



## Sistemas Básicos

GONGORA TUN  
ANDREA ELIZABETH

*Proyecto Final*

# UNIVERSIDAD AUTÓNOMA DE QUERÉTARO FACULTAD DE INFORMÁTICA

Costas Rueda Juan Pablo

Javier Valencia Vázquez

Jesús Alberto Vázquez Rioja

Juan Carlos Salas Gallegos

Diego Emmanuel López Palacios

Grupo 30

28/11/2023



## Introducción

*Explicar qué es un lenguaje de alto nivel y qué es un lenguaje de bajo nivel. Y cuál es la diferencia entre estos.*

La programación es la escritura de una serie de instrucciones que le indican a un computador cómo realizar alguna operación. A los individuos que escriben estas instrucciones se les conoce como programadores, este acto se debe de hacer con base en normas establecidas y las necesidades de la operación a realizar.

Además de basarse en normas, existen 2 categorías o niveles de programación:

- **Alto Nivel**
  - Consiste en un lenguaje de programación más apegado al lenguaje humano, su ejecución no depende del sistema en que está. Dado a que es más abstracto requiere un compilador para traducir el código a una versión más entendible por la computadora. Algunos ejemplos son: Java, JavaScript, Python, C++, etc.
- **Bajo Nivel**
  - Es más apegado al lenguaje máquina, comparación con el de alto nivel. Su nivel lo acerca más al hardware y no requieren de un compilador para que la computadora entienda las instrucciones. Algunos ejemplos son: Assembly y Fortran, lenguajes como C y C++ pueden incluir ensamblador en su código.

*Mencionar que es un compilador, cómo se diseña y cómo se crea un compilador.*

Un compilador es un programa de computadora que convierte código de alto nivel en código ejecutable, por ejemplo, código binario o máquina.

Un compilador tiene varias funciones:

- **Análisis léxico:** El compilador divide el código en tokens, que representan factores clave para hacer un programa, diferenciar entre valores, variables, palabras clave, etc.
- **Análisis sintáctico:** Cada lenguaje de programación tiene su gramática, y el compilador se encarga de verificar que el código fuente la respete.
- **Análisis semántico:** El compilador verifica que las instrucciones tengan sentido, por ejemplo, un int no puede guardar un valor flotante, por lo que el compilador puede arrojar un error o truncar el valor según como se haya diseñado el lenguaje.
- **Optimización de código y código intermedio:** El compilador puede crear un código intermedio con optimizaciones, por ejemplo, eliminando código que no se usa pero se declara, entre otros.
- **Generación de código objeto:** El compilador genera el objeto que es ejecutable sobre la computadora.
- **Vinculación y carga:** El compilador vincula el uso que se le da a bibliotecas para que sean usados en el código objeto final.

Una invocación de compilador típica ejecuta algunas o todas estas actividades en secuencia. Para las optimizaciones de tiempo de enlace, algunas actividades se ejecutan más de una vez durante una compilación.

1. Preproceso de archivos de origen
2. Compilación, que puede constar de las fases siguientes, en función de las opciones de compilador que se especifiquen:
  - a. Análisis frontal y análisis semántico
  - b. Optimización de alto nivel
  - c. Optimización de bajo nivel
  - d. Registrar asignación
  - e. Montaje final
3. Ensamblaje de los archivos de ensamblaje (.s) y los archivos de ensamblador no preprocesados (.S) después de que se hayan preprocesado
4. Enlace de objeto para crear una aplicación ejecutable

## CREACIÓN DE UN COMPILADOR

La creación de un compilador es un proceso complejo el cual involucra distintas etapas, desde la definición de la gramática del lenguaje hasta la generación de código ejecutable.

En la creación de un compilador se sigue un proceso estructurado y multifacético que implica varias etapas(ya mencionadas) para transformar el código en lenguaje de alto nivel en código ejecutable por la máquina.

Es necesario realizar pruebas para garantizar que el compilador maneje correctamente el lenguaje y produzca código sin errores, además es importante la documentación desde la gramática del lenguaje hasta los detalles de su implementación para facilitar el mantenimiento, resolución de problemas y entender el compilador en la parte de los desarrolladores.

*Describir de manera general su algoritmo propuesto para probar las instrucciones del macro ensamblador.*

Será una serie de operaciones matemáticas simples ejecutadas en bucles anidados. Se utilizarán los siguientes conceptos:

- Suma e iteración:
  - El bucle for externo se encarga de la iteración (repetición) y suma de valores a una variable (num).
- Multiplicación y bucle While:
  - El bucle “while” interno se encarga de la multiplicación sucesiva de la variable num por los valores de “j”.
- Estructuras de control:
  - El uso de bucles “for” y “while” son estructuras de control de flujo comunes en algoritmos.
- Operaciones matemáticas:
  - El código realiza operaciones básicas de suma y multiplicación.

- Impresión de valores:
  - El código imprime el valor de num después de realizar, tanto sumas como multiplicaciones.

*Escribir el algoritmo en un lenguaje de alto nivel.*

**Código en C.**

```
#include <stdio.h>

int main() {
    // Declaración e inicialización de la variable "num" con el valor 0.
    int num = 0;

    // Bucle "for" que itera cinco veces, con la variable de control "i" incrementándose de 0 a 4.
    for (int i = 0; i < 5; i++) {
        // Incrementa "num" por el valor de "i".
        num += i;

        // Declaración e inicialización de la variable "j" con el valor 1.
        int j = 1;
        // Imprime la suma actual con el mensaje "Suma: ".
        printf("Suma: %d\n", num);

        // Bucle "while" que se ejecuta mientras "j" sea menor o igual a 5.
        while (j <= 5) {
            // Multiplica "num" por el valor de "j".
            num *= j;
            // Imprime el producto actual con el mensaje "Multiplicacion: ".
            printf("Multiplicacion: %d\n", num);
            // Incrementa "j" en 1.
            j++;
        }
    }

    // El programa finaliza.
    return 0;
}
```

*Ejecución del código.*

```
Suma: 0
Multiplicacion: 0
Multiplicacion: 0
Multiplicacion: 0
Multiplicacion: 0
Multiplicacion: 0
Suma: 1
Multiplicacion: 1
Multiplicacion: 2
Multiplicacion: 6
Multiplicacion: 24
Multiplicacion: 120
Suma: 122
Multiplicacion: 122
Multiplicacion: 244
Multiplicacion: 732
Multiplicacion: 2928
Multiplicacion: 14640
Suma: 14643
Multiplicacion: 14643
Multiplicacion: 29286
Multiplicacion: 87858
Multiplicacion: 351432
Multiplicacion: 1757160
Suma: 1757164
Multiplicacion: 1757164
Multiplicacion: 3514328
Multiplicacion: 10542984
Multiplicacion: 42171936
Multiplicacion: 210859680
```

Desarrollo

*Escribir el algoritmo en un lenguaje de bajo nivel (Microsoft Macro Assembler x86).*




## Ciclo for




```
1  ;Codigo MASM - Codigo for loop
2
3  ; Protocolos de iniciacion
4  .386
5  .model flat, stdcall
6  .stack 4096
7  ExitProcess proto, dwExitCode: dword
8
9  ; Creacion de variables
10 .data
11
12 addition DWORD ?
13
14 ; Iniciacion del programa
15 .code
16 main proc
17     mov eax, 10      ; Valor inicial
18     mov ecx, 3       ; Numero de loops
19
20 ; Se inicia el ciclo for
21 for_loop:
22     add eax, 5       ; Sumar eax 5
23     loop for_loop   ; Checa el registro ecx, le resta un valor y se regresa a for_loop
24                     ; En caso de que ecx sea 0, se termina el ciclo
25
26     mov addition, eax ; Guarda el valor final en addition
27
28     invoke ExitProcess, 0 ; Se termina el proceso
29
30 main endp
31 end main
```




*Describir cada paso de instrucción de ensamblador.*




1. **mov ecx, 3:** Inicializa el contador del **loop** con un valor de 3.
2. **for\_loop::** Marca el inicio del ciclo, el código dentro de este se ejecutará hasta que el ciclo termine,
3. **add eax, 5:** Suma el valor en el registro **eax** 5 valores.
4. **loop for\_loop:** En automatico, busca el registro **ecx** y le resta un valor. En caso de que el valor termine siendo 0, este termina el ciclo y ejecuta el código después del ciclo.
5. **mov product, eax:** Almacena el valor final en la variable **addition**.

Señalar el incremento, decremento o resultado de sus variables en memoria o registro.

Name	Value	Type
 eax	10	unsigned int
 ecx	3	unsigned int
 addition	0	unsigned long
<i>Add item to watch</i>		

Name	Value	Type
 eax	15	unsigned int
 ecx	2	unsigned int
 addition	0	unsigned long
<i>Add item to watch</i>		

Name	Value	Type
 eax	20	unsigned int
 ecx	1	unsigned int
 addition	0	unsigned long
<i>Add item to watch</i>		

Name	Value	Type
 eax	25	unsigned int
 ecx	0	unsigned int
 addition	25	unsigned long
<i>Add item to watch</i>		

## Ciclo while

```
1 ;Codigo MASM - Codigo While loop
2
3 ; Protocolos de iniciacion
4 .386
5 .model flat, stdcall
6 .stack 4096
7 ExitProcess proto, dwExitCode: dword
8
9 ; Creacion de variables
10 .data
11
12 product DWORD ?
13 counter DWORD 3 ; Numero de loops en contador
14
15 ; Iniciacion del programa
16 .code
17 main proc
18     mov eax, 5      ; Valor inicial
19
20 while_loop: ; Empieza el ciclo while
21     cmp counter, 0  ; Comparar el contador con 0
22     je end_while   ; ir a end_while si el contador es 0
23
24     imul eax, 2     ; Multiplicar eax por 2
25     dec counter     ; Restar 1 al contador
26     jmp while_loop  ; Ir a while_loop
27
28 end_while: ; Se ejecuta cuando el contador llega a 0
29     mov product, eax ; Almacenar el valor final en product
30
31     invoke ExitProcess, 0 ; Se termina el proceso
32
33 main endp
34 end main
```




*Describir cada paso de instrucción de ensamblador.*




1. **mov eax, 5**: Inicializa **eax** con un valor de 5
2. **while\_loop::**: Marca el inicio del ciclo while
3. **cmp counter, 0**: Compara el valor de **counter** con 0.
4. **je end\_while**: Salta a **end\_while** si la comparación de **0** con **counter** es igual.









5. **imul eax, 2**: Multiplica el valor de **eax** por **2**.
6. **dec counter**: Decrementa el valor de **counter** **-1**.
7. **jmp while\_loop**: Sin condición, salta de regreso a **while\_loop**, creando un bucle hasta ser interrumpido.
8. **end\_while**:: Marca el final del ciclo.
9. **mov product, eax**: Se guarda el resultado final en **product**.
10. **invoke ExitProcess, 0**: Se finaliza el proceso.




*Señalar el incremento, decremento o resultado de sus variables en memoria o registros.*

Name	Value	Type
 <b>eax</b>	5	unsigned int
 <b>counter</b>	3	unsigned long
 <b>product</b>	0	unsigned long
Add item to watch		

Name	Value	Type
 <b>eax</b>	10	unsigned int
 <b>counter</b>	2	unsigned long
 <b>product</b>	0	unsigned long
Add item to watch		

Name	Value	Type
 <b>eax</b>	20	unsigned int
 <b>counter</b>	1	unsigned long
 <b>product</b>	0	unsigned long
Add item to watch		

Name	Value	Type
 <b>eax</b>	40	unsigned int
 <b>counter</b>	0	unsigned long
 <b>product</b>	0	unsigned long
Add item to watch		

Name	Value	Type
 <b>eax</b>	40	unsigned int
 <b>counter</b>	0	unsigned long
 <b>product</b>	40	unsigned long
Add item to watch		

## Conclusión

*Describir el nivel de dificultad que obtuvieron al escribir en un lenguaje de alto nivel y en un lenguaje de bajo nivel.*

Conociendo el lenguaje de programación, escribir código en lenguajes de alto nivel es muy sencillo y fácil de entender, las palabras clave y los símbolos utilizados por los lenguajes de alto nivel permiten una fácil comprensión de lo que el programa va a hacer.

Por el lado contrario, no estamos muy acostumbrados a trabajar con lenguajes de bajo nivel, así que nos encontramos con varios retos a la hora de trabajar con

Microsoft Macro Assembler. El primero fue preparar el entorno, el cual no era complicado, pero se llegaron a saltar pasos y no se podía identificar con facilidad lo que faltaba. Otro de los retos es la falta de contexto a la hora de ejecutar un error, no te indica que fue lo que estaba mal escrito en el código, y por último es trabajar con los registros y la sintaxis de MASM, nosotros solemos trabajar con puras variables y definimos nuestra propia lógica, pero en MASM, mucho de los registros del procesador tienen un propósito específico y en algunos casos la lógica de los operadores ya está definida, como la operación loop que busca el registro ecx, le resta un valor y chequea que sea 0 para detener el loop.

*Describir por qué creen que el LLM resultó con modificaciones en memoria, registro u otras instrucciones.*

Porque la arquitectura x86 de los procesadores ya tiene lógica predeterminada que se encarga de alterar la memoria, establecer valores por default y ejecutar código. Como es el caso con los operadores loop, add, jmp, je y imul, en este caso.

Esto permite que el mismo código se pueda ejecutar en múltiples computadoras mientras tengan la arquitectura de procesador x86.