

1

سنحتاج الى بعض المتطلبات الاساسية لتعلم لغة التجميع (assembly language programming) باستخدام 64-bit MASM وايضا text editor للتعامل مع MASM source files و linker و C++ Compiler .

Setting Up MASM

MASM هو احد منتجات Microsoft وهو جزء من ادوات visual studio developer tools . يعمل على كافة اصدارات ويندوز لكن الان انا اتعامل معها على اصدار windows 10 ويمكنك ايضا التعامل معها على اصدار windows 11 او windows 7 الخ. لذلك على الاقل يجب عليك ان تفهم اساسيات الـ visual studio community والتعامل معه ارجح لك هذا الشرح :

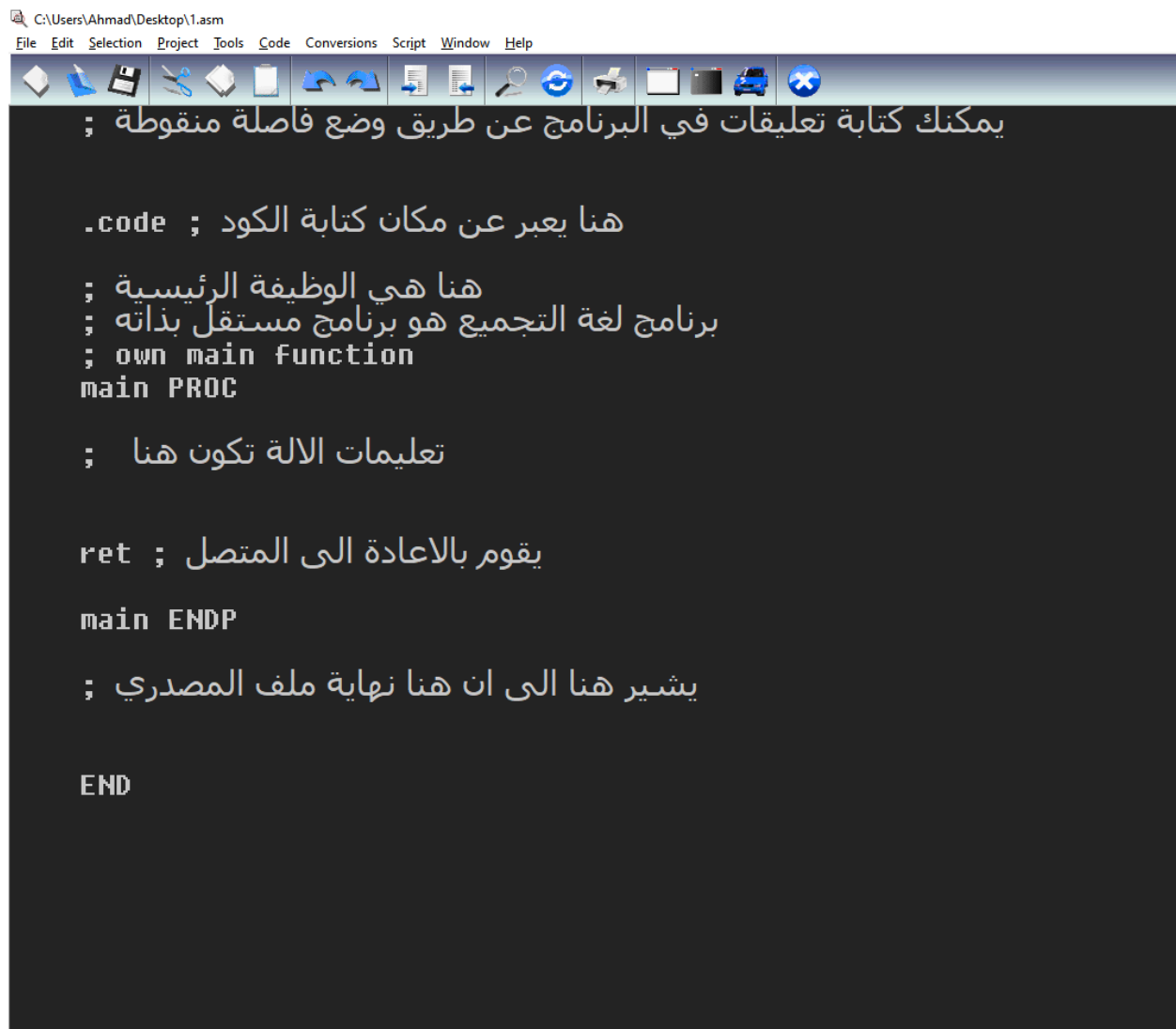
https://www.youtube.com/watch?v=REG-p_eFNlw

Setting Up a Text Editor

Visual Studio يتضمن محرر نصوص (Text Editor) ويمكنك استخدامه لإنشاء برامج C++ & MASM . اذا قمت بتنزيل Visual Studio للحصول على MASM فانك تحصل تلقائيا على محرر نصوص عالي الجودة ويمكنك استخدامه للتعامل مع الملفات المصدريه للغة التجميع. وعلى كل حال يمكنك استخدام اي محرر يمكنه التعامل مع ASCII Files لإنشاء ملفات مصدريه لـ C++ & MASM مثل Notepad++ او sublime او المتاح في <https://www.masm32.com> او حتى على الـ notepad العادي. برامج معالجة النصوص مثل Microsoft Word ليست مناسبة للملفات المصدريه للبرامج.

The Anatomy of a MASM Program

2-1



يحتوي برنامج MASM على قسم واحد او اكثر يمثل نوع البيانات التي تظهر في الذاكرة. تبدأ هاذي الاقسام (sections) مثل .code or .data. المتغيرات وقيم في الذاكرة تكون في .data. الـ Machine instructions تظهر ضمن قسم التعليمات البرمجية (.code). الخ تعتبر الاقسام الفردية التي تظهر في ملف مصدر لغة التجميع اختيارية (optional).

الـ code . هي مثال لتوجيه الـ Assembler عبارة تخبر MASM بشيء عن البرنامج ولكنها ليست تعليمة من الـ X86 OR 64 الفعلية. بمعنى آخر يخبر بشكل خاص MASM ان يجمع العبارات التالية له في قسم خاص من الذاكرة خاص في machine instructions .

Running Your First MASM Program

طباعة شيء بسيط في لغة التجميع مثل "Hello World" يعتبر اساسي . لطباعة هذا النص يجب عليك اولاً تعلم العديد من تعليمات الالة وتوجيهات المجمع وايضا استدعاءات النظام في ويندوز لطباعة سلسلة "Hello World" . النقطة من هذا الموضوع يعتبر طباعة Hello World امراً مختلفاً واعقد من لغات مثل C++ او Python بتعليمة واحدة يمكنك الطباعة لكن هنا الامر مختلف. صورة 1-2 يعتبر برنامج كامل يمكنك عمل عليه assemble (compile) وتشغيله. في الواقع هو لا ينتج اي outputs انه ببساطة ينتج شيء واحد فقط هو return . مع ذلك هو يعمل ويمكن توضيح اليه تجميع (assemble) ملف مصدر للغة التجميع وربطه (link) وتشغيله. MASM يعتبر (traditional command line assembler) بمعنى يتم استخدامه في Command line ومتوفر في cmd.exe . للقيام بذلك يمكنك تجربة :

ml64 1.asm /link /subsystem:console /entry:main

```
C:\Users\Ahmad\Desktop>ml64 1.asm /link /subsystem:console /entry:main
Microsoft (R) Macro Assembler (x64) Version 14.00.24247.2
Copyright (C) Microsoft Corporation. All rights reserved.

Assembling: 1.asm
Microsoft (R) Incremental Linker Version 14.00.24247.2
Copyright (C) Microsoft Corporation. All rights reserved.

/OUT:1.exe
1.obj
/subsystem:console
/entry:main

C:\Users\Ahmad\Desktop>
```

هذا الامر يخبر بتجميع ملف المصدري 1.asm حيث قمت بحفظ الملف 1-2 الى ملف قابل للتنفيذ وربط النتيجة لانتاج تطبيق console application (تطبيق يمكن تشغيله من خلال سطر الاوامر) بعدها يمكنك تنفيذ البرنامج . بافتراض عدم حدوث اي خلل في البرنامج يمكنك تشغيله هكذا :

C:\Users\Ahmad\Desktop>1.exe

صورة من داخل IDA :

```
;
; +-----+
; |       This file was generated by The Interactive Disassembler (IDA)       |
; |       Copyright (c) 2023 Hex-Rays, <support@hex-rays.com>                 |
; |       Freeware version                                                     |
; +-----+
;
; Input SHA256 : 9EF16EC3DB7343232B4FFB18B00A863669FF86BAAA63CD39668F6A93A03E5BC7
; Input MD5    : 75C2439323D6BE229D643B54770D31DC
; Input CRC32  : 94C11286
;
; File Name   : C:\Users\Ahmad\Desktop\1.exe
; Format      : Portable executable for AMD64 (PE)
; Imagebase   : 140000000
; Timestamp   : 659B4876 (Mon Jan 08 00:57:26 2024)
; Section 1. (virtual address 00001000)
; Virtual size      : 00000001 (    1.)
; Section size in file : 00000200 (   512.)
; Offset to raw data for section: 00000200
; Flags 60000020: Text Executable Readable
; Alignment        : default
;
;.686p
;.mmx
;.model flat
;
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64
assume cs:_text
;org 140001000h
assume es:nothing, ss:nothing, ds:_text, fs:nothing, gs:nothing
;
public start
start proc near
retn
start endp
```

بعد تشغيل البرنامج يجب Windows ان يستجيب فورا من خلال تنفيذ سطر في الـ cmd حيث 1.exe يقوم بارجاع التحكم الى windows بعد بدء تشغيله.

Running Your First MASM/C++

عملية الترجمة مع Assembly & C تختلف قليلا عن برنامج MASM مستقل سنوضح هنا كيفية انشاء وتجميع وتشغيل البرنامج التجميع المختلط مع C++ هذا هو البرنامج :

1.cpp

```
#include <stdio.h>

extern "C"
{
    // هنا الوظيفة الخارجية المكتوبة بلغة التجميع
    // في لغة سي هكذا يتم الاستدعاء

    void asmfunc(void);
}

int main()
{
    printf("%s", "Calling");
    asmfunc();
    printf("%s", "asmMain");
}
```

2.asm

```
.code
```

```
; no case mapping (يجب على البرنامج ان يتعامل مع الاحرف الكبيرة والصغيرة كل وحده على حده ولا يعدل تلقائيا تحويل)  
option casemap:none
```

```
; هنا الوظيفة  
public asmfunc  
asmfunc PROC  
ret  
asmfunc ENDP  
END
```

في 2.asm تحتوي على ثلاثة متغيرات واثنان منهم جديدة **option statement** و **public statement** . وهذا ماذا يعني ؟ الـ **option statement** تشير الى MASM لجعل كافة الرموز حساسة لحالة الاحرف. يعد هذا ضروريا لان MASM افتراضيا غير حساس لهاذي لحالة الاحرف ويقوم بتعين كافة المعرفات الى احرف كبيرة بحيث **asmfunc** تصبح **ASMFUNC** . C++ هي حساسة لحالة الاحرف تتعامل مع **asmfunc** و **ASMFUNC** كحالتين مختلفتين. الـ **public statement** يعلن انه عام هذا المعرف **asmfunc** وسيكون مرئيا خارج **source/object file** الخاص في MASM . بدون هذا البيان يمكن الوصول الى **asmfunc** فقط داخل MASM Module وسوف يشكل تحويل البرمجي في C++ الى معرف غير محدد. والفرق الثالث تم تغير **main** الى **asmfunc** لانه اذا تم استخدام نفس الاسم في كلتا الملفين **1.cpp & 2.asm** لهذا سيتم خلط لانه يوجد اسمين.

لتجميع هاذي الملفات المصدريه وتشغيلها بهذا الامر :

```
C:\Users\Ahmad\Desktop>ml64 /c 2.asm
Microsoft (R) Macro Assembler (x64) Version 14.00.24247.2
Copyright (C) Microsoft Corporation. All rights reserved.

Assembling: 2.asm

C:\Users\Ahmad\Desktop>cl 1.cpp 2.obj
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24247.2 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

1.cpp
Microsoft (R) Incremental Linker Version 14.00.24247.2
Copyright (C) Microsoft Corporation. All rights reserved.

/out:1.exe
1.obj
2.obj
```

المخرجات :

```
C:\Users\Ahmad\Desktop>1.exe
CallingasmMain
```

تم تشابك الاسماء بسبب عدم استخدام (\n) new line character .

صورة من داخل IDA :

```

; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64
assume cs:_text
;org 140001000h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing

; int __fastcall main(int argc, const char **argv, const char **envp)
main proc near
sub     rsp, 28h
lea     rdx, aCalling    ; "Calling"
lea     rcx, aS           ; "%s"
call    sub_1400010B0
call    nullsub_2
lea     rdx, aAsmmain     ; "asmMain"
lea     rcx, aS_0         ; "%s"
call    sub_1400010B0
xor     eax, eax
add     rsp, 28h
retn
main endp

```

asmfunc

الـ ml64 تم استخدام option /c الذي يرمز الى تحويل البرمجي فقط ولا يحاول تشغيل الـ linker لانه سيفشل لانه برنامج غير مستقل. Output من MASM هو ملف رمز كائن (object code) 2.obj والذي يعمل كادخال الى برنامج Compiler (Microsoft Visual C++ (MSVC)). يقوم الامر cl بتشغيل MSVC compiler على ملف 1.cpp والارتباطات موجودة في التعليمات البرمجية الي صار لها assembled code في 2.obj . الاخراج من برنامج MSVC compiler هو الملف القابل للتنفيذ 1.exe .

An Introduction to the Intel x86-64 CPU Family

حتى الان قمنا بتجميع وتشغيل برنامج واحد MASM وهو فقط يقوم باعادة التحكم الى Windows . قبل ان نتقدم اكثر وتعلم لغة التجميع من الضروري ان نلقي نظره على البنية الاساسية في Intel x86-64 CPU اذ لم تفهم هاذي البنية الـ machine instructions لن يكون لها اي معنى.

الـ Intel CPU family تصنف من الـ von Neumann architecture machine
تحتوي على ثلاث وحدات اساسية هي : CPU , Memory & I/O . ترتبط جميع هاذي الناقلات عبر
Data Bus ترتبط هاذي المكونات عبر system bus الذي يتكون من (address, data, and control buses).

لمعلومات اكثر تفصيلا :

<https://github.com/AhmaddF/Computer-Architecture/tree/main/Computer%20Architecture%20%26%20Organization/InputOutput>

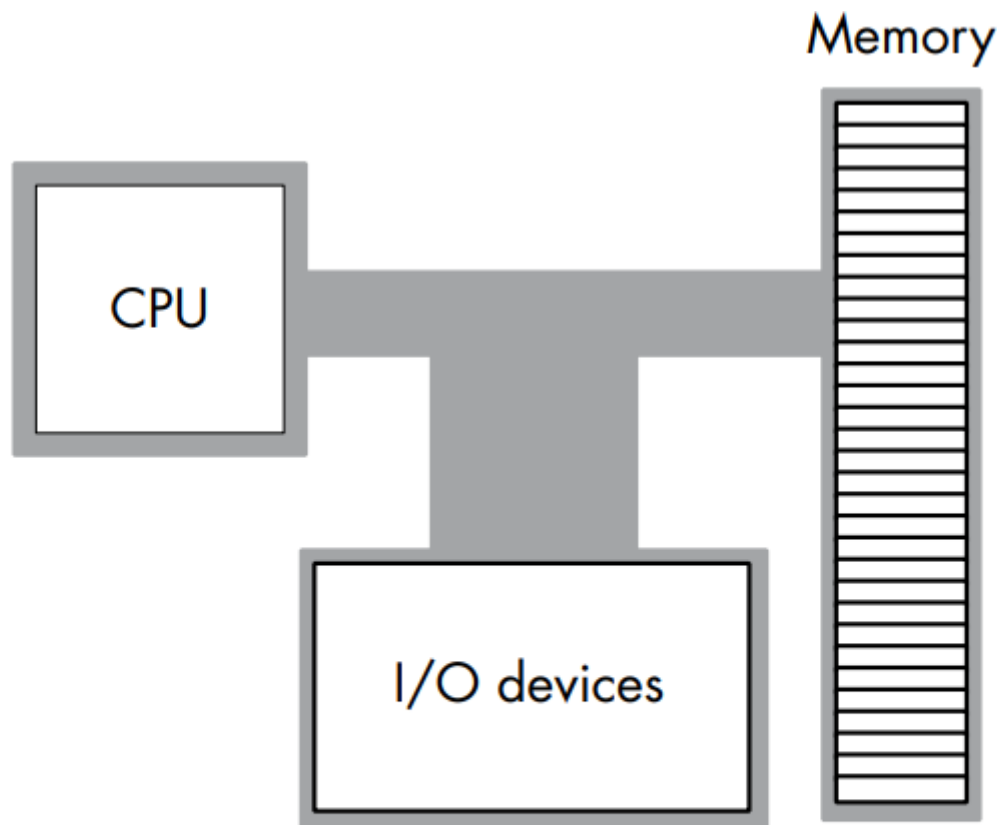


Figure 1-1: Von Neumann computer system block diagram

الصورة هاذي توضح هاذي العلاقة.

داخل الـ CPU يتم استخدام مواقع خاصة تعرف بالسجلات (Registers) لمعالجة البيانات يمكن تقسيم هاذي السجلات الى في CPU x86-64 الى اربع فئات : general-purpose registers و special-purpose application-accessible registers و segment registers و special-purpose kernel-mode registers .

الـ special-purpose kernel-mode registers مخصصة لكتابة انظمة التشغيل و debuggers و system-level tools اخرى. Intel family توفر العديد من السجلات للاغراض العامة للتطبيقات :

- Sixteen 64-bit registers that have the following names: RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, and R15
- Sixteen 32-bit registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, and R15D
- Sixteen 16-bit registers: AX, BX, CX, DX, SI, DI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, and R15W
- Twenty 8-bit registers: AL, AH, BL, BH, CL, CH, DL, DH, SIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B, and R15B

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

لسوء الحظ هاذي جميعها ليست سجلات مستقلة بل مقسمة الى بتات. مثال سجل 64 ينقسم الى 32 وينقسم الى 16 وينقسم الى 8. بمعنى انها ليست مستقلة تغير سجل واحد مثال من سجل ذو 64 بت يمكن ان يغير ثلاثة سجلات مع بعض مثال قمت بتعديل rdx الى edx,dx,dl ستتغير. اعتمادا على البيانات. احد الاخطاء الشائعة التي من الممكن الشخص يتعلم برمجة اسمبلي يصبح عنده (register value corruption) تلف قيمة السجل بسبب عدم الفهم الشامل للسجلات. بالاضافة الى السجلات العامة ايضا يوجد 8 سجلات خاصة الفاصلة العائمة (floating-point) يتم تنفيذها في وحدة floating-point unit (FPU). شركة Intel قامت بتسمية هاذي السجلات - ST(0) ST(7) هكذا. على عكس general-purpose registers لا يمكن للبرنامج الوصول اليها مباشرة. بدلا من ذلك يتعامل البرنامج مع floating-point register file باعتباره eight-entry-deep stack مكون من 8 entries ويصل الى مدخل او مدخلين علوين. يبلغ كل سجل فاصلة عائمة الى 80-bit wide. ويحمل قيمة حقيقة ذات دقة ممتدة (extended precision). على رغم Intel اضافت سجلات الفاصلة العائمة الاخرى الى x86-64 CPUs الان ان سجلات FPU لا تزال استخداما جيدا وشائع لانها تدعم 80-bit floating-point format.

في التسعينات قدمت Intel مجموعة من MMX register set وتعليمات لدعم single instruction والبيانات المتعددة (multiple data (SIMD) operations).
الى MMX register عبارة عن مجموعة من ثمانية سجلات 64 بت تتراكم مع سجلات ST(0) - ST(7) على FPU. الافضل لك الان الافضل ان تستخدم XMM registers (and instruction set) وترك FPU. الى RFLAGS او الى FLAGS هو سجل سجل 64 بت يحتوي على single-bit Boolean قيم. ثمانية من هاذي الى FLAGS او (البتات) تهم مبرمجي التطبيقات الذين يكتبون في Assembly يوجد interrupt disable و 2 sign و auxiliary carry و parity و carry flags. وايضا هاذي الاعلام تتيح لك حالات لاختبار نتيجة الحساب السابقة.

يوجد الكثير التعارضات الذي ظهرت وتم حلها لكن لن اطرق اليها جميعها موجودة في مجلدات Intel.

The Memory Subsystem

الى memory subsystem يحتفظ ببيانات البرنامج مثل متغيرات البرنامج والثوابت وتعليمات الجهاز ومعلومات اخرى. يتم تنظيم الذاكرة عن طريق الخلايا (cells). يمكن لنظام دمج المعلومات والتعامل معها على هذا الاساس.

لمعلومات اكثر تفصيلا :

x86-64 يدعم الذاكرة القابلة للعنونة بالبايت (byte-addressable memory) بمعنى ان وحدة الذاكرة الرئيسية هي عبارة عن بايتات. فكر بذاكرة كمصفوفة خطية من البايتات عنوان البايت الاول هو 0 وعنوان البايت الاخير هو $2^{32} - 1$. بالنسبة للمعالج x86 مع الذاكرة بسعة 4 جيجا يحد تعريف المصفوفة ك pseudo-Pascal array تقريبي جيد للذاكرة :

Memory: array[0....4294967295]

او ك :

Byte Memory[4294967295];

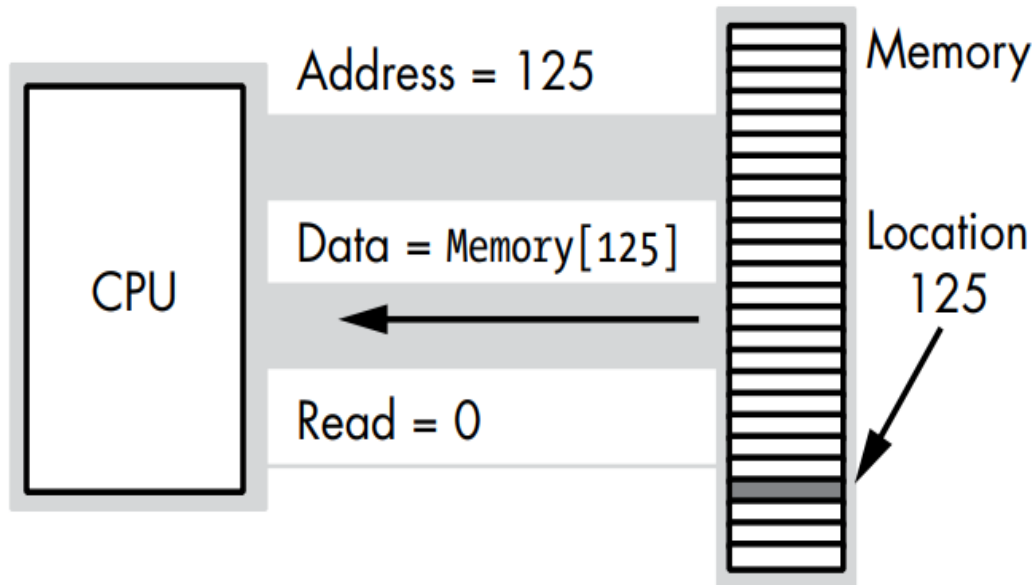


Figure 1-4: Memory read operation

على سبيل المثال لتنفيذ `Memory[125] := 0` ; تضع الـ CPU قيمة 0 على Data Bus وتضع العنوان على address bus وتؤكد write line .

Declaring Memory Variables in MASM

يمكن الاشارة الى عناوين الذاكرة باستخدام عناوين رقمية في لغة الاسمبلي الان ان القيام بذلك امر عرضه للخطأ وليس افضل شيء بدلا من ذلك يمكنك استخدام اسماء وتقم بتخزينها في متغيرات والاشارة عليها بمتغيرات استخدام اسماء المتغيرات يعد افضل بكثير من استخدام العناوين الرقمية للاشارة وايضا افضل في الكتابة والقراءة والصيانة. لأنشاء متغيرات بيانات يجب عليك وضعها في قسم البيانات data section في ملف المصدر في MASM يخبر هذا التوجيه MASM كافة العبارات التالية هي بيانات وسيتم تعرف عليها في البرنامج ك Data . في data section . تسمح MASM لك بتعريف كائنات ك متغيرات باستخدام (data declaration directives) الشكل الاساسي لتعريف البيانات هو :

`label directive ?`

حيث ال label هي معرف قانوني في MASM وهو احد التوجيهات (directives) التي تظهر في جدول هذا :

Table 1-2: MASM Data Declaration Directives

Directive	Meaning
byte (or db)	Byte (unsigned 8-bit) value
sbyte	Signed 8-bit integer value
word (or dw)	Unsigned 16-bit (word) value
sword	Signed 16-bit integer value
dword (or dd)	Unsigned 32-bit (double-word) value
sdword	Signed 32-bit integer value
qword (or dq)	Unsigned 64-bit (quad-word) value
sqword	Signed 64-bit integer value
tbyte (or dt)	Unsigned 80-bit (10-byte) value
oword	128-bit (octal-word) value
real4	Single-precision (32-bit) floating-point value
real8	Double-precision (64-bit) floating-point value
real10	Extended-precision (80-bit) floating-point value

علامة الاستفهام (?) تخبر MASM ان الكائنات لن يكون له قيمة صريحة عند تحميل البرنامج في الذاكرة بمعنى انها صفر مبدئيا. اذا كنت ترغب بوضع قيمة صريحة فاستبدل علامة الاستفهام مثل :

Initvalue sqword 1;

```
.data
u8 byte -1
s8 sbyte 32

.code

main PROC

mov al,u8
mov bl,s8

ret
main ENDP

END
```

MASM يتجاهل بغالب الاحيان بتجاهل بادئة S الي هي تعبر عن Signed او Unsigned . بل هي تعتمد اكثر على الـ machine instructions تفرق بين Signed or Unsigned لا يهتم اذا كان المتغير يحمل قيمة Signed او لا في هذا الكود يظهر ان لا يوجد مشكله ويمكن ان يعمل هذا الكود ويحمل قيمة Signed ويمكن عرض هذا من خلال خبرتك في الـ Software Debugging .

من الممكن حجز تخزين لقيم بيانات متعددة (multiple data) في single data declaration directive . الـ string يعد multi-valued data type يمكنك انشاء سلسلة من الاحرف منتهية بقيمة خالية في الذاكرة باستخدام byte directive على سبيل المثال :

```
StringName byte "Hello World",0
```

Declaring (Named) Constants in MASM

MASM يسمح لك باعلان constants باستخدام = . الـ constant هو symbolic name (identifier) يقرنه MASM بقيمة. في كل مكان يظهر الـ symbol في البرنامج مباشرة يقوم MASM باستبدال قيمة هذا الرمز بالرمز المعروف. مثال :

label = expression -> dataSize = 256 or dataSize equ 256 (equates);

Some Basic Machine Instructions

الـ x86-64 CPU family توفر ما يزيد عن مئات الى الاف من machine instruction اعتمادا على كيفية تحديد machine instruction . لكن معظم الـ assembly language programs تستخدم بين 30 – 50 تعليمة وهنا سنشرح كتابة العديد من البرامج ذات معنى القليل سنوفر مجموعة صغيرة من تعليمات الـ machine instructions حتى تتمكن من البدء في كتابة البرنامج MASM بسيط.

The mov Instruction

مبدئيا بدون اي شك تعليمة mov هي التعليمة الاكثر استخداما في برامج الـ اسمبلي في اي برنامج نموذجي ما بين 25% - 40% من التعليمات عبارة عن mov . من اسمها يمكنك فهمها تقوم بنقل البيانات من موقع الى اخر كيفية بناء هاذي الجملة :

mov destination_operand, source_operand

يمكن ان يكون الـ `source_operand` عبارة عن سجل للاغراض العامة (general-purpose) او متغير في الذاكرة (memory) (variable) او ثابت (constant) . وقد يكون الـ `destination_operand` عبارة عن سجل او متغير ذاكرة. في لغات مثل C++ تكون `mov instruction` مكافئة لهذا البيان التالي :

`destination_operand = source_operand`

يجب ان تكون الـ `mov instruction's operands` ان يكون بنفس الحجم. اي انه يمكنك نقل البيانات بين `byte (8-bit) objects` او `word (16-bit) objects` او `double-word (32-bit)` او `quad-word (64-bit) objects` .

يوضح هذا الجدول ما اقصد به

Source*	Destination
reg ₈	reg ₈
reg ₈	mem ₈
mem ₈	reg ₈
constant**	reg ₈
constant	mem ₈
reg ₁₆	reg ₁₆
reg ₁₆	mem ₁₆
mem ₁₆	reg ₁₆
constant	reg ₁₆
constant	mem ₁₆
reg ₃₂	reg ₃₂
reg ₃₂	mem ₃₂
mem ₃₂	reg ₃₂
constant	reg ₃₂
constant	mem ₃₂
reg ₆₄	reg ₆₄
reg ₆₄	mem ₆₄
mem ₆₄	reg ₆₄
constant	reg ₆₄
constant ₃₂	mem ₆₄

MASM يفرض بعض التحقيقات على المعاملات التعليمات. يجب ان يتوافق حجم المعاملات التعليمات.
على سبيل المثال هذا خطأ :

```
.data
i8 byte ?

main PROC

mov ax,i8

ret
main ENDP

END
```

المشكلة هي انك تحاول عمل Load لـ 8-bit variable الى 16-bit register . نظرا لان احجامها غير متوافقة يفترض MASM انه خطأ منطقي في البرنامج ويقوم بالابلاغ عن المشكلة :

```
3.asm(7) : error A2022:instruction operands must be the same size
```

الصحيح :

```
.data
i8 byte ?

main PROC

mov bl,i8

ret
main ENDP

END
```

The add and sub Instructions

تعليمات الـ add & sub تقوم بعمل عملية الاضافة (add) او الطرح (subtract) لمعاملين الـ syntax :

add destination_operand, source_operand -> **destination_operand = destination_operand + source_operand**

sub destination_operand, source_operand -> **destination_operand = destination_operand - source_operand**

The lea Instruction

بعض الاحيان تحتاج الى عمل load لعنوان معين على سبيل المثال متغير وعلى سبيل المثال ايضا موجود في سجل بدلا من القيمة نفسها. يمكنك استخدام lea (load effective address) لهذا الغرض. الـ syntax :

lea reg64, memory_var

الـ reg64 هو سجل 64-bit general-purpose و memory_var (ليس من الضروري ان يكون المتغير qword variable كما هو الحال مع mov & add,sub). كل متغير (variable) له عنوان في الذاكرة مرتبط به ويكون هذا العنوان في بنيات x64 دائما 64-bit مثال بسيط :

```
.data
i8 byte ?
strVar byte "String",0

main PROC

lea rax,strVar
ret
main ENDP

END
```

هذا المثال يقوم بتحميل في الـ RAX Register عنوان اول حرف في strVar . في تمثيل لغة C++ :

```
char strVar[] = "String";

char* RAX;
RAX = &strVar[0];
```

The call and ret Instructions and MASM Procedures

لأجراء function calls وبالإضافة كتابة وظائفك البسيطة تحتاج الى تعليمات call & ret . الـ ret instruction نفس الـ return الي موجودة في C++ فهي تعيد التحكم من assembly language procedure (الـ assembly language functions تسمى procedures في لغة اسمبلي). في هذا الشرح سنستخدم ret لوحدها لا تحتوي على معاملات :

ret

الـ call instruction تكون هكذا :

call proc_name

الـ proc_name هو اسم procedure التي تريد عمل لها call . بعد كتابة القليل من البرامج الجدا بسيطة MASM procedure يتكون من هذا السطر :

proc_name proc

;any

proc_name endp

هكذا تبدأ وهكذا تنتهي من عند endp وفي نصف يمكنك كتابة التعليمات البرمجية التي تخص هذا . procedures

```

.code

myproc proc

    ret ; يعود الاتصال الى المتصل

myproc ENDP

main PROC

call myproc ; هنا يتم الاتصال بـ procedure
ret

main ENDP

END

```

الكود هذا يوضح MASM procedure .

Calling C/C++ Procedures

على الرغم ان كتابة procedure واستدعاءها امر مفيد للغاية. فان سبب ادخال الـ procedures في هذه المرحلة ليس السماح لك بكتابة الـ procedures الخاص بك بل لاعطائك القدرة على استدعاء الـ procedures المكتوبة بـ C/C++. تعد كتابة الـ procedures الخاصة بك لتحويل البيانات واخراجها الى الـ Console لحد الان هي تعتبر معقدة قليلا. بدلا من ذلك يمكنك استدعاء دالة printf() الموجودة في لغة C/C++ لانتاج مخرجات البرنامج والتحقق ان برنامجك تقوم بالفعل بشيء عند تشغيلها . لسوء الحظ اذا قمت باستدعاء printf() في لغة اسمبلي الخاص بك دون توفير procedure printf() مجمع MASM سيعطي خطأ ان الرمز غير محدد (undefined symbol). لاستدعاء اجراء خارج الـ Source Code تحتاج الى استخدام "externdef directive" بناء الجملة على النحو التالي :

externdef symbol:type

الـ symbol هذا هو رمز خارجي الذي تريد تعريفه والـ type هو نوع الـ Symbol والذي سيكون (proc for external procedure definitions) لتحديد رمز printf() هكذا سيكون :

externdef printf:proc

الـ externdef لا يتيح لك تحديد معلمات لتمريرها الى اجراء printf() ولا توفير تعليمات لتحديد الـ (parameters) . بدلا من ذلك يمكنك تمرير ما يصل الى 4 معلمات الى دالة printf في سجلات RCX,RDX,R8 & R9 . تتطلب دالة printf() ان تكون المعلمة الاولى هو الـ format string . لذلك يجب تحميل (Load) RCX بعنوان سلسلة منتهية بصفر قبل استدعاء printf() . اذا كانت سلسلة التنسيق تحتوي على احد محددات التنسيق (على سبيل المثال مثل %s or %d) فيجب عليك تمرير قيم المعلمات المناسبة في R9 & R8 , RDX .

Hello, World!

الان لدينا المعلومات الكافية لكتابة Hello, World! .

```
option casemap:none

.data
Str0g byte "Hello, World!",0

.code

قمنا بتعريف دالة printf
externdef printf:proc ; printf

    public asmfunc
    asmfunc proc; asmfunc قمنا بصنع وظيفة تسمى

    sub rsp,56; setup stack (بدون تفسير الان)

    lea rcx,Str0g ; قمنا بتحميل السلسلة بريجستير لاستدعاءها
    call printf; تم الاتصال

    add rsp,56; end stack (بدون تفسير الان)

    asmfunc ENDP

END
```

```

#include <stdio.h>

extern "C"
{

    void asmfunc(void);

}

int main()
{

    printf("%s", "Calling\n");
    asmfunc();
    printf("%s", "\nReturning");

}
=

```

عملية التجميع :

1 - ml64 /c 3.asm

2 - cl 4.cpp 3.obj

3 – output 4.exe :

```


C:\Users\Ahmad\Desktop\assembly ess>4.exe
Calling
Hello, World!
Returning

```

يمكنك إضافة "Hello, World!",10,0 Stri0g byte أيضا 10 لعمل new line لكن انا قمت بعمل new line عن طريق printf("%s","\nReturning") .

Returning Function Results in Assembly Language

شرحنا كيفية تمرير تعليمات الى procedure مكتوب بلغة اسمبلي هنا سنصف عملية معاكسة : ارجاع قيمة الى التعليمات البرمجية التي استدعت احد procedures الخاص بك. في pure assembly language حيث يستدعي احد الاجراءات اجراء اخر يعد تمرير الملمات (passing parameters) ونتائج الوظائف المرتجعة (returning function) عبارة عن اتفاقية تتشاركها اجراءات المتصل والمستدعي مع بعض. يمكن للمتدعي (calle) (الاجراء الذي يتم استدعاؤه) او المتصل (الاجراء الذي يقوم بالاتصال) (caller) اختيار مكان ظهور النتائج. من وجهة نظر الـ calle يحدد الاجراء الذي يعيد القيمة المكان الذي يمكن للمتصل العثور على نتيجة الوظيفة فيه ومن يستدعي تلك الوظيفة يجب ان يحترم هذا الاختيار . على سبيل المثال اذا قام احد الـ procedure بارجاع قيمة في سجل XMM0 سجل شائع لحفظ قيمة الـ Return الفاصلة العائمة (floating-point). فيجب على من يستدعي هذا الاجراء ان يتوقع العثور على النتيجة في XMM0 . قد يؤدي اجراء اخر حفظ النتيجة في سجل RBX . من وجهة نظر الـ caller's يتم عكس الاختيار. تتوقع التعليمات البرمجية الموجودة ان تقوم الوظيفة بارجاع نتيجتها في موقع معين ويجب ان تحترم الوظيفة التي يتم استدعاؤها . في النهاية من المهم ان تعرف عندما تكتب اسمبلي فان طريقة تمرير البيانات الى الاجراء . المره الوحيدة التي يجب ان تقلق هو بشأن الالتزام بـ ABI هي عندما تتصل برمز خارج عن سيطرتك اذا كان هذا الرمز الخارجي يتصل بكودك. سنغطي هنا كتابة لغة التجميع ضمن Microsoft Windows هو كود لغة الاسمبلي الذي يتفاعل مع MSVC . لذلك عند التعامل مع التعليمات البرمجية الخارجية (Windows and C++ code) يجب عليك استخدام Windows/MSVC ABI . ينص Microsoft ABI ايضا على ان الوظائف (الاجراءات) ترجع قيما صحيحة ومؤشرا (pointer) تتناسب مع X64 في سجل RAX . لذلك اذا كانت بعض التعليمات C++ تتوقع ان يقوم اجراء الخاص بك بارجاع قيمة صحيحة فيمكنك تحميل (Load) النتيجة العدد الصحيح الى RAX مباشرة قبل العودة (returning) الى الاجراء الخاص بك.



```

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern "C"
{
    void asmmain(void);

    char* gettitle(void);

    int readline(char* dest, int maxLen);
};

int readline(char* dest, int maxLen)
{
    char* result = fgets(dest, maxLen, stdin);
    if (result != NULL)
    {
        int len = strlen(result);
        if (len > 0)
        {
            dest[len - 1] = 0;
        }
        return len;
    }
    return -1;
}

int main(void)
{
    try
    {
        char* title = gettitle();

        printf("Calling %s:\n", title);
        asmmain();
        printf("%s terminated\n", title);
    }
    catch (...)
    {
        printf
        (
            "Exception \n"

        );
    }
}

```

اول شيء الـ `try..catch` تلتقط اي استثناءات ينشئها كود اسمبلي بحيث تحصل على نوع الاشارة اذا تم احباط هذا البرنامج . هذا الكود يوفر مفاهيم جديدة ابرزها عن الـ `return` . تقوم دالة `gettitle` بارجاع `pointer` الى سلسلة سيطبعتها كود C .

في قسم الـ `data` . ستجد هذا :

input byte maxLen dup (?)

الـ `maxLen dup (?)` تخبر MASM يفعل تكرار (`duplicate`) بعدد مرات ؟ (اي بايت غير مهياً).

```

option casemap:none
NewLine = 10 ; ASCII code for newline
maxLen = 256 ; Maximum Size String

.data

titleString byte "Assembly Language",0
prompt byte "Enter String:",0
fmtStr byte "User entered: '%s'" , NewLine , 0

input byte maxLen dup (?)

.code
externdef printf:proc
externdef readline:proc

    public gettitle
gettitle proc

    lea rax,titleString
    ret

gettitle ENDP

asmmain proc

    sub rsp,56

    lea rcx,prompt
    call printf

    mov input, 0

    lea rcx, input
    mov rdx, maxLen
    call readline

    lea rcx,fmtStr
    lea rdx,input
    call printf

    add rsp,56
    ret

asmmain ENDP

END

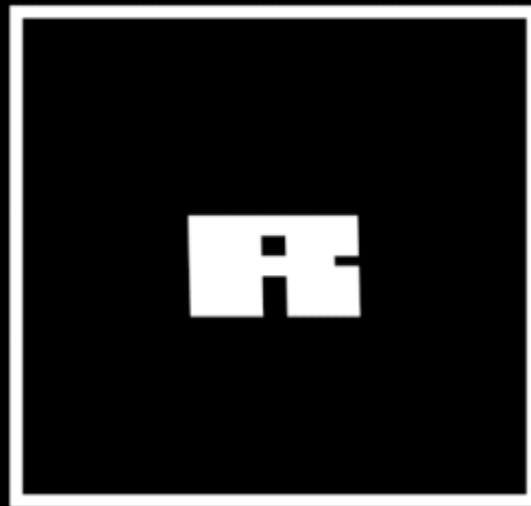
```

لنقوم بعمل compile :

ml64 /c new.asm

cl /EHa c.cpp new.obj

```
C:\Users\Ahmad\Desktop\assembly ess>c.exe  
Calling Assembly Language:  
Enter String:Ahmad  
User entered: 'Ahmad'  
Assembly Language terminated
```



Twitter : https://twitter.com/dr_retkit

YouTube : <https://www.youtube.com/@retkit1823>