de la mémoire tampon, mais cela n'est pas garanti en raison de l'indirection.

```
; =============================================================================
; Accept canonical input from operator for a maximum of EDX bytes and replace
; terminating CR with NULL.

;       ENTER: RSI = Pointer to input buffer
;              EDX = Maximum number of characters

;       LEAVE: EAX = Number of characters entered
;              R11 = Modified by syscall, all others preserved.

;       FLAGS:  ZF = Null entry, NZ otherwise.
; _____________________________________________________________________________

     gets:  push    rcx
            push    rdi
```

```
            xor     eax, eax            ; RAX = SYS_READ
            mov     edi, eax            ; RDI = STDIN
            syscall

; TODO:    Should probably do some error trapping here, especially for
;           buffer overrun, but I'll see if it becomes an issue over time.

            dec     eax                 ; Bump back to CR
            mov     byte [rsi+rax], 0   ; Replace it with NULL

            pop     rdi
            pop     rcx
            ret
```

Pour commencer, cela visait à contourner le besoin de coder ou de calculer manuellement une longueur de chaîne pour write (2). Puis j'ai décidé d'incorporer un délimiteur, maintenant il peut être utilisé pour rechercher n'importe quel caractère (0 - FF). Cela ouvre la possibilité pour le texte d'emballage de texte par exemple, de sorte que le ***strlen*** d'étiquette est un peu un abus de langage que l' on pourrait penser généralement le résultat va être le nombre de caractère visible.

```
; =============================================================================
; Determine length, including terminating character EOS. Result may include
; VT100 escape sequences.

;        ENTER: RDI = Pointer to ASCII string.
;               RCX   Bits 31 - 08 = Max chars to scan (1 - 1.67e7)
;                            07 - 00 = Terminating character (0 - FF)

;        LEAVE: RAX = Pointer to next string (optional).

;        FLAGS:  ZF = Terminating character found, NZ otherwise (overrun).
; _____________________________________________________________________________

   strlen:  push    rcx                 ; Preserve registers used by proc so
            push    rdi                 ; it's non-destructive except for RAX.

            mov      al, cl             ; Byte to scan for in AL.
            shr     ecx, 8              ; Shift max count into bits 23 - 00

; NOTE: Probably should check direction flag here, but I always set and
;       reset DF in the process that is using it.

            repnz   scasb               ; Scan for AL or until ECX = 0
            mov     rax, rdi            ; Return pointer to EOS + 1

            pop     rdi                 ; Original pointer for proglogue
            jz      $ + 5               ; ZF indicates EOS was found
            mov     rax, rdi            ; RAX = RDI, NULL string
            pop     rcx

            ret
```

L'intention de cette procédure est de simplifier la conception de la boucle dans la procédure d'appel.

```
; =============================================================================
```

```
; Display an ASCIIZ string on console that may have embedded VT100 sequences.

;       ENTER: RDI = Points to string

;       LEAVE: RAX = Number of characters displayed, including EOS
;                  = Error code if SF
;              RDI = Points to byte after EOS.
;              R11 = Modified by syscall all others preserved

;       FLAGS:  ZF = Terminating NULL was not found. NZ otherwise
;               SF = RAX is negated syscall error code.
;______________________________________________________________________________

    print:  push    rsi
            push    rdx
            push    rcx

            mov     ecx, -1 << 8         ; Scan for NULL
            call    strlen
            push    rax                  ; Preserve point to next string
            sub     rax, rdi             ; EAX = End pntr - Start pntr
            jz      .done

     ; size_t = write (int STDOUT, char *, size_t length)

            mov     edx, eax             ; RDX = length
            mov     rsi, rdi             ; RSI = Pointer
            mov     eax, SYS_WRITE
            mov     edi, eax             ; RDI = STDOUT
            syscall
            or      rax, rax             ; Sets SF if syscall error

; NOTE:    This procedure is intended for console, but in the event STDOUT is
;       redirected by some means, EAX may return error code from syscall.

    .done:  pop     rdi                  ; Retrieve pointer to next string.
            pop     rcx
            pop     rdx
            pop     rsi

            ret
```

Enfin, un exemple de la façon dont ces fonctions peuvent être utilisées.

Generic.asm

```
global  _start

    extern  print, gets, atoq

    SYS_EXIT  equ   60
         ESC  equ   27

       BSize  equ   96

            section .rodata
   Prompt:  db  ESC, '[2J'      ; VT100 clear screen
            db  ESC, '[4;11H'   ;   "   Position cursor to line 4 column 11
```

```
            db  'ASCII -> INT64 (binary, octal, hexidecimal, decimal), '
            db  'Packed & Unpacked BCD and floating point conversions'
            db  10, 10, 0, 9, 9, 9, '=> ', 0
            db  10, 9, 'Bye'
            db  ESC, '[0m'      ; VT100 Reset console
            db  10, 10, 0

            section .text
    _start: pop    rdi
            mov    rsi, rsp
            and    rsp, byte 0xf0       ; Align stack on 16 byte boundary.

            call   main
            mov    rdi, rax             ; Copy return code into ARG0

            mov    eax, SYS_EXIT
            syscall

; int main ( int argc, char *args[] )

      main: enter   BSize, 0            ; Input buffer on stack
            mov     edi, Prompt
            call    print
            lea     rsi, [rbp-BSize]    ; Establish pointer to input buffer
            mov     edx, BSize          ; Max size for read(2)

     .Next: push    rdi                 ; Save pointer to "=> "
            call    print
            call    gets
            jz      .done

            call    atoq                ; Convert string pointed to by RSI

            pop     rdi                 ; Restore pointer to prompt
            jmp     .Next

     .done: call    print               ; RDI already points to "Bye"
            xor     eax, eax
            leave
            ret
```

Makefile

```
OBJECTS = Subrtx.o Generic.o

Generic : $(OBJECTS)
    ld -oGeneric $(OBJECTS)
    readelf -WS Generic

Generic.o : Generic.asm
     nasm -g -felf64 Generic.asm

Subrtx.o : Subrtx.asm
    nasm -g -felf64 Subrtx.asm

clean:
    rm -f $(OBJECTS) Generic
```

Lire Exemples Linux elf64 n'utilisant pas la glibc en ligne:
https://riptutorial.com/fr/assembly/topic/7059/exemples-linux-elf64-n-utilisant-pas-la-glibc

Chapitre 4: La pile

Remarques

La pile d'ordinateurs est comme une pile de livres. *PUSH* ajoute l'un au sommet et *POP* prend le plus haut. Comme dans la vraie vie, la pile ne peut pas être sans fin, elle a donc une taille maximale. La pile peut être utilisée pour trier des algorithmes, gérer une plus grande quantité de données ou des valeurs sûres de registres lors d'une autre opération.

Examples

Zilog Z80 Stack

Le registre `sp` est utilisé comme *pointeur de pile* , pointant vers la dernière valeur stockée dans la pile ("top" de la pile). Donc `EX (sp),hl` échangera la valeur de `hl` avec la valeur en haut de la pile.

Contrairement à ce mot « top », la pile grandit dans la mémoire en diminuant la `sp` , et libère (« POP ») en augmentant les valeurs `sp` .

Pour `sp` = `$4844` avec les valeurs `1` , `2` , `3` stockées sur la pile (le `3` étant poussé sur la pile comme dernière valeur, donc en haut), la mémoire ressemblera à ceci:

```
|    address | value bytes | comment (btw, all numbers are in hexadecimal)
| ---------- | ----------- | ---------------------------------
|       4840 | ?? ??       | free stack spaces to be used by next push/call
|       4842 | ?? ??       | or by interrupt call! (don't expect values to stay here)
| sp -> 4844 | 03 00       | 16 bit value "3" on top of stack
|       4846 | 02 00       | 16 bit value "2"
|       4848 | 01 00       | 16 bit value "1"
|       484A | ?? ??       | Other values in stack (up to it's origin)
|       484C | ?? ??       | like for example return address for RET instruction
```

Exemples, comment les instructions fonctionnent avec la pile:

```
    LD   hl,$0506
    EX   (sp),hl           ; $0003 into hl, "06 05" bytes at $4844
    POP  bc                ; like: LD c,(sp); INC sp; LD b,(sp); INC sp
                           ; so bc is now $0506, and sp is $4846
    XOR  a                 ; a = 0, sets zero and parity flags
    PUSH af                ; like: DEC sp; LD (sp),a; DEC sp; LD (sp),f
                           ; so at $4844 is $0044 (44 = z+p flags), sp is $4844
    CALL $8000             ; sp is $4842, with address of next ins at top of stack
                           ; pc = $8000 (jumping to sub-routine)
                           ; after RET will return here, the sp will be $4844 again
    LD   (L1+1),sp         ; stores current sp into LD sp,nn instruction (self modification)
    DEC  sp                ; sp is $4843
L1  LD   sp,$1234          ; restores sp to $4844 ($1234 was modified)
    POP  de                ; de = $0044, sp = $4846
    POP  ix                ; ix = $0002, sp = $4848
    ...
```

```
...
ORG  $8000
RET                     ; LD pc,(sp); INC sp; INC sp
                        ; jumps to address at top of stack, "returning" to caller
```

Résumé : `PUSH` stockera la valeur en haut de la pile, `POP` récupérera la valeur en haut de la pile, c'est une file d'attente *LIFO* (dernier *entré* , premier sorti). `CALL` est identique à `JP` , mais pousse également l'adresse de la prochaine instruction après `CALL` en haut de la pile. `RET` est similaire à `JP` , en sautant l'adresse de la pile et en y sautant.

Attention: lorsque les interruptions sont activées, le `sp` doit être valide pendant signal d' interruption, avec assez d' espace libre réservé à la routine de gestionnaire d' interruption, comme le signal d'interruption stockera l'adresse de retour (réelle `pc`) avant d' appeler la routine de gestionnaire, qui peut stocker des données supplémentaires sur empiler aussi bien. Toute valeur en avance sur `sp` peut donc être modifiée "de manière inattendue" si une interruption se produit.

Trick avancé: le Z80 avec `PUSH` prenant 11 cycles d'horloge (11t) et `POP` prenant 10t, le déroula `POP` / `PUSH` creux tous les registres, y compris `EXX` pour des variantes d'ombre, était le meilleur moyen de copier bloc de mémoire, encore plus vite que déroulé `LDI` . Mais vous devez chronométrer la copie entre les signaux d'interruption pour éviter la corruption de la mémoire. De même, `PUSH` déroulé était le moyen le plus rapide de remplir la mémoire avec une valeur particulière sur ZX Spectrum (à nouveau avec le risque de corruption par interruption, si ce n'était pas chronométré correctement, ou sous `DI`).

Lire La pile en ligne: https://riptutorial.com/fr/assembly/topic/4957/la-pile

Chapitre 5: Les interruptions

Remarques

Pourquoi avons-nous besoin d'interruptions

Imaginons: notre ordinateur est connecté à un clavier. Nous voulons entrer quelque chose. Lorsque nous appuyons sur la touche, rien ne se passe parce que l'ordinateur traite des choses différentes et ne remarque pas que nous voulons quelque chose de lui. Nous avons besoin d'interruptions!

Les interruptions sont déclenchées par un logiciel (*INT* 80h) ou du matériel (pression de touche), elles se comportent comme un *appel* (elles sautent à un endroit spécifique, exécutent du code et reviennent en arrière).

Examples

Travailler avec des interruptions sur le Z80:

Le Z80 n'a pas de table d'interruption comme les processeurs modernes. Les interruptions exécutent toutes le même code. En mode d'interruption 1, ils exécutent le code dans un emplacement non modifiable spécifique. En mode d'interruption 2, ils exécutent le code du registre de pointage sur lequel je pointe. Le Z80 a un minuteur, qui déclenche l'interruption tous les ~ 0,007s.

```
EI      ;enables Interrupts
DI      ;disables Interrupts
IM 1    ;sets the Normal Interrupt Mode


IM 2    ;sets the Advanced Interrupt Mode
LD I,$99;sets the Interrupt Pointer to $99 (just possible in IM 2)
```

Lire Les interruptions en ligne: https://riptutorial.com/fr/assembly/topic/6555/les-interruptions

Chapitre 6: Registres

Remarques

Que sont les registres?

Le processeur peut fonctionner sur des valeurs numériques (nombres), mais celles-ci doivent être stockées quelque part en premier. Les données sont stockées principalement dans la mémoire, ou à l'intérieur de l'opcode d'instruction (qui est généralement stocké dans la mémoire), ou dans une mémoire sur puce spéciale placée directement dans le processeur, appelé **registre** .

Pour travailler avec la valeur dans le registre, vous n'avez pas besoin de l'adresser par adresse, mais des mnémoniques spéciaux sont utilisés, comme par exemple `ax` sur x86, ou `A` sur Z80, ou `r0` sur ARM.

Certains processeurs sont construits d'une manière, où presque tous les registres sont égaux et peuvent être utilisés à toutes fins utiles (souvent de groupe RISC de processeurs), d' autres ont une spécialisation distincte, lorsque seuls certains registres peuvent être utilisés pour l' arithmétique (*« accumulateur »* sur les processeurs début) et d'autres registres pour l'adressage de mémoire uniquement, etc.

Cette construction en utilisant la mémoire directement sur la puce de processeur a un énorme implication de la performance, l' addition de deux nombres de registres stockant Retour à l' inscription se fait généralement dans le temps le plus court possible par ce processeur (exemple sur processeur ARM: `ADD r2,r0,r1` définit `r2` à `(r0 + r1)` valeur, en cycle de processeur unique).

Au contraire, lorsque l'un des opérandes fait référence à un emplacement mémoire, le processeur peut rester en attente pendant un certain temps, en attendant que la valeur vienne de la puce mémoire (sur x86, des valeurs de cache L0 Le processeur tourne lorsque la valeur ne se trouve dans aucun cache et doit être lu directement à partir de la mémoire DRAM.

Ainsi, lorsque le programmeur crée un code de traitement de données, il souhaite généralement que toutes les données soient en cours de traitement dans les registres pour obtenir les meilleures performances. Si ce n'est pas possible, et que des lectures / écritures de mémoire sont requises, celles-ci doivent être minimisées et former un modèle qui coopère avec l'architecture de cache / mémoire de la plate-forme particulière.

La taille native du registre en bits est souvent utilisée pour grouper les processeurs, comme *Z80* étant un *"processeur 8 bits"* , et *80386* étant un *"processeur 32 bits"* - bien que ce regroupement soit rarement clair. Par exemple, *Z80* fonctionne également avec des paires de registres, formant des valeurs natives de 16 bits, tandis que des CPU 32 bits *80686* ont des instructions MMX pour fonctionner avec des registres de 64 bits en mode natif.

Examples

Zilog Z80 enregistre

Registres: 8 bits: `A` , `B` , `C` , `D` , `E` , `H` , `L` , `F` , `I` , `R` , 16 bits: `SP` , `PC` , `IX` , `IY` et ombres de certains registres 8b: `A'` , `B'` , `C'` , `D'` , `E'` , `H'` , `L'` et `F'` .

La plupart des registres à 8 bits peuvent également être utilisés par paires en tant que registres 16 bits: `AF` , `BC` , `DE` et `HL` .

`SP` est un *pointeur de pile* , marquant le bas de la mémoire de la pile (utilisé par les instructions `PUSH` / `POP` / `CALL` / `RET`).
`PC` est un *compteur de programme* , pointant sur l'instruction en cours d'exécution.
`I` est un *registre d'interruption* , fournissant un octet haut de l'adresse de table vectorielle pour le mode d'interruption `IM 2` .
`R` est *un registre de rafraîchissement* , il s'incrémente chaque fois que le CPU récupère un opcode (ou un préfixe d'opcode).
Des instructions non officielles existent sur certains processeurs Z80 pour manipuler des parties de 8 bits de `IX` comme `IXH:IXL` et `IY` comme `IYH:IYL` .

Aucune instruction ne permet d'accéder directement aux variantes d'ombre, l'instruction `EX AF,AF'` `EXX` entre `AF` et `AF'` , et l'instruction `EXX` échangera `BC,DE,HL` avec `BC',DE',HL'` .

Chargement de la valeur dans un registre:

```
; from other register
LD   I,A        ; copies value in A into I (8 bit)
LD   BC,HL      ; copies value in HL into BC (16 bit)
; directly with value encoded in instruction machine code
LD   B,d8       ; 8b value d8 into B
LD   DE,d16     ; 16b value d16 into DE
; from a memory (ROM/RAM)
LD   A,(HL)     ; value from memory addressed by HL into A
LD   A,(a16)    ; value from memory with address a16 into A
LD   HL,(a16)   ; 16b value from memory with address a16 into HL
POP  IX         ; 16b value popped from stack into IX
LD   A,(IY+a8)  ; IX and IY allows addressing with 8b offset
; from I/O port (for writing value at I/O port use "OUT")
IN   A,(C)      ; reads I/O port C, value goes to A
```

Les combinaisons correctes des opérandes source et destination possibles sont limitées (par exemple, `LD H,(a16)` n'existe pas).

Stocker de la valeur dans une mémoire:

```
LD   (HL),D     ; value D stored into memory addressed by HL
LD   (a16),A    ; value A into memory with address a16
LD   (a16),HL   ; value HL into 16b of memory with address a16
LD   (IX+a8),d8 ; value d8 into memory at address IX+a8
LD   (IY+a8),B  ; value B into memory at address IY+a8
; specials ;)
PUSH DE         ; 16b value DE pushed to stack
CALL a16        ; while primarily used for execution branching
  ; it also stores next instruction address into stack
```

x86 registres

Dans le monde à 32 bits, les registres à usage général se divisent en trois classes générales: les registres à usage général à 16 bits, les registres à usage général étendus à 32 bits et les moitiés de registre à 8 bits. Ces trois classes ne représentent pas du tout trois ensembles de registres entièrement distincts. Les registres 16 bits et 8 bits sont en fait des noms de régions *à* l' *intérieur* des registres 32 bits. La croissance des registres dans la famille de processeurs x86 a été obtenue en *étendant les* registres existants dans les anciens processeurs

Il existe huit registres à usage général 16 bits: AX, BX, CX, DX, BP, SI, DI et SP; et vous pouvez y placer toute valeur pouvant être exprimée en 16 bits ou moins.

Quand Intel a étendu l'architecture x86 à 32 bits en 1986, il a doublé la taille des huit registres et leur a donné de nouveaux noms en préfixant un E devant chaque nom de registre, résultant en EAX, EBX, ECX, EDX, EBP, ESI, EDI et ESP.

Avec x86_64, on a doublé la taille du registre et ajouté de nouveaux registres. Ces registres ont une largeur de 64 bits et sont nommés (barre oblique utilisée pour afficher un autre nom de registre): RAX / r0, RBX / r3, RCX / r1, RDX / r2, RBP / r5, RSI / r6, RDI / r7, RSP / r4 , R8, R9, R10, R11, R12, R13, R14, R15.

Bien que les registres à usage général puissent être utilisés à des fins techniques, chaque registre a également un objectif alternatif / principal:

- AX (accumulateur) est utilisé dans les opérations arithmétiques.
- CX (counter) est utilisé dans les instructions shift et rotation, et utilisé pour les boucles.
- DX (data) est utilisé dans les opérations arithmétiques et E / S.
- BX (base) utilisé comme pointeur vers les données (en particulier en tant que décalage vers le registre de segment DS en mode segmenté).
- SP (stack) pointe vers le haut de la pile.
- BP (base de pile) pointe vers la base de la pile.
- SI (source) pointe vers une source en mémoire pour les opérations de flux (par exemple, `lodsb`).
- DI (destination) pointe vers une destination en mémoire pour les opérations de flux (par exemple, `stosb`).

Les registres de segments, utilisés en mode segmenté, pointent vers différents segments de la mémoire. Chaque registre de segment de 16 bits donne une vue à 64 ko (16 bits) de données. Une fois qu'un registre de segment a été défini pour pointer sur un bloc de mémoire, les registres (tels que `BX` , `SI` et `DI`) peuvent être utilisés comme décalages pour le registre de segment, permettant ainsi d'accéder à des emplacements spécifiques dans l'espace 64k.

Les registres à six segments et leurs utilisations sont les suivants:

registre	Nom complet	La description
SS	Segment de pile	Points à la pile

registre	Nom complet	La description
CS	Segment de code	Utilisé par le CPU pour récupérer le code
DS	Segment de données	Registre par défaut pour les opérations MOV
ES	Segment supplémentaire	Segment de données supplémentaire
FS	Segment supplémentaire	Segment de données supplémentaire
GS	Segment supplémentaire	Segment de données supplémentaire

x64 registres

L'architecture x64 est l'évolution de l'ancienne architecture x86, elle est restée compatible avec son prédécesseur (les registres x86 sont toujours disponibles), mais elle a également introduit de nouvelles fonctionnalités:

- Les registres ont maintenant une capacité de 64 bits;
- Il y a 8 autres registres à usage général;
- Les registres de segment sont forcés à 0 en mode 64 bits;
- Les 32, 16 et 8 bits inférieurs de chaque registre sont maintenant disponibles.

Polyvalent

registre	prénom	Sous-registres (bits)
RAX	Accumulateur	EAX (32), AX (16), AH (8), AL (8)
RBX	Base	EBX (32), BX (16), BH (8), BL (8)
RCX	Compteur	ECX (32), CX (16), CH (8), CL (8)
RDX	Les données	EDX (32), DX (16), DH (8), DL (8)
RSI	La source	ESI (32), SI (16), SL (8)
RDI	Destination	EDI (32), DI (16), DL (8)
RBP	Pointeur de base	EBP (32), BP (16), BPL (8)
RSP	Pointeur de pile	ESP (32), SP (16), SPL (8)
R8-R15	Nouveaux registres	R8D-R15D (32), R8W-R15W (16), R8B-R15B (8)

Remarque

Les suffixes utilisés pour adresser les bits inférieurs des nouveaux registres signifient:

- B octet, 8 bits;

- W mot, 16 bits;
- D double mot, 32 bits.

Lire Registres en ligne: https://riptutorial.com/fr/assembly/topic/4802/registres

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec le langage d'assemblage	Community, Edward, FedeWar, godisgood4, old_timer, Ped7g, Pichi Wuana, sigalor, stackptr, TheFrenchPlays Hd Micraftn, Zopesconk
2	Contrôle de flux	SpilledMango, tofro
3	Exemples Linux elf64 n'utilisant pas la glibc	Shift_Left
4	La pile	Jonas W., Ped7g
5	Les interruptions	Jonas W.
6	Registres	FedeWar, godisgood4, Jonas W., Ped7g, Pichi Wuana, SirPython