GUÍA TEÓRICA PARA EL LABORATORIO DE ORGANIZACIÓN Y ARQUITECTURA DE COMPUTADORAS

SESIÓN N°:	1	TEMA:	PROGRAMACIÓN EN LENGUAJE ENSAMBLADOR

INDICACIONES GENERALES

- Revisar este material antes del desarrollo de la sesión de laboratorio correspondiente.
- Para un mejor desempeño, complementar el estudio de esta guía con el material del curso y la bibliografía indicada.
- Consultas puntuales sobre el contenido de esta guía se deben realizarse a través del foro correspondiente.

OBJETIVOS

- Brindar al alumno claridad de conceptos acerca de la programación en modo real y modo protegido.
- Aprender a utilizar la herramienta nasm para compilar archivos .asm.
- Codificar programas en lenguaje ensamblador.

ÍNDICE

1. Programación en lenguaje ensamblador

La programación en lenguaje ensamblador requiere el uso de registros, espacios de memoria, etiquetas e interrupciones para que el programador pueda realizar las acciones que requiera.

1.1. Registros High-Low 16 bits

Existen cuatro registros de propósito general: AX, BX, CX y DX. Cada registro puede contener un "word" de datos. Estos registros tienen la capacidad de poder separarse en partes más pequeñas de 8 bits cada uno; por ejemplo, AH y AL son la parte HIGH y LOW (8 bits), respectivamente (ver Lista 1).

Lista 1: Ejemplo de High-Low

```
1 ax = 0x7842 -> ah = 0x78     &    al = 0x42
```

1.2. Registros High-Low 32 bits

Los registros se extienden a 32 bits de tamaño y se denotan prefijando la letra "e" al nombre del registro, haciendo referencia a "extended" o registros extendidos. (Ver Lista 2).

Lista 2: Ejemplo registros 32 bits

```
1 mov edx, 0xb8000
2 add edx, 2
3 mov eax, ebx
4
5 variable1  db  41        ;sintaxis en 32 bits para reservar
6                          ;variables con valor
7 variable2  dw  41
```

1.3. Asignación de variables

Para la asignacíon de valores a nuestros registros se utiliza el operador **mov**. En la Lista 3 se observa el uso de dicha instrucción.

Lista 3: Ejemplo de uso de mov

```
1 mov ax, 0x7842          ;Guardar en ax el valor 0x7842
2 mov bx, 7842            ;Guardar en bx el valor 7842
3 mov cx, 'A'             ;Guardar en cx el valor ASCII de 'A'
4 mov bx, ax              ;Guardar en bx el valor de ax
```

Por otro lado, es posible declarar espacios de memoria mediante el uso de etiquetas. Dependiendo el espacio de memoria que se quiere utilizar, se usa “db” (declare byte(s)) o “dw” (declare word) seguido de los datos a guardar. (ver Lista 4).

Lista 4: Ejemplo declarar variable

```
1 mi_variable:
2     db "Hola Mundo",10
```

Las etiquetas también tienen la utilidad de definir una ubicación dentro del programa, de manera que, en conjunto con una operación de salto, permite el apuntar a direcciones específicas del código (ver Lista 5).

Lista 5: Ejemplo referencia de la etiqueta

```
1
2 mov ax,expresion     ;mover la direccion de la expresion
3                      ;en el registro ax
4 mov ax,[expresion]   ;mover el contenido de la variable
5                      ;expresion en el registro ax
```

IMPORTANTE: Para que una instrucción pueda realizar una operación, todos los operandos deben ser del mismo tamaño. No se puede operar un registro de 16 bits con uno de 8 bits (i.e. mov ax, bl).

1.4. Operaciones y Banderas

Para realizar operaciones se seguirá el formato definido para la asignación de variables: “INSTRUCCIÓN DESTINO, FUENTE”. Se pueden realizar operaciones aritméticas, lógicas y de control las cuales estarán definidas en el anexo[1].

Cuando se ejecuta una instrucción se alteran valores dentro del registro de banderas. Estas banderas están definidas en la Tabla 1.

Flag	Descripción	Valor : 1	Valor: 0
CF	Bandera de acarreo	Acarreo	No acarreo
ZF	Bandera de cero	Resultado cero	No cero
SF	Bandera de signo	Negativo	Positivo
OF	Bandera de desborde	Overflow	No overflow
PF	Bandera de paridad	Par	Impar
AF	Bandera de ajuste	Acarreo auxiliar	No acarreo auxiliar
IF	Bandera de interrupciones	Habilitadas	Desahabilitadas

Tabla 1: Descripción del registro de banderas

[1] Paideia: /Laboratorio 1: Programación en lenguaje ensamblador /Anexo.pdf

1.5. Modos de direccionamiento

Las operaciones se realizan de diversas maneras. Por ello, los modos de direccionamiento determinan el lugar en que reside un operando, un resultado o la siguiente instrucción a ejecutar (según sea el caso). Existen diversas formas de direccionar cuando se trabaja con registros de 16 bits (ver Figura 1) y con registros de 32 bits (ver Figura 2). **NOTA: En la Lista 6 se muestra un ejemplo con los modos de direccionamiento que utilizaremos en el laboratorio.**

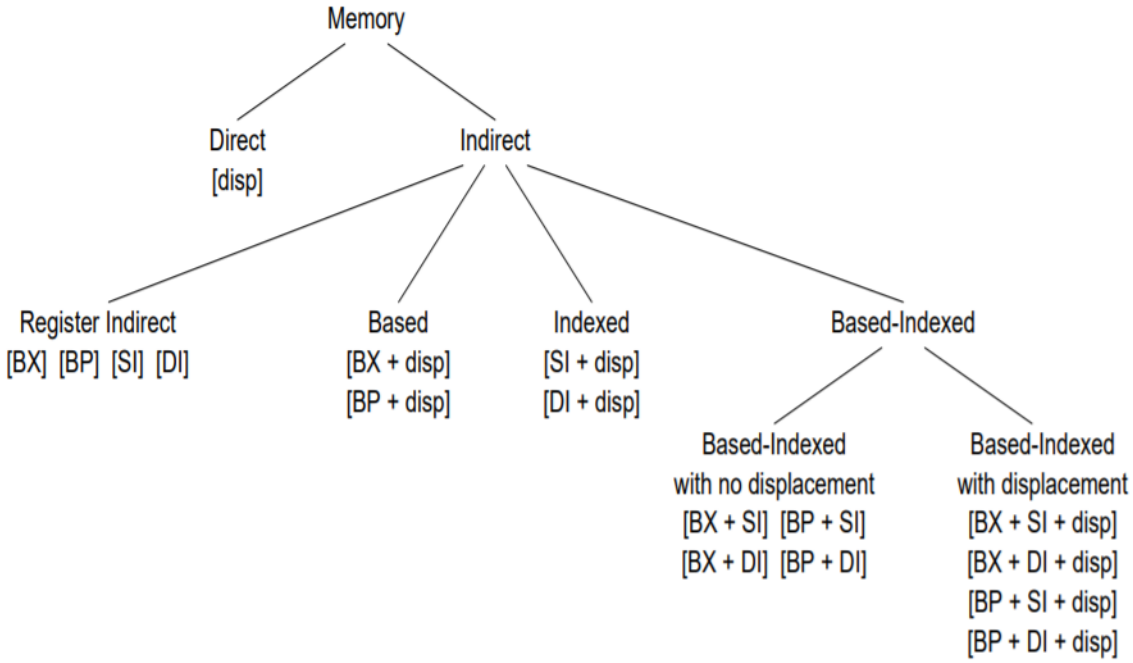

Fig. 1: Modos de direccionamiento de memoria para 16 bits.

	Direccionamiento de 16 bits	**Direccionamiento de 32 bits**
Registro base	BX, BP	EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
Registro índice	SI, DI	EAX, EBX, ECX, EDX ESI, EDI, EBP
Factor de escalamiento	Ninguno	1, 2, 4, 8
Desplazamiento	0, 8, 16 bits	0, 8, 32 bits

Tabla 2: Diferencias entre registros utilizados para direccionamiento de 16 y 32 bits

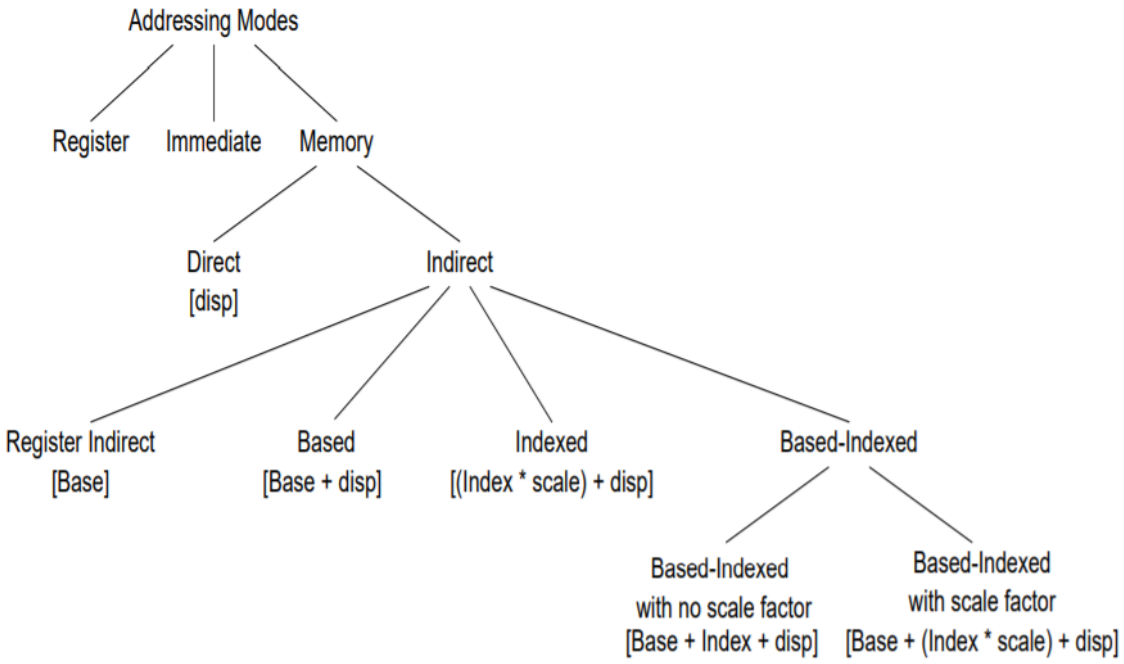

Fig. 2: Modos de direccionamiento para registros de 32 bits.

1.5.1. Direccionamiento Inmediato

El operando aparece directamente en la instrucción (ver ejemplo en Lista 6).

1.5.2. Modo registro

El operando es un registro (ver ejemplo en Lista 6).

1.5.3. Directo absoluto a memoria

El operando es una dirección de memoria a la que se desea acceder (ver ejemplo en Lista 6, notar que en dicho ejemplo, se copia en AX el contenido de DS:078AH).

1.5.4. Directo relativo a un registro base

El operando es una dirección de memoria a la que se desea acceder y se calcula mediante un registro base (BX o BP si es segmento DS o SS, respectivamente).

1.5.5. Directo relativo a un registro índice

El operando es una dirección de memoria a la que se desa acceder, y se calcula en base a un registro índice. Este registro índice es SI o DI.

Lista 6: Ejemplos de direccionamiento usando registros de 16 bits

```
1 mov ax,1234H          ;Direccionamiento directo
2 mov ax,bx             ;Modo registro
3 mov ax,[078AH]        ;Directo absoluto a memoria
4 mov ax,[bx+2]         ;Directo relativo a un registro base
5 mov ax,[si+10]        ;Directo relativo a un registro indice
```

1.6. Pila o Stack

El stack es un espacio de memoria el cual se utiliza para almacenar temporalmente nuestras variables. Para realizar este almacenamiento se emplean “push” registro y “pop” registro para almacenar y devolver el valor del registro, respectivamente. Si se usa “pusha” y “popa” se hará el almacenamiento y la devolución de todos los registros en un solo llamado (ver Lista 7).

El stack almacena la información con el método “LIFO” (Last in, First Out).

Lista 7: Operaciones con la pila

```
1 mov ax, 4         ; ax = 4
2 push ax           ; almacena el valor 4 en el stack
3 mov ax, 5         ; ax = 5
4 pop ax            ; ax = 4
5
6 pusha    ;Guarda los valores dentro de los registros en la pila
7 popa     ;Recupera los valores en sus registros correspondientes
```

Los registros “bp” y “sp” almacenan las direcciones de memoria de la base del stack y del punto tope del stack, respectivamente. Esto significa que cuando bp sea igual a sp, el stack esta vacío. Si trataramos de hacer pop en nuestro stack vacío causaremos un error de **overflow**.

1.7. Saltos, estructuras de control y ciclos iterativos

Para evitar la repetición de bloques de código se puede hacer uso de estructuras de control utilizando etiquetas y las instrucciones de control que se encuentran en el anexo[2]. Por ejemplo, en la Lista 8 se tiene un bloque escrito en lenguaje ensamblador y en la Lista 8 se tiene su equivalente en un lenguaje de alto nivel:

[2]Paideia: /Laboratorio 1: Programación en lenguaje ensamblador/Anexo.pdf

Lista 8: Estructura de control(ensamblador)

```
1  cmp ax, 4
2  je if_cuatro
3  mov bx, 45
4  jmp fin
5
6  if_cuatro:
7      mov bx, 23
8  fin:
9      .
10     .
```

Lista 9: Estructura de control en C

```
1  if (ax == 4){
2      bx = 23}
3  else{
4      bx=45}
5      .
6      .
7      .
8      .
9      .
10     .
```

1.8. Interrupciones

Las interrupciones son un mecanismo que permite a la CPU detener su acción presente y ejecutar otras líneas de instrucciones de mayor prioridad para luego volver a la tarea original. Para ello, se hace el llamado a las interrupciones con el comando **int** seguido del código de la interrupcion y funciona en conjunto con determinados registros para accionar. Es importante entender que toda interrupción es única y que su uso dependerá de ciertos códigos provistos en determinados registros. Por ejemplo, vamos a analizar la instrucción **INT 10H**. Esta instrucción controla los servicios de pantalla de la PC; sin embargo, necesita de un código para saber si debe entrar en modo video, imprimir un caracter,cambio de color, entre otros[3].

Uno de sus usos se puede ver en la Lista 10. En la parte high de AX (AH), se le asigna el código para mostrar en pantalla, mientras que en la parte low de AX (AL) se almacena el caracter que se desa imprimir.

Lista 10: Ejemplo interrupción

```
1  ;Uso de la  interrupcion 0x10 con el codigo 0x0e
2  mov ah, 0x0e          ; funcion 0x0e mostrar caracter en pantalla
3  mov al, 'A'
4  int 0x10              ; Interrupcion que en conjunto con ah,
5                        ; Muestra  el caracter guardado en al
```

Otro ejemplo que cobra gran relevancia es la interrupción 80H. De forma similar a la anterior, la manera de utilizarla es colocando **INT 80H**. Esta instrucción provoca una interrupción de software y realiza llamadas al sistema (*system calls*) en GNU/Linux. Esta interrupción también funciona dependiendo el código que se le asigne a EAX, EBX, ECX y EDX.

En la Lista 11, se observa un ejemplo representativo de esta interrupción. En este caso, se usan dos veces la interrupción pero con diferente tanto en EAX como en EBX. Su primer uso es para imprimir el mensaje ´´Hola". Para ello, lo que realiza es activar el servicio de escritura (EAX=4), el estandar de salida (EBX=1), la dirección de memoria

[3]Para mayor detalle de los códigos, ingresar a "Lista de INT 10H, INT 16H y INT 21H".

(ECX=texto) y el tamaño del texto a imprimir (EDX=5). Por otro lado, en ese mismo ejemplo, se puede ver que al terminar de realizar esta impresión, se activa el servicio de salida y retorno al proceso que lo invocó (EAX=1), así como retornar 0 sin errores(EBX=0).

Lista 11: Ejemplo interrupción

```
1  ;Uso de la  interrupcion 0x80H
2  section .data
3  texto db "Hola",10
4  section .text
5  global _start
6
7  _start:
8      mov EAX,4
9      mov EBX,1
10     mov ECX,texto
11     mov EDX,5
12     int 80H
13 fin:
14     mov EAX,1
15     mov EBX,0
16     int 80H
```

2. Modos de operación del CPU

2.1. Modo Real

El modo real trabaja con lo mínimo necesario que necesita el sistema para poder operar. Es por ese motivo que trabaja con 16 bits. Para la compatibilidad con versiones anteriores, es importante que las CPU se inicien en modo real. Esto permite que los sistemas operativos (SO) modernos puedan cambiar a modo protegido, y a su vez que los SO más antiguos continuen ejecutándose en una computadora moderna de manera consistente.

2.2. Modo Protegido

El modo protegido es mucho más potente que el modo real y es utlizado por los SO multitarea de la actualidad. Su nombre proviene de que cada bloque de programa tiene su propio espacio de memoria asignado, estando protegidos del conflicto con otros programas; y el manejo de registros ahora de 32 bits. Éste modo nos trae ventajas tales como acceso completo a la memoria (no el límite de 1MiB del modo real) y la posibilidad de realizar multitareas.

2.3. Cambio a Modo Protegido

Para hacer uso más completo de la CPU y comprender mejor como los desarrollos de las arquitecturas de los CPU pueden beneficiar al funcionamiento de los sistemas modernos como lo es la protección de la memoria en hardware, es necesario abordar a modo protegido.

2.3.1. Global Descriptor Table (GDT)

Para cambiar el CPU de modo real a protegido, para lo que debemos preparar una estructura de datos compleja en la memoria llamada Tabla de Descriptores Globales (GDT).

Un descriptor de segmento es una estructura de 8 bytes que define las siguientes propiedades de un segmento en modo protegido:

- Dirección base (32 bits) que define dónde comienza el segmento en la memoria física.
- Límite de segmento (20 bits) que define el tamaño del segmento.
- Varios indicadores (12 bits) que afectan la forma en que la CPU interpreta el segmento como el privilegio o si es de escritura o lectura.

En la Figura 3 se observa el orden de estos 64 bits. No todas las secciones se encuentran agrupadas, por ello al momento de preparar la GDT se recomienda realizarlo por partes. A continuación, se menciona las configuraciones que el GDT nos permite:

- Base: 0x0.
- Límite: 0xFFFF , hasta máximo 4GiB.
- Presente: 1 si el segmento está presente en la memoria, o sea, si es un sector válido.
- Privilegio: son 2 bits, siendo 00 el más alto (kernel), 11 el más bajo (aplicaciones de usuario).
- Tipo de descriptor: 1 para código o segmento de datos, 0 es usado para segmentos del sistema.
- Bits de segmento de tipo:
  -código: 1 para código
  -bit de conformidad: 0 sólo si se puede ejecutar desde el nivel de privilegio 1, es 1 si el código en este segmento se puede ejecutar desde un nivel de privilegio igual o inferior.
  -bit de escritura: 1 si se puede escribir, 0 si solo se puede ejecutar.
  -bit de acceso: 0, cuando la CPU acceda al segmento establece el bit en 1.
  -bit de granularidad: 0 si el límite máximo es 1MB, es 1 si el límite es 4GB.
  -bit 32 por defecto: 1 para 32 bits, 0 en 16 bits.
  -bit de segmento de código de 64 bits: 0 en modo 32 bits, 1 en 64 bits
  -AVL: 0, para uso del software del sistema operativo.

Fig. 3: Tabla de descriptores globales

2.3.2. Cambio a modo protegido

Una vez preparado el descriptor GDT dentro del sector de arranque, se procederá con indicarle al CPU que cambie de modo real (16 bits) a modo protegido (32 bits). Primero, se debe desactivar las interrupciones usando el operador “cli”, lo que dará prioridad al código por ejecutar sin que sea interrumpido por factores en paralelo u otros programas. Luego, indicar al CPU sobre el GDT que acabamos de preparar haciendo uso de la instrucción “lgdt [gdt_descriptor]”. Para terminar, solo queda hacer el cambio a modo protegido alterando el primer bit del registro de control de CPU “cr0”. Para poder modificar directamente este bit se procede a cargar el registro “cr0” en uno de nuestros registros generales como “ax” y utilizar el operador lógico “or” para actualizar solo su último bit.

Lista 12: Ejemplo tabla de descriptores

```
1  ;GDT
2  gdt_start:
3  gdt_null:
4      dd 0x0
5      dd 0x0
6  gdt_code:
7  ;base 0x0 , limite 0xfffff(4 gigas)
8  ;Flags pt1:(present)1(privilegio)00(tipo descriptor)1 = 1001b
9  ;Flags de tipo:(code)1(conforming)0(readable)1(accessed)0 = 1010b
10 ;Flags pt2:(granularity)1(32-bit default)1
11 ;            (64-bit seg)0(AVL)0 = 1100b
12     dw 0xffff        ;Limite(bits 0-15)
13     dw 0x0           ;Base (Bits 0-15)
14     db 0x0           ;Base (Bits 16-23)
15     db 10011010b     ;High:Flags parte 1 , Low: Flags de tipo
16     db 11001111b     ;High:Flags parte 2 , Low: Limite(bits 16-19)
17     db 0x0           ;24-31 bits base
18
19 gdt_data:
20 ;Igual que la seccion anterior excepto la parte de
21 ;Flags de tipo: (code)0(expand down)0
22 ;                (writable)1(accessed)0 = 0010b
23     dw 0xffff        ;Limite(bits 0-15)
24     dw 0x0           ;Base (Bits 0-15)
25     db 0x0           ;Base (Bits 16-23)
26     db 10010010b     ;High:Flags parte 1 , Low: Flags de tipo
27     db 11001111b     ;High:Flags parte 2 , Low: Limite(bits 16-19)
28     db 0x0           ;24-31 bits base
29
30 gdt_end:         ; Con esta etiqueta al final podemos
31                  ; calcular la extension de nuestra GDT
32
33 ;Descriptor GDT
34 gdt_descriptor:
35     dw gdt_end - gdt_start - 1      ;Extension de nuestro GDT,
          siempre uno menos
36     dw gdt_start                    ;Direccion de Inicio de
          nuestro GDT
37
38 CODE_SEG equ gdt_code - gdt_start
39 DATA_SEG equ gdt_data - gdt_start
40
41 lgdt [gdt_descriptor]
42 mov eax, cr0     ;Para hacer el cambio al modo protegido
43 or eax, 0x1      ;El primer bit de cr0, un registro de control
44 mov cr0, eax     ;Actualizar el registro de control
```

Luego de haber actualizado “cr0” el CPU ya se encuentra en modo protegido (32 bits).

3. Ejemplos de aplicación

3.1. Problema 1

Crear un **boot loader** que tenga como función calcular la media aritmética de 3 números menores a 10 los cuales se declararán como constantes.

3.2. Solución

Debido a que el enunciado pide codificar un **boot loader**, esto nos direcciona a brindar archivo de sistema con extensión *bin*. Siendo así, la solución se hará en **Modo real**. En la Lista 12 se observa la solución del ejercicio.

Lista 13: Ejemplo media aritmética como boot loader bits

```
;Calculo de la media de 3 numeros de una cifra en 16 bits
bits 16
org 0x7c00
MSG_MEDIA: db "Media: ", 0
vNum1 equ 3
vNum2 equ 7
vNum3 equ 8
boot:
        mov ax, 0
        mov bx, MSG_MEDIA
        call print_string
        mov ax, vNum1
        add ax, vNum2
        add ax, vNum3
        mov bx, 3
        div bx
        mov ah,0x0e
        mov dx, ax
        mov al, dl
        add al, '0'
        int 0x10
        jmp $
print_string:
        pusha
        mov ah,0x0e
        lazo2:
        mov al, [bx]
        cmp al,0
        je return
        int 0x10
        inc bx
        jmp lazo2
        return:
        popa
        ret
times 510 - ($-$$) db 0
```

```
37 || dw 0xaa55
```

Para compilar y ejecutar el programa seguir las instrucciones(nombre de archivo: media16.asm):

- nasm -f bin media16.asm -o media16.bin
- qemu-system-x86_64 -fda media16.bin

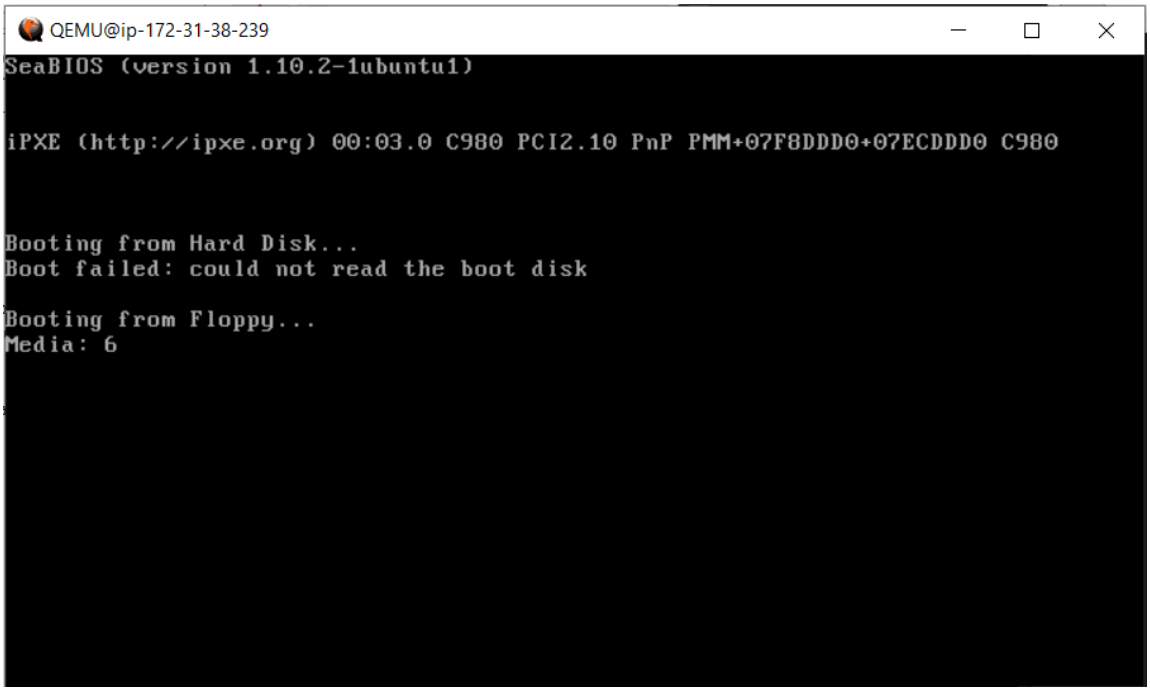

Fig. 4: Resultado del bootloader en QEMU

3.3. Problema 2

Codificar un programa en lenguaje ensamblador que permita calcular la media aritmética de 3 números menores a 10.

3.4. Solución

En este caso, el problema es bastante explícito al pedir un programa en lenguaje ensamblador, a diferencia del problema previo que pedía un **bootloader**. Para ello, se codificará la solución con los segmentos y variables previamente definidos. Así mismo, si bien es posible generar la impresión por medio de interrupciones, una de las ventajas de compiladores como GCC es que permite utilizar funciones en C como *printf*[4].

La solución del problema se muestra en la Lista 13.

[4]Instalar la siguiente libreria en linux: sudo apt-get install gcc-multilib g++-multilib

Lista 14: Ejemplo media aritmética 32bits

```
segment .data
        SYS_EXIT  equ 1
        SYS_READ  equ 3
        SYS_WRITE equ 4
        STDIN     equ 0
        STDOUT    equ 1
        num1 dd 8
        num2 dd 4
        num3 dd 6
        message db "La media es: %d", 10, 0
segment .bss
segment .text
        global main
        extern printf
main:
;SIEMPRE LIMPIAR
        mov eax, 0
        mov ebx, 0
    mov ecx, 0
    mov edx, 0
;-------------------------
        mov eax,[num1]
        mov ebx,[num2]
        add eax, ebx
        mov ebx, [num3]
        add eax, ebx
        mov ebx, 3
        idiv ebx
        push eax
        push message
        call printf
        mov eax, SYS_EXIT
        xor ebx, ebx
        int 0x80
```

Para ejecutar su programa, debe utilizar tanto nasm como gcc de la siguiente manera:

- nasm -f elf32 -o media32.o media32.asm
- gcc -m32 -o media32 media32.o
- ./media32

```
ubuntu@ip-172-31-38-239:~/2020-1/ASM32$ nasm -f elf32 -o media32.o media32.asm
ubuntu@ip-172-31-38-239:~/2020-1/ASM32$ gcc -m32 -o media32 media32.o
ubuntu@ip-172-31-38-239:~/2020-1/ASM32$ ./media32
La media es: 6
```

Fig. 5: Resultado de ejecutar el código en un terminal de Linux

Nota: Si aún presenta dificultades para aprender la sintaxis del lenguaje ensamblador se le recomienda revisar bibliografía adicional (`https://asmtutor.com/`) [1].

Referencias

[1] Learn Assembly Language. Nasm assembly language tutorials.

[2] Nick Blundell. *Writing a Simple Operating System — from Scratch*. School of Computer Science, University of Birmingham, UK, 2010.