Lenguaje Ensamblador

Estrategia: Es una materia muy teórica en los exámenes por lo que una guía de estudio con la mayor información posible vista en el parcial, será lo mejor. Las tareas pueden rescatar puntos así como condenar la calificación

▼ **Primer Parcial**

▼ **09-08-2021**

No hubo clase

▼ **10-08-2021**

Consideraciones del curso

Consideraciones del Aula virtual y Teams

- Todos los archivos de texto se deben entregar en formato pdf
- Cuando se entrega una práctica con mas archivos se adjunta en un zip.
- Las actividades en general se deben de enviar en tiempo y forma (menos calificación)
- Se debe tomar el debido respeto en las conferencias virtuales.
- Actualizar imagen de perfil en las dos plataformas.
- Cualquier duda pueden enviar WhatsApp o correo electrónico.
- 4493842243, cristian.mejia@edu.uaa.mx
- Se seguirá la evaluación del programa de la materia.
- Formato para entrega de reportes de investigación: Portada, Introducción, Contenido, Conclusiones, Referencias.
- Respetar formato (justificado, misma fuente, tamaño, etc.)

Formato digital y TODO debe llevar portada

▼ **11-08-2021**

No hubo clase

▼ **12-08-2021**

Todos los archivos se deben entregar en PDF

Si se piden archivos aparte, se debe subir un archivo ZIP

Respeto a conferencias

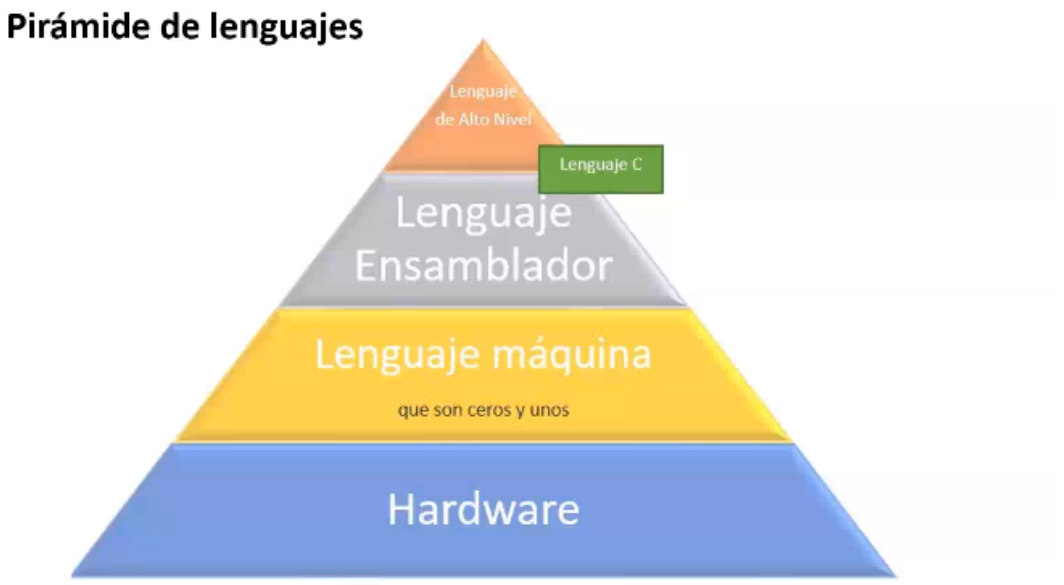
Criterios de evaluación

Criterio de evaluación por parcial:
- Examen parcial 50%
- Tareas 25%
- Prácticas 25%

Criterio de evaluación semestral:
- Primer parcial 20%
- Segundo parcial 20%
- Tercer parcial 30%
- Proyecto final 30%

Las entregas son individuales al menos que se especifique
Las entregas son en linea en la plataforma Aula virtual
Aunque las actividades sean en equipo todos los integrantes deben subir el reporte.

Pirámide de lenguajes



Están Intel y AMD que comparten un set de instrucciones para ser compatibles

Lenguaje Ensamblador

Lenguaje de programación de bajo nivel

Consiste en un conjunto de mnemónicos que representan instruciones básicas para microprocesadores y mocrocontroladores

Es una representación simbólica de los códigos de máquina binarios y otras constantes necesarias para programar una arquitectura de procesador y constituye la representación más directa del código máquina específico para cada arquitectura legible por un programador

Mnemónico

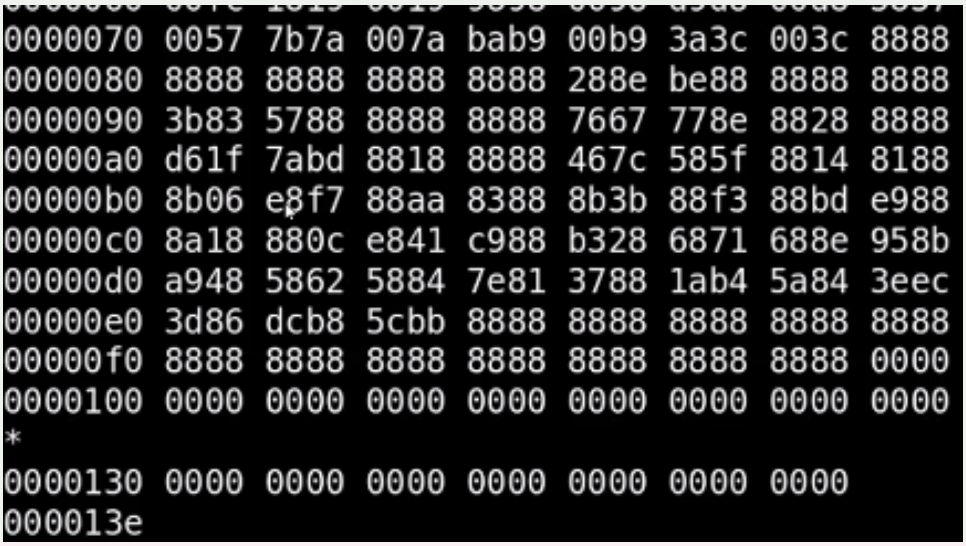
Es una palabra que sustituye a un código de operación en lenguaje máquina

Simbolizan los pasos de procesamiento, los registro de procesador, las posiciones de memoria y otras características

Código binario	Código hexadecimal	Código ASCII	Nemónico	Acción
101000	50	P	ADD	Suma al acumulador
10110001	B1	±	SUB	Resta al acumulador
010000xx	40	@	INC	Incrementa al registro
010010xx	12	◆	DEC	Decrementa al registro
00110010	32	◆	Jxx	Salto al registro o xxx

Lenguaje máquina

Es un sistema de códigos directamente interpretable por hardware de una computadora o sistema electrónico



▼ **16-08-2021**

Historia de los microprocesadores INTEL

Intel fue la primera compañía de microprocesadores del mundo fundada en 1968 por Gordon E. Moore y Robert Moyce llamando Integrated Electronic

Fabricaba memorias para saltar a los microprocesadores. Hasta los 70's fueron líderes gracias al competitivo mercado de memorias

La ley de Moore

Expresa que aproximadamente cada dos años se duplica el número de transistores en un microprocesador

El 15 de noviembre de 1971 lanzaron el primer microprocesador conocido como el Intel 4004 facilitando su diseño de calculadora

En 1972 anunciaba Intel una mejor versión de su procesador: el 8008. Podía acceder a más memoria y procesar 8 bits. La velocidad de su reloj alcanzaba los 740 KHz. EL 8080 fue su sucesor

El 8086 y el 8088 son los primeros microprocesadores de 16 bits de Intel y los primeros miembros de la arquitectura x86

El 8086 comenzó a desarrollarse en 1976 y lanzado en 1978. El 8088 en 1979

Luego salieron los 80186 y 80188 que son dos microprocesadores desarrollados alrededor de 1981. Los i80186 e i80188 tiene un bus externo de 16 bits, mientras que el i80188 lo tiene de 8 bits como el i8088 para hacerlo más económico

En 1982 salió el procesador 286 o mejor conocido como 80286 que reconocía y ulitizaba software para los procesadores anteriores. Era un procesador de 16 bits y 134,000 transistores capaz de direccionar hasta 16 MB de RAM con soporte de memoria física incrementado, con un chip capaz de trabajar con memoria virtual, permitiendo una gran capacidad de expansión

El sucesor fue el Intel 80386 (i386, 386) es un microprocesador CISC con arquitectura x86. Durante su diseño se le llamó P3 por ser prototipo de tercer generación. Arquitectura de 32 bits

Luego salió el Intel 486DX (i486). Fue el primero procesador con más de 1 millón de transistores de 32 bits y funcionaba con relojes de hasta 100 MHz

El siguiente fue el Intel Pentium con más de 3 millones de transistores. Hubo algunos problemas legales para llamarlo 80586

El siguiente fue el Intel Pentium Pro, era P6. Era un chip RISC con emulador de hardware 486 que funcionaba a 200 MHz

El siguiente fue el Intel Pentium 2 en 1997. Mejoraba el rendimiento en ejecución de código de 16 bits pero nada tan eficiente

Luego salió el Intel Celeron de bajo costo para acceder a mercados cerrados a los Pentium de mayor rendimiento y precio basado en el Pentium 2

Después salió el Pentium 3 de arquitectura i686 en 1999

Después salió el Pentium 4 con arquitecturas x86 con GHz lanzado el 20 de noviembre del 2000

Luego salió el Pentium Inside en 2003 para laptops

Luego salió el Pentium D en 2005

Luego siguió el Intel Core 2 de 64 bits que incluí un procesador de un núcleo, doble núcleo y cuádruple núcleo basado en la Core de Intel (2006). Los Intel Core son los primeros procesadores de 64 bits. Por eso son x86-64

Después salió en 2008 el Intel Atom de 64 bits

Salieron después los Intel Core i3, i5 e i7 consecutivamente en el 2014, 2016 y 2017

Como en 2019 salió el i9

Más información:

https://www.intel.la/content/www/xl/es/products/details/processors/core.html

▼ **17-08-2021**

Arquitectura de una computadora

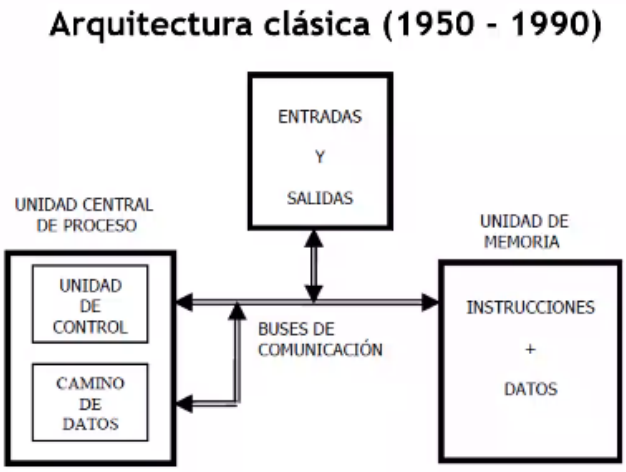
Integración de su estructura física con la lógica

Enfatiza sus elementos de ingeniería y ciencias computacionales

Sus fases son:

- Definición de las necesidades que pretende cubrir el computador.
- Planificación general del computador.
- Diseño del computador y sus componentes.
- Análisis del sistema obtenido.
- Especificación del sistema: características del sistema, de sus componentes y de las instrucciones ejecutables.

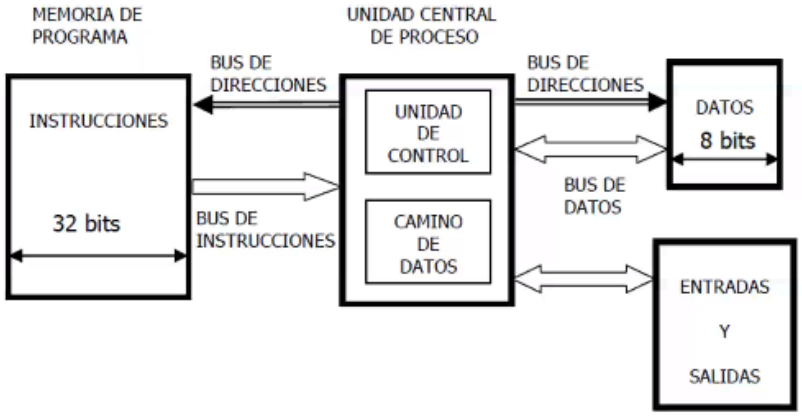
Arquitectura clásica (Arquitectura de Von Neumann)



Fué definida por John Von Neumann, es una arquitectura en la cual la CPU (Unidad Central de proceso) está conectada a una única memoria donde se guardan conjuntamente instrucciones (programas) y datos (con los cuales operan estos programas). Además existe un módulo de entradas y salidas para permitir la comunicación de la máquina con los periféricos extremos que maneja el usuario.

Arquitectura moderna (Arquitectura Harvard)

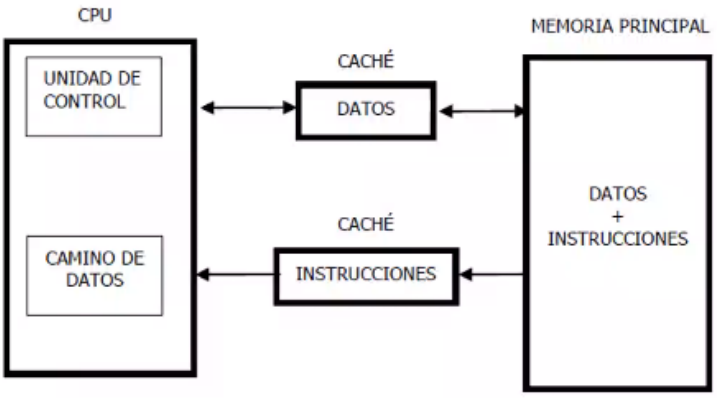
Arquitectura moderna (1990 - hoy).



La arquitectura Harvard, que está especialmente diseñada para atacar las debilidades de la arquitectura Von Neumann, la solución, conceptualmente, es harto sencilla, se construye un procesador que está unido a dos tipos de memoria diferentes por medio de dos buses independientes.

Arquitectura actual Pentium

Arquitectura actuales Pentium.



Una forma de potenciar el aislamiento entre las instrucciones y los datos es la incorporación de memorias caché ultrarápidas, que como sucede en los últimos modelos Pentium, una se encarga de guardar los datos que va a precisar la CPU y otra las instrucciones.

ARM se especializa en dispositivos móviles

▼ **18-08-2021**

Taxonomía de Flynn

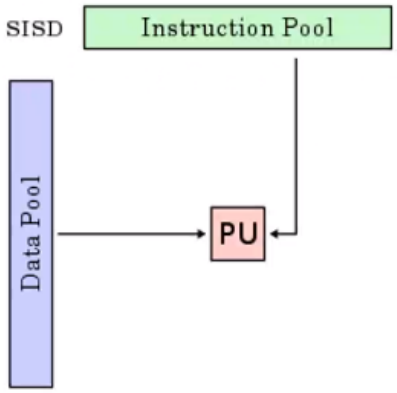
Clasificación de arquitecturas de computadoras por Michael J. Flynn de la Universidad de Stanford

Clasificaciones:

- Una instrucción, un dato (SISD)

SISD (Single Instruction Single Data)

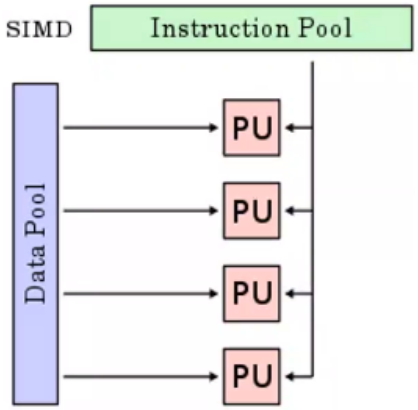
Esta clasificación se refiere a las computadoras tradicionales y secuenciales en las cuales una instrucción a la vez se ejecuta sobre un único dato cada ciclo de reloj. Los datos en cuestión se almacenan en una única memoria en la cual se usan técnicas como la segmentación para evitar errores de fragmentación interna. Un ejemplo sencillo de estas computadoras son los antiguos mainframe basados en la arquitectura de Von-Neumann.



- Una instrucción, múltiples datos (SIMD)

SIMD (Single Instruction Multiple Data)

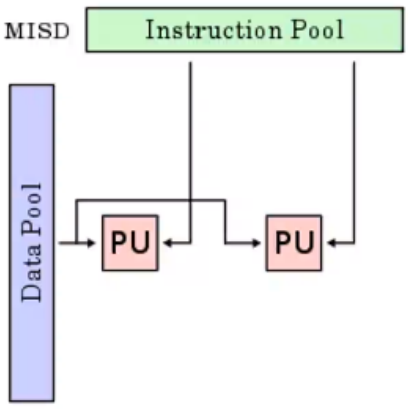
Representa la ejecución de una misma instrucción sobre un conjunto de datos. La misma es comúnmente vista en ciclos de programación que ejecutan una misma instrucción una y otra vez sobre datos de un arreglo o conjunto de datos. En la arquitectura SIMD estos datos son procesados por múltiples CPU que ejecutan la misma instrucción sobre una parte del conjunto o arreglo, cada uno, hasta llegar a procesar la totalidad de los mismos.



- Múltiples instrucciones, un dato (MISD)

MISD (Multiple Instruction Single Data)

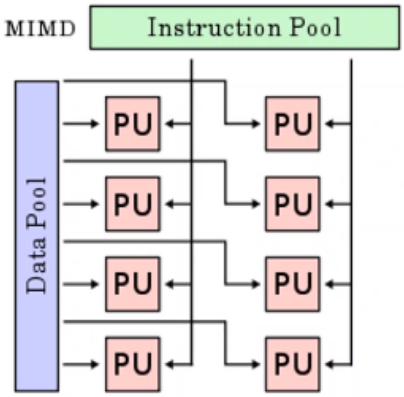
Se refiere a múltiples instrucciones ejecutándose sobre un único dato. Comúnmente se considera esta arquitectura poco práctica ya que en tiempo de ejecución la efectividad del paralelismo requiere un múltiple flujo de datos y, además, el acceso concurrente a un mismo dato en memoria puede ocasionar que un CPU tenga que esperar a que el recurso (dato) esté disponible para poder acceder a él.



- Múltiples instrucciones, múltiples datos (MIMD)

MIMD (Multiple Instruction Multiple Data)

Esta arquitectura representa a un conjunto de instrucciones que se ejecutan sobre un conjunto múltiple de datos. La misma es muy usada hoy en día para explotar el paralelismo ya sea con memoria distribuida y memoria compartida o híbridos como los clústers de computadoras. Muchos multiprocesadores modernos (como los de la tecnología Core i de Intel) entran en esta clasificación.



▼ **19-08-2021**

No hubo clase

▼ **23-08-2021**

Investigación CISC vs RISC

▼ **24-08-2021**

CISC y RISC

RISC

Reduced Instruction Set Computing

Tipo de diseño de CPU utilizado en microprocesadores o microcontroladores para celulares o tablets o las características:

- Instrucciones de tamaño fijo y presentadas en un reducido número de formato
- Sólo las instrucciones de carga y almacenamiento acceden a la memoria de datos

- Estos procesadores suelen disponer de muchos registros de propósito general

Cada instrucción puede ser ejecutada en un solo ciclo de CPU

Usa un sistema de direcciones no destructivas en RAM

CISC

Complex Instruction Set Computing

Más versátil

Conjunto de instrucciones amplio y permite operaciones complejas

Ejecuta varios centenares de instrucciones complejas diferentes

Mejora la compactación de código

Facilita la depuración de errores

CISC	RISC
Muchas instrucciones	Pocas Instrucciones
Instrucciones tienen longitud variable	Instrucciones tienen longitud fija
Muchas instrucciones puede acceder a memoria	Pocas instrucciones puede acceder a memoria (e.j. load y store)
En muchas instrucciones el procesador puede leer y escribir en memoria en la misma instrucción	En ninguna instrucción el procesador puede leer y escribir en memoria.
Pocos y más especializados registros (ej. Registros de datos, de instrucciones)	Muchos registros de propósito general
Muchos tipos de modos de direccionamiento	Limitado número de modos de direccionamiento

▼ **25-08-2021**

Tarea de x86

▼ **26-08-2021**

No hubo clase

▼ **30-08-2021**

Banderas del Registro de Estado

Registro de estado (Status register)

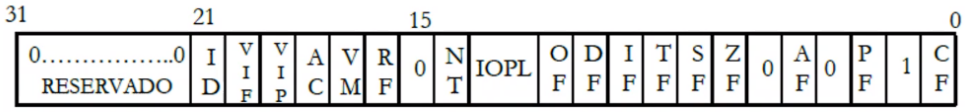
El registro de estado es un registro de hardware que contiene información sobre el estado del procesador . Los bits individuales se leen o escriben mediante las instrucciones del código de máquina que se ejecutan en el procesador. El registro de estado permite que una instrucción actúe en función del resultado de una instrucción anterior.

Normalmente, los indicadores (FLAGS) en el registro de estado se modifican como efectos de operaciones aritméticas y de manipulación de bits. Por ejemplo, un bit Z puede establecerse si el resultado de la operación es cero y borrarse si es distinto de cero. Otras clases de instrucciones también pueden modificar las banderas para indicar el estado.

guirre

Registro de estado (Status register)

En la familia x86, este registro también conocido como el registro EFLAGS consta de 32 bits de los cuales la mayoría son indicadores de estado, controlados por la ALU (acarreo, paridad, acarreo auxiliar, cero, signo y sobrepasamiento), actuando los restantes como indicadores del sistema, ligados al mecanismo de protección y a otros recursos de que dispone el sistema de explotación cuya misión será mejor interpretada a medida que se profundice en el estudio del procesador



Carry Flag

CF: Señalizador de acarreo en el más significativo. Si trabajamos en modo real es en el bit 16 y si trabajamos en modo protegido es en el bit 32.

1: Si hay acarreo en las operaciones de suma aritmética.
0: En las restas cuando hay acarreo en el bit de más peso. También se utiliza este bit en operaciones de desplazamiento y rotación.

Parity Flag

PF: Bit de paridad impar.

1: Para generar la paridad impar con los bits que conforman el resultado de una operación
0: En caso de no producirse paridad impar.

Auxiliar Flag

AF: Señalizador de acarreo auxiliar (o intermedio).

1: Si ha habido acarreo en el bit 3 del resultado (contando sería el 4º bit). Se utiliza en las operaciones BCD.

0: En caso de no haber acarreo auxiliar.

Zero flag

ZF: Señalizador de cero.

1: Si todos los bits del resultados son cero. Es muy útil al comparar dos registros, ya que si tienen el mismo contenido ZF pasa a valer 1.

0: En caso de que el resultado no sea 0

Sign flag

SF: Señalizador de signo.

1: Si el bit de más peso del resultado de la operación es 1.

0: Si el bit de mas peso del resultado de la operación en 0.

Trap flag

TF: Excepción al terminar la ejecución de la instrucción.

1: Provoca una excepción al completarse la ejecución de la instrucción en curso. Posibilita la depuración de los programas al permitirse la ejecución paso a paso, es decir instrucción por instrucción.

0: No hay excepción
Si un programa de aplicación dispone de un flag TF utilizando una instrucción POPF, POPFD, o IRET, se genera una “limpieza de excepción” después de la instrucción que sigue.

Interrupt enable flag

IF: Flag de habilitación de interrupciones.

1: Permite el reconocimiento de las peticiones de interrupción.

0: Prohíbe el reconocimiento de la interrupción externa y no la atiende.

Direction flag

DF: Flag de dirección de exploración de las cadenas de caracteres o strings.

1: Postdecremento automático de los registros ESI, EDI (registros de propósito general), que direccional la cadena.

0: Postincremento automático en ESI, EDI.

Overflow flag

OF: Flag de sobrepasamiento

1: En operaciones con números enteros con signo se activa si el resultado es muy grande (es positivo) o muy pequeño (si es negativo). Indica resultados erróneos por ejemplo si existe acarreo en el penúltimo bit.

0: Si no existe sobrepasamiento

I/O privilege level

IOPL: Nivel de privilegio de las entradas y salidas (input/output).

Para proteger el código de multitarea, algunas tareas necesitaban privilegios para acceder a los puertos de E / S, mientras que otras tenían que dejar de acceder a ellos. Intel introdujo una escala de privilegios de cuatro niveles, con 00 2 siendo los más privilegiados y 11 2 siendo los menos. Si IOPL era menor que el Nivel de privilegio actual, cualquier intento de acceder a los puertos de E / S, o habilitar o deshabilitar interrupciones, causaría un error de protección general en su lugar.

Nesked

Resume flag

RF: Flag de reanudación. Su activación provoca la ejecución de la siguiente instrucción cuando se produce un fallo de depuración en una instrucción, es decir, se ignora el fallo producido en la depuración.

1: Se ignoran los puntos de parada

0: No se ignoran los puntos de parada

Virtual 8086 mode

VM: Modo virtual – 86. Trabajando en modo protegido, se permite que algunas tareas trabajen en Modo Real.

1: Estando el procesador en Modo Protegido, se pasa a Modo Virtual 86

0: No se hace nada

Alignment check

AC: Bit de chequeo de alineamiento.

1: Se produce una excepción para alinear una palabra
0: No hay excepción

Las direcciones de palabra de palabras de 2 bytes deben ser múltiplos de 2, las de 4 bytes deben serlo de 4 y las de 8 bytes, múltiplos de 8. Si un programa de aplicación con el mínimo nivel de privilegio, es decir, nivel de privilegio 3, tiene una palabra desalineada y este flag AC = 1, se produce una excepción en concreto la excepción número 17. Las referencias de memoria que por defecto tienen un nivel de privilegio 0 no generan esta excepción, incluso cuando son ejecutadas en modo usuario.

Virtual interrupt flag

VIF: Interrupción virtual. Es un flag equivalente a IF cuando se trabaja en modo virtual 8086. Este flag se utiliza con el flag VIP. El procesador sólo reconoce el flan VIF cuando bien el flag VME o el PVI en el registro de control CR4 están activados y el IOPL es menor que 3.

Virtual interrupt pending

VIP: Interrupción (mascarable) virtual pendiente. Trabaja en combinación con el flag VIF para que cada tarea en modo virtual disponga de su flag IF. Así se aceleran notablemente las interrupciones y las instrucciones CLI y STI no provocan excepción. Se activa por software para indicar que una interrupción está pendiente. El procesador lee este flag pero nunca lo modifica. El procesador sólo reconoce el flag VIP cuando el flag VME o el PVI en el registro de control CR4 están activados y el IOPL es menor que 3.

1: Interrupción pendiente
0: No hay interrupción pendiente

Able to use CPUID instruction

ID: Bit de identificación. El flag ID indica si el Pentium soporta la instrucción CPUID que sirve para su identificación. Mediante CPUID se informa sobre las características más importantes del procesador. Le informa al software acerca del modelo de microprocesador en que se está ejecutando.

▼ **31-08-2021**

Ensambladores

Programa que se encarga de traducir un lenguaje ensamblador a código máquina, ejecutable directamente por el microprocesador

Los ensambladores más populares son:

- GNU Assembler (gas)
- Flat assembler (FASM)
- Netwide Assembler (NASM)
- Microsoft Macro Assembler (MASM)
- High Level Assembly (HLA)
- RosASM

GNU Assembler (gas)

Proyecto GNU

Usado para compilar en Linux

Se puede acceder con el comando as desde el shell

Software libre y licenciado bajo GNU

Flat Assembler (FASM)

Flat assembler (FASM)

Flat assembler (FASM) es un ensamblador libre, multi-paso, con el estilo de la sintaxis de Intel que soporta las arquitecturas IA-32 y x86-64. Es notable por su velocidad rápida, optimizaciones de tamaño, portabilidad, poderosas capacidades de macro, y la comunidad del foro en línea. Sin embargo, casi no usa opciones en la línea de comandos. Hay disponibles archivos binarios y de código fuente para Linux, Windows (incluyendo un IDE de desarrollo), DOS, OpenBSD, MenuetOS, OctaOS, y DexOS. FASM contiene vínculos (bindings) para la GUI de Windows y OpenGL.

Netwide Assembler (NASM)

El Netwide Assembler o NASM, es un ensamblador libre para la plataforma Intel x86. Puede ser usado para escribir programas de 16-bit, 32-bit (IA-32) y 64-bit (x86-64). En el NASM, si se usan las bibliotecas correctas, los programas de 32 bits se pueden escribir de una manera tal para que sean portables entre cualquier sistema operativo x86 de 32 bits. El paquete también incluye un desensamblador, el NDISASM.

Microsoft Macro Assembler (MASM)

Fue producido originalmente por Microsoft para el trabajo de desarrollo en su sistema operativo MS-DOS, y fue durante cierto tiempo el ensamblador más popular disponible para ese sistema operativo. El MASM soportó una amplia variedad de facilidades para macros y programación estructurada, incluyendo construcciones de alto nivel para bucles, llamadas a procedimientos y alternación. Por lo tanto, MASM es un ejemplo de un ensamblador de alto nivel.

High Level Assembly (HLA)

Es un lenguaje ensamblador desarrollado por Randall Hyde, que puede usar construcciones de lenguaje de alto nivel para ayudar, en el lenguaje ensamblador x86, tanto a programadores principiantes como a desarrolladores avanzados por igual. El HLA soporta, en lenguaje ensamblador, tipos de datos avanzados y programación orientada a objetos.

RosASM

Es un lenguaje ensamblador x86 de 32 bits para Win32 lanzado bajo la licencia GNU General Public License de GNU. El nombre significa ReactOS ASseMbler, sin embargo, no está relacionado con ese proyecto. RosAsm es un IDE con integración completa del ensamblador, enlazador, editor de recursos, depurador y desensamblador. La sintaxis es inspirada en el NASM, otro ensamblador.

Tarea de todos los ensambladores que hay

Usaremos NASM

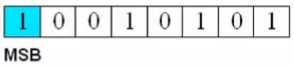
▼ **01-09-2021**

Enmascaramiento

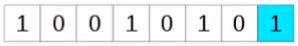
El enmascaramiento es un proceso en el que un número binario se convierte en otro número por medio de una operación lógica a nivel de bits. El número binario original es uno de los dos operandos en la operación. El segundo operando, llamado mascara, es un número binario especialmente escogido que realiza la transformación deseada.

Bit más significativo y menos significativo

Most Significant Bit (MSB), es el bit, que de acuerdo a su posición, tiene el mayor valor. Se hace referencia al MSB como el bit del extremo izquierdo.



Least Significant Bit (LSB),es la posición de bit en un número binario que tiene el menor valor. Se hace referencia al LSB como el bit del extremo derecho.



Operadores bit a bit (bitwise)

Son operaciones lógicas AND, OR, XOR y NOT sobre los bits individuales de los operandos.

NOT

Esta operación realiza una negación lógica en cada bit. Ejemplo:

NOT 10011
= 01100

AND

Esta operación realiza una AND lógica en cada par correspondiente de bits. Ejemplo:

```
      1101
AND 0111
   = 0101
```

El AND puede ser usado para filtrar determinados bits, permitiendo que unos bits pasen y los otros no. Ejemplo:

```
      0111
AND 0100  (máscara)
   = 0100
```

AND

Se puede usar para extraer bits de un byte. Ejemplo:

```
      1001 0011                    1001 0011
AND 1111 0000  (máscara)     AND 0000 1111  (máscara)
   = 1001 0000                  = 0000 0011
        LSB                          MSB
```

También, para apagar bits. Ejemplo:

```
      0111
AND 1101  (máscara)
   = 0101
```

OR

La operación bitwise OR realiza una OR lógica para cada par de bits.
Ejemplo:

```
    0111
OR 0110
  = 0111
```

Se utiliza para activar bits independiente mente si afectar al registro.
Ejemplo:

```
    0101
OR 0010  (máscara)
  = 0111
```

XOR

Realiza la operación OR exclusivo en cada par de bits. Ejemplo:

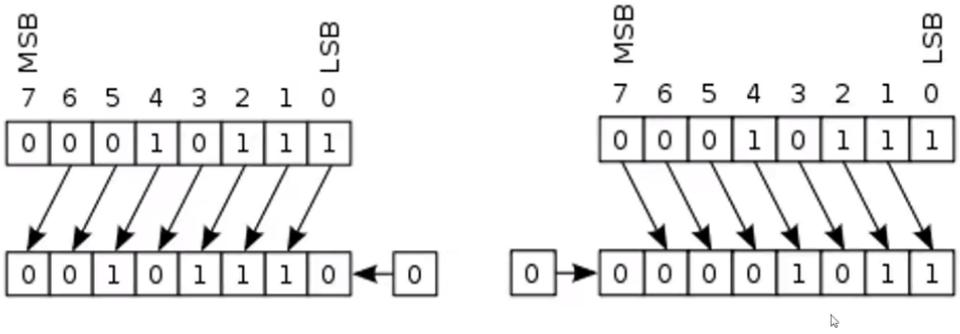
```
    0101
XOR 0011
  = 0110
```

Se puede utilizar para invertir bits:

```
    0011
XOR 0010  (máscara)
  = 0001
```

Desplazamientos de bits (bit shifts)

En estas operaciones los bits son desplazados, hacia la izquierda o hacia la derecha. Los registros tienen un ancho fijo, así que algunos bits serán desplazados hacia fuera (shifted out) mientras el mismo número de bits son desplazados hacia adentro (shifted in), es decir, entran por el otro extremo.



Desplazamiento lógico

Existen dos desplazamientos lógicos, hacia la izquierda (left shift) y hacia la derecha (right shift). Los bits de un registro son desplazados una o más posiciones hacia la derecha o hacia la izquierda. Los bit que salen del registro por un extremo se pierden y en el otro extremo del registro se rellena con un bit cero por cada bit desplazado.

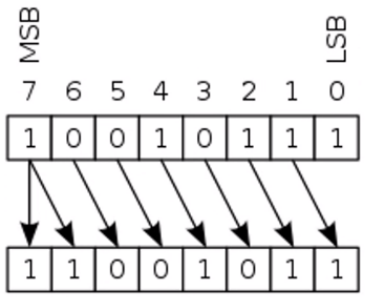
```
      10110011                10110011
1 <-- 0110011 <-- 0     0 --> 1011001 --> 1
      01100110                01011001
```

Se puede utilizar para mover bits de la parte LSE y MSB (viseversa)

Desplazamiento aritmético

Son similares a los desplazamientos lógicos, solo que los aritméticos están pensados para trabajar sobre números enteros con signo en representación de complemento a dos en lugar de enteros sin signo.

Permiten la multiplicación y la división por dos, de números enteros con signo, por una potencia de dos. Desplazar n bits hacia la izquierda o a la derecha equivale a multiplicar o dividir por 2.



Signo y magnitud

Los valores negativos en un sistema de computo se pueden representar usando el MSB para indiciar el signo. Con 8 bits se puede representar 256 números, pero si se utiliza un bit para indicar el signo solo tendríamos -127 a +127.

▼ **02-09-2021**

No hubo clase

▼ **06-09-2021**

Signo y magnitud

Los valores negativos en un sistema de computo se pueden representar usando el MSB para indiciar el signo. Con 8 bits se puede representar 256 números, pero si se utiliza un bit para indicar el signo solo tendríamos -127 a +127.

Número decimal	Signo y magnitud	Complemento a 1	Complemento a 2
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	No existe
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	no existe	No existe	1000

Complemento a uno

Complemento a dos

Complemento a dos

La forma de obtener el complemento a dos de un número binario es mediante la obtención del complemento a uno y sumarle uno.

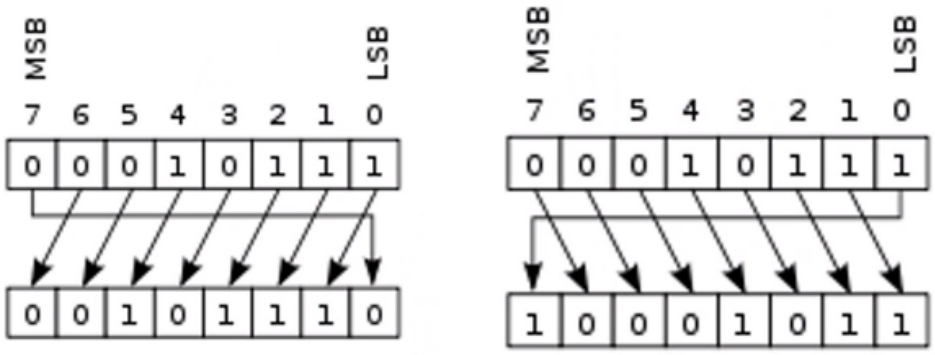
Número decimal	Signo y magnitud	Complemento a 1	Complemento a 2
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	No existe
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	no existe	No existe	1000

Rotación de bits

Distinto a mover a la izquierda o derecha

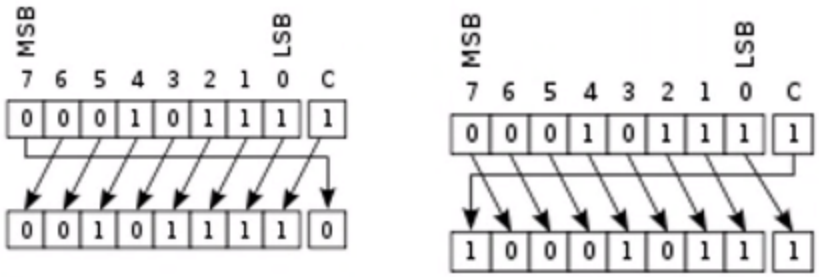
Rotación de bits

En esta operación, los bits de un registro son rotados de una manera circular como si los extremos izquierdo y derecho del registro estuvieran conectados.



Rotación con bit de acarreo

Es similar a la rotación tradicional, la diferencia es que el bit que sale por un extremo va al flag del acarreo, y el bit original que estaba en el flag del acarreo entra al registro por el extremo opuesto.



▼ **07-09-2021**

Estructura de un programa

Estructura de un programa en ensamblador

Un programa ensamblador escrito con sintaxis NASM está formado por tres secciones:

section .data *(para los datos inicializados)*

section .bss *(para los datos no inicializados)*

section .text *(para el código)*

Estructura de un programa en ensamblador

- Para definir una sección se pueden utilizar indistintamente las directivas section y segment.

- La sección .bss no es necesaria si se inicializan todas las variables que se declaran en la sección .data.

- La sección .text permite también definir variables y permitiría prescindir también de la sección .data, aunque no es recomendable.

- Por lo tanto, esta es la única sección realmente obligatoria en todo programa.

```
                    ;1: fichero hola.asm
section .data       ;2: Inicio de la sección de datos
  msg db "Hola!",10 ;3:
                    ;4: El 10 corresponde al código ASCII del salto de línea.
                    ;5:
section .text       ;6: Inicio de la sección de código.
  global main       ;7: Esta directiva es para hacer visible
                    ;8: una etiqueta para el compilador de C.
                    ;9:
  main:             ;10: Por defecto el compilador de C reconoce como
                    ;11: punto de inicio del programa la etiqueta main.
                    ;12: Mostrar un mensaje
  mov rax,4         ;13: Pone el valor 4 en el registro rax
                    ;14: para hacer la llamada a la funció write (sys_write)
  mov rbx,1         ;15: Pone el valor 1 en el registro RBX
                    ;16: para indicar el descriptor que hace referencia
                    ;17: a la salida estándar.
  mov rcx,msg       ;18: Pone la dirección de la variable msg
                    ;19: en el registro RCX
  mov rdx,6         ;20: Pone la longitud del mensaje incluido el 10
                    ;21: del final en el registro RDX
  int 80h           ;22: llama al sistema operativo
                    ;23:
                    ;24: devuelve el control al terminal del sistema operativo.
  mov rax,1         ;25: Pone el valor 1 en el registro rax
                    ;26: para hacer la llamada a la función exit (sys_exit)
  mov rbx,0         ;27: Pone el valor 0 en el registro RBX
                    ;28: para indicar el código de retorno (0=sin errores)
  int 80h           ;29: llama al sistema operativo
                    ;30:
```

Directivas

Las directivas son pseudooperaciones que solo son reconocidas por el ensamblador. No se deben confundir con las instrucciones, a pesar de que en algunos casos pueden añadir código a nuestro programa. Su función principal es declarar ciertos elementos de nuestro programa para que puedan ser identificados más fácilmente por el programador y también para facilitar la tarea de ensamblaje.

Definición de constantes

Una constante es un valor que no puede ser modificado por ninguna instrucción del código del programa. Realmente una constante es un nombre que se da para referirse a un valor determinado.

Para definir constantes se utiliza la directiva equ, de la manera siguiente:

nombre_constante equ valor

Definición de variables

La declaración de variables en un programa en ensamblador se puede incluir en la sección .data o en la sección .bss, según el uso de cada una.

Las variables de esta sección se definen utilizando las siguientes directivas:

db: define una variable de tipo byte, 8 bits.

dw: define una variable de tipo palabra (word), 2 bytes = 16 bits.

dd: define una variable de tipo doble palabra (double word), 2 palabras = 4 bytes = 32 bits.

dq: define una variable de tipo cuádruple palabra (quad word), 4 palabras = 8 bytes = 64 bits.

Definición de variables

El formato utilizado para definir una variable empleando cualquiera de las directivas anteriores es el mismo:

nombre_variable directiva valor_inicial

Ejemplos:

```
var1 db 255 ; define una variable con el valor FFh
Var2 dw 65535 ; en hexadecimal FFFFh
var4 dd 4294967295 ; en hexadecimal FFFFFFFFh
var8 dq 18446744073709551615 ; en hexadecimal FFFFFFFFFFFFFFFFh
```

Definición de variables

Si la cadena se define entre comillas abiertas (` `), también se admiten algunas secuencias de escape iniciadas con \:

\n: salto de línea
\t: tabulador
\e: el carácter ESC (ASCII 27)
\0x[valor]: [valor] ha de ser 1 byte en hexadecimal, expresado con 2 dígitos hexadecimales.
\u[valor]: [valor] ha de ser la codificación hexadecimal de un carácter en formato UTF.

▼ **08-09-2021**

Actividades y clase libre para estudiar para el examen

▼ **09-09-2021**

No hubo clase dado la realización del primer examen parcial

▼ **Segundo Parcial**

▼ **13-09-2021**

Registros x86

Registros de propósito general

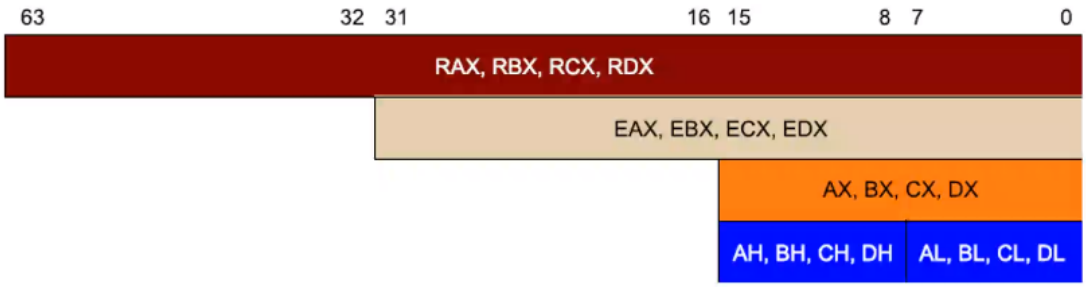
Son 16 registros de datos de 64 bits (8 bytes): RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP y R8-R15. Los 8 primeros registros se denominan de manera parecida a los 8 registros de propósito general de 32 bits disponibles en la arquitectura IA-32 (EAX, EBX, ECX, EDX, ESI, EDI, EBP y ESP).

Los registros son accesibles de cuatro maneras diferentes:

1) Como registros completos de 64 bits (quad word).
2) Como registros de 32 bits (double word), accediendo a los 32 bits de menos peso.
3) Como registros de 16 bits (word), accediendo a los 16 bits de menos peso.
4) Como registros de 8 bits (byte), permitiendo acceder individualmente a uno o dos de los bytes de menos peso según el registro.

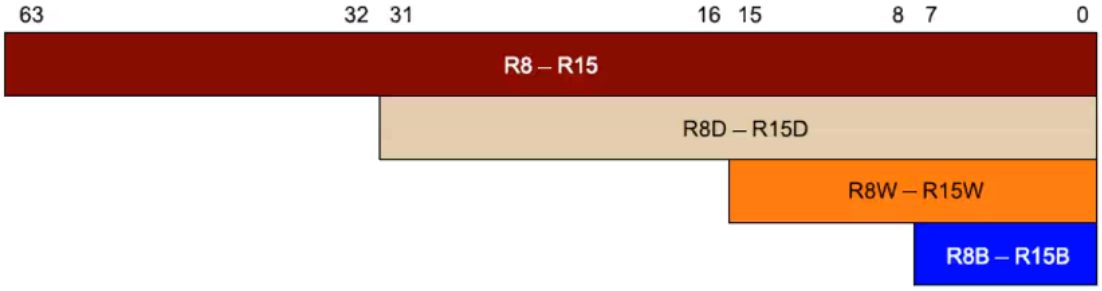
Se presenta la nomenclatura que se utiliza según si se quiere acceder a registros de 8, 16, 32 o 64 bits y según el registro.

Registros RAX, RBX, RCX y RDX



Registros RSI, RDI, RBP, RSP

Registros R8-R15



Hay algunas limitaciones en el uso de los registros de propósito general:

• En una misma instrucción no se puede usar un registro del conjunto AH, BH, CH, DH junto con uno del conjunto SIL, DIL, BPL, SPL, R8B – R15B.

• Registro RSP: tiene una función especial, funciona como apuntador de pila, contiene siempre la dirección del primer elemento de la pila. Si lo utilizamos con otras finalidades, perderemos el acceso a la pila.

• Cuando se utiliza un registro de 32 bits como operando destino de una instrucción, la parte alta del registro está fijada en 0.

Podemos distinguir varios registros de propósito específico:

1) Registros de segmento

2) Registro de instrucción o *instruction pointer (RIP)*

3) Registro de estado o *Flags register (RFLAGS)*

Hay 6 registros de segmento de 16 bits:

- CS: code segment
- DS: data segment
- SS: stack segment
- ES: extra segment
- FS: extra segment
- GS: extra segment

Estos registros se utilizan básicamente en los modelos de memoria segmentados (heredados de la arquitectura IA-32). En estos modelos, la memoria se divide en segmentos, de manera que en un momento dado el procesador solo es capaz de acceder a seis segmentos de la memoria utilizando cada uno de los seis registros de segmento.

Es un registro de 64 bits que actúa como registro contador de programa (PC) y contiene la dirección efectiva (o dirección lineal) de la instrucción siguiente que se ha de ejecutar.

Cada vez que se lee una instrucción nueva de la memoria, se actualiza con la dirección de la instrucción siguiente que se tiene que ejecutar; también se puede modificar el contenido del registro durante la ejecución de una instrucción de ruptura de secuencia (llamada a subrutina, salto condicional o incondicional).

▼ **14-09-2021**

Instrucciones de transferencia de datos

• **mov** destino, fuente: instrucción genérica para mover un dato desde un origen a un destino.

• **push** fuente: instrucción que mueve el operando de la instrucción a la cima de la pila.

• **pop** destino: mueve el dato que se encuentra en la cima de la pila al operando destino.

• **xchg** destino, fuente: intercambia contenidos de los operandos.

Instrucciones aritméticas y de comparación

• **add** destino, fuente: suma aritmética de los dos operandos.
• **adc** destino, fuente: suma aritmética de los dos operandos considerando el bit de transporte.
• **sub** destino, fuente: resta aritmética de los dos operandos.
• **sbb** destino, fuente: resta aritmética de los dos operandos considerando el bit de transporte.
• **inc** destino: incrementa el operando en una unidad.
• **dec** destino: decrementa el operando en una unidad.
• **mul** fuente: multiplicación entera sin signo.
• **imul** fuente: multiplicación entera con signo.
• **div** fuente: división entera sin signo.
• **idiv** fuente: división entera con signo.
• **neg** destino: negación aritmética en complemento a 2.
• **cmp** destino, fuente: comparación de los dos operandos; hace una resta sin guardar el resultado.

Operaciones lógicas

- **and** destino, fuente: operación 'y' lógica.
- **or** destino, fuente: operación 'o' lógica.
- **xor** destino, fuente: operación `o exclusiva´ lógica.
- **not** destino: negación lógica bit a bit.
- **test** destino, fuente: comparación lógica de los dos operandos; hace una 'y' lógica sin guardar el resultado.

Operaciones de desplazamiento

- **sal** destino, fuente / shl destino, fuente: desplazamiento aritmético/lógico a la izquierda.
- **sar** destino, fuente: desplazamiento aritmético a la derecha.
- **shr** destino, fuente: desplazamiento lógico a la derecha.
- **rol** destino, fuente: rotación lógica a la izquierda.
- **ror** destino, fuente: rotación lógica a la derecha.
- **rcl** destino, fuente: rotación lógica a la izquierda considerando el bit de transporte.
- **rcr** destino, fuente: rotación lógica a la derecha considerando el bit de transporte.

Salto incondicional

- **jmp** etiqueta: salta de manera incondicional a la etiqueta.

Saltos que consultan un bit de resultado concreto

- **je** etiqueta / **jz** etiqueta: salta a la etiqueta si igual, si el bit de cero es activo.
- **jne** etiqueta / **jnz** etiqueta: salta a la etiqueta si diferente, si el bit de cero no es activo.
- **jc** etiqueta / **jnc** etiqueta: salta a la etiqueta si el bit de transporte es activo.
- **jnc** etiqueta: salta a la etiqueta si el bit de transporte no es activo.
- **jo** etiqueta: salta a la etiqueta si el bit de desbordamiento es activo.
- **jno** etiqueta: salta a la etiqueta si el bit de desbordamiento no es activo.
- **js** etiqueta: salta a la etiqueta si el bit de signo es activo.
- **jns** etiqueta: salta a la etiqueta si el bit de signo no es activo.

Saltos condicionales sin considerar el signo

- **jb** etiqueta **/ jnae** etiqueta**:** salta a la etiqueta si es más pequeño.
- **jb** eetiqueta **/ jna etiqueta:** salta a la etiqueta si es más pequeño o igual.
- **ja** etiqueta **/ jnbe** etiqueta**:** salta a la etiqueta si es mayor.
- **jae** etiqueta **/ jnb** etiqueta**:** salta a la etiqueta si es mayor o igual.

Saltos condicionales considerando el signo

- **jl** etiqueta / **jnge** etiqueta: salta si es más pequeño.
- **jle** etiqueta / **jng** etiqueta: salta si es más pequeño o igual.
- **jg** etiqueta / **jnle** etiqueta: salta si es mayor.
- **jge** etiqueta / **jnl** etiqueta: salta si es mayor o igual.

Otras instrucciones de ruptura de secuencia

- **loop etiqueta**: decrementa el registro rcx y salta si rcx es diferente de cero.
- **call etiqueta**: llamada a subrutina.
- **ret**: retorno de subrutina.
- **iret**: retorno de rutina de servicio de interrupción (RSI).
- **int servicio**: llamada al sistema operativo.

Instrucciones de entrada/salida

- **in destino, fuente**: lectura del puerto de E/S especificado en el operando fuente y se guarda en el operando destino.

- **out destino, fuente**: escritura del valor especificado por el operando fuente en el puerto de E/S especificado en el operando destino.

▼ **15-09-2021**

No hubo clase

▼ **20-09-2021**

Descargar NASM

https://dman95.github.io/SASM/english.html

Descargar EMU 8086

https://emu8086.waxoo.com/descargar

Actividad: Instalar los dos softwares

▼ **21-09-2021**

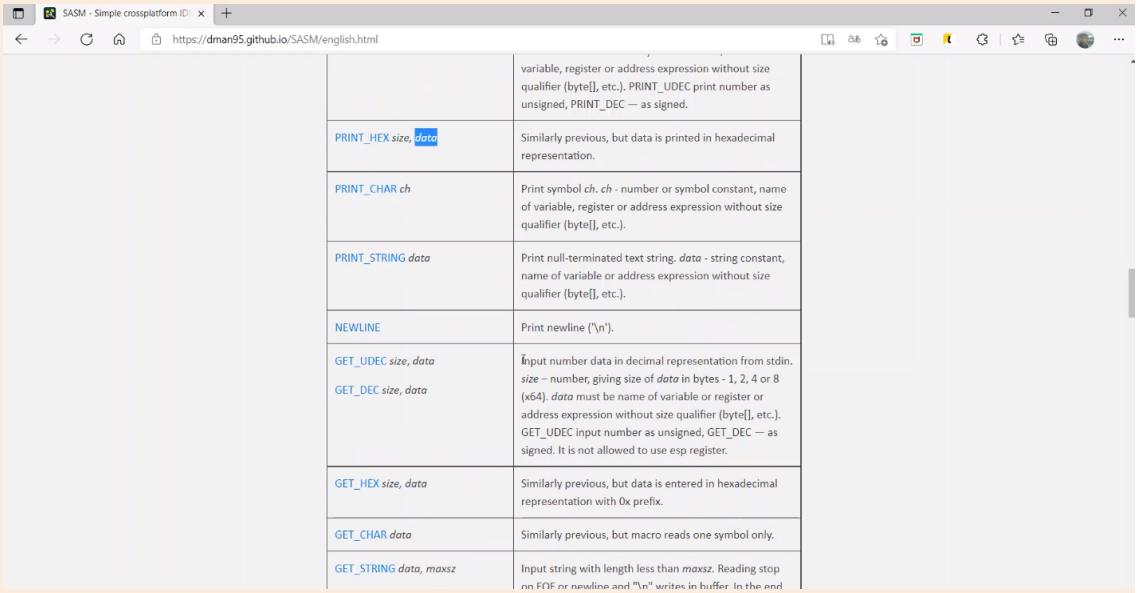
Ejemplo de uso de SASM

```asm
%include "io64.inc"

section .text
    global CMAIN

CMAIN:
    mov rbp, rsp; for correct debugging
    mov     eax, 17      ;
    mov     ebx, 9       ;
    add     eax, ebx     ; add ebx to eax
    PRINT_DEC 1, eax
    NEWLINE
    ;xor rax, rax
    ret                  ;retorno de subrutina
```

Página de documentación:




```asm
%include "io64.inc"
section .data
constante equ 80
var dw 33h
section .text
global CMAIN
CMAIN:
mov rbp, rsp; for correct debugging
mov eax, constante ;
```

```
mov ebx, var ;
add eax, ebx ; add ebx to eax
PRINT_DEC 1, eax
NEWLINE
;xor rax, rax
ret ;retorno de subrutina
```

▼ 22-09-2021

No hubo clase

▼ 23-09-2021

No hubo clase

▼ 27-09-2021

Revisión de SASM

Programas en diferentes ensambladores:

GAS:

```
.data
msg:
    .asciz "Hello, world!\n"

.extern _printf
.text
    .global _main # entry point
_main:
    movl %esp, %ebp # for correct debugging
    pushl $msg
    call _printf
    addl $4, %esp
    xorl %eax, %eax
    ret
```

Se realizará una suma

```
%include "io.inc"

section .text
global CMAIN
CMAIN:
    MOV eax, -200000
    MOV ebx, -200
    ADD eax, ebx
    PRINT_DEC 4, eax

    xor eax, eax
    ret
```

```
%include "io64.inc"

section .text
global CMAIN
CMAIN:
    mov rax, 200000    ; MOVER UN VALOR A EL REGISTRO RAX de 64bits
    mov rbx, 2000      ; MOVER UN VALOR A EL REGISTRO RBX de 64bits
    mov rax, rbx       ; SUMAR RAX Y RBX, RESULTADO EN RAX
    PRINT_DEC 8, rbx     ;PRINT_DEC size, data

    xor rax, rax
    ret
```

```
%include "io64.inc"

section .text
global CMAIN
CMAIN:


    mov al, 0b10101101
    mov bl, 0b11111110
    and al, bl
    PRINT_DEC 1, al     ;PRINT_DEC size, data
    NEWLINE

    xor rax, rax
    ret
```

```
%include "io64.inc"

section .text
global CMAIN
CMAIN:


    mov al, 0b10101101
    mov bl, 0b11111110
    and al, bl
    PRINT_HEX 1, al     ;PRINT_DEC size, data
    NEWLINE

    mov al, 0b10101101 ; 0xAD
    mov bl, 0b01000000
    or al, bl
    PRINT_HEX 1, al     ;PRINT_DEC size, data
    NEWLINE

    xor rax, rax
    ret
```

Entrada

Salida

ac
ed

▼ **28-09-2021**

En wikipedia hay un anexo de instrucciones del x86

▼ **29-09-2021**

No hubo clase

▼ **30-09-2021**

No hubo clase

▼ **04-10-2021**

Ejemplos de la Macrolibrería y Programas con lazos

Usar todas las macros:

https://dman95.github.io/SASM/english.html

libraries ("CEXTERN printf" for example).

Macro name	Description
PRINT_UDEC *size, data* PRINT_DEC *size, data*	Print number *data* in decimal representation. *size* – number, giving size of *data* in bytes - 1, 2, 4 or 8 (x64). *data* must be number or symbol constant, name of variable, register or address expression without size qualifier (byte[], etc.). PRINT_UDEC print number as unsigned, PRINT_DEC — as signed.
PRINT_HEX *size, data*	Similarly previous, but data is printed in hexadecimal representation.
PRINT_CHAR *ch*	Print symbol *ch*. *ch* - number or symbol constant, name of variable, register or address expression without size qualifier (byte[], etc.).
PRINT_STRING *data*	Print null-terminated text string. *data* - string constant, name of variable or address expression without size qualifier (byte[], etc.).
NEWLINE	Print newline ('\n').
GET_UDEC *size, data* GET_DEC *size, data*	Input number data in decimal representation from stdin. *size* – number, giving size of *data* in bytes - 1, 2, 4 or 8 (x64). *data* must be name of variable or register or address expression without size qualifier (byte[], etc.).

x64	x86		
8 bytes	4 bytes	2 bytes	1 byte
rax	eax	ax	al , ah
rcx	ecx	cx	cl , ch
rdx	edx	dx	dl , dh
rbx	ebx	bx	bl , bh
rsp	esp	sp	spl*
rbp	ebp	bp	bpl*
rsi	esi	si	sil*
rdi	edi	di	dil*
r8-r15	r8d-r15d*	r8w-r15w*	r8b-r15b*

```
%include "io64.inc"

section .text
global CMAIN
CMAIN:
    GET_UDEC 4, eax
    GET_DEC 4, ebx
    GET_DEC 4, ecx
    GET_CHAR 1, AH

    PRINT_UDEC 4, eax
    NEWLINE
    PRINT_DEC 4, ebx
    NEWLINE
    PRINT_DEC 4, ecx
    NEWLINE
    PRINT_CHAR 1, edx

    ret ; Retorno desde un procedimiento
```

```
section .text
global CMAIN

CMAIN:
    GET_UDEC 4, eax
    GET_DEC 4, ebx
    GET_DEC 4, ecx
    GET_STRING msg, 4
    GET_CHAR r8B
    GET_HEX 1, r9B

    PRINT_UDEC 4, eax
    NEWLINE
    PRINT_DEC 4, ebx
    NEWLINE
    PRINT_DEC 4, ecx
    NEWLINE
    PRINT_STRING msg
    NEWLINE
    PRINT_CHAR R8B
    NEWLINE
    PRINT_HEX 1, R9B

    ret ; Retorno desde un procedimiento
```

▼ **05-10-2021**

No hubo clase

▼ **06-10-2021**

No hubo clase por el Congreso de Ciencias Exactas

▼ **07-10-2021**

No hubo clase por el Congreso de Ciencias Exactas

▼ **11-10-2021**

Más códigos de ejemplos

▼ **12-10-2021**

Consideraciones de entregables

Las actividades y reportes de prácticas se realizan en formato pdf.
Las actividades y reportes de prácticas se realizan en forma individual.
Todas las prácticas deben tener reporte.
Todos los entregables deben tener portada.
El texto debe ir justificado y con la misma fuente.
Los reportes de practica deben de incluir: Portada, objetivos, contenido y conclusión.
Los reportes deben de evidenciar el desarrollo de la práctica (capturas y resultados)
No se deben adjuntar hojas en blanco en los documentos.

▼ **13-10-2021**

No hubo clase para hacer las prácticas

▼ **14-10-2021**

No hubo clase

▼ **18-10-2021**

Clase de aclaración de nuestras prácticas pendientes

▼ **19-10-2021**

No hubo clase

▼ **20-10-2021**

No hubo clase

▼ **21-10-2021**

No hubo clase debido a que se realizó la evaluación del segundo parcial

▼ **Tercer Parcial**

▼ **25-10-2021**

UNIVERSIDAD AUTONOMA DE AGUASCALIENTES

Lenguaje ensamblador

Representación de punto fijo y punto flotante

Profesor: MSE. Cristian Jael Mejia Aguirre

Representación de punto fijo y punto flotante

Los números reales en base diez que nosotros habitualmente utilizamos para realizar cálculos matemáticos ya sea para tareas cotidianas o científicas generalmente son expresados en una de dos formas:

En punto fijo (por ejemplo: $345.70, 230Kg, -10°C) donde se emplean tres campos para la representación: signo, la parte entera y la parte decimal.

En notación científica o punto flotante, donde un número se expresa también con tres campos: signo, mantisa y exponente (por ejemplo: 10E-09 seg. = 0,000000001 seg. , 12.74E04 metros = 127400 metros, etc.).

Representación de números binarios en punto fijo

La parte decimal, también queda embebida dentro de la representación del número completo, ya que cuando se deban realizar operaciones aritméticas, al igual que como se procede en números de base 10, se toma todo el número para la operación.

En la suma y resta, la posición del punto nunca se modifica. Si hay que sumar o restar dos números, se debe primero hacer coincidir las posiciones del punto y luego realizar la operación aritmética como si el número fuera entero.

En caso de realizar operaciones de multiplicación y división, el procedimiento es análogo; lo único que cambia es que se debe correr el punto al número binario resultado, según corresponda, de igual forma que en el caso de números decimales.

Representación de números binarios en punto fijo

Ejemplo: Suma en Ca2 de dos números:

A = 0111001,110001 y B = 1110011,001

Se disponen los números de la siguiente forma y se suma:

$$\begin{array}{r} 0111001,110001 \\ 1110011,001000 \\ \hline \underline{1}0101100,111001 \end{array}$$

Representación de números binarios en punto fijo

Un número expresado en punto fijo, tendrá un número limitado de símbolos, lo que nos limitará el rango de representación. En general tendrá m símbolos en la parte entera y n símbolos en la parte decimal.

Suponiendo que tenemos un número binario expresado con 5 cifras significativas (5 bits después del punto) y de la siguiente expresión obtenemos que:

La más pequeña representación es: $2^{-5} = 1/(2^5) = 1/32_{10} = 0.03125_{10}$

Representación de números binarios en punto flotante

Como en el caso de números de base decimal, también es necesario en algunos casos trabajar con números muy grandes o pequeños. Para ello resulta mas cómodo poder expresarlos en un formato que permita operar con números de diferente posición del punto.

Su uso se hace imprescindible cuando hay que realizar por ejemplo cálculos entre números muy grandes, donde el formato permite poder describirlos con mayor comodidad para su operativa y visualización del resultado.

La forma usual es expresarlos con tres campos: uno de signo del número total, otro con la mantisa y el tercero con el exponente del número.

Representación de números binarios en punto flotante

En binario es igual. Se han propuesto varias formas de representación de números de coma flotante. El que más ha prevalecido es el definido por la IEEE bajo la norma IEEE P754. En ella se define el formato que se debe respetar para la representación de un número binario con signo en punto flotante como así también el tratamiento de errores, operaciones, etc. Según esta norma un número binario en punto flotante normalizado se puede expresar de tres formas diferentes, según el rango del mismo y son de menor a mayor:

Punto flotante de simple precisión.
Punto flotante de doble precisión.
Punto flotante de precisión extendida.
Punto flotante en formato BCD empaquetado.

Punto flotante de simple precisión

Signo	Exponente	Fracción del Significando
1 bit:: "0" positivo	8 bits	23 bits

El número expresado en este formato consta de un total de 32 bits. El primero es para expresar el signo del número y como convención se adopta: "0" positivo y "1" negativo.

Punto Flotante Precisión Extendida

Signo	Exponente	Campo nulo	Significando
1 bit: "0" positivo	15 bits	16 bits: todos "0"	64 bits: x,x.....x

Este formato de representación tiene como diferencia sustancial con los anteriores que hay un campo denominado "significando" y no "fracción del significando", dado que en el primero se permite poder expresar números no normalizados ya que se dispone de la representación de la parte entera que puede valer "1"como el los dos primeros casos ó "0", que no puede ser realizado en los mismos.

Punto Flotante BCD empaquetado

Signo Signific.	Signo exp	Ind. esp	Exponente	Todos "0"	Significando
1 bit	1 bit	2 bits	12 bits	12 bits	68 bits

Signo del significando: "0": positivo, "1": negativo.
Signo del exponente: Ídem anterior.
Indicador especial.
Exponente: 3 dígitos BCD (12 bits).
Campo nulo: Todos "0".
Significando: 17 dígitos BCD de la forma x.x.....x con el primer dígito en la parte entera y los 16 restantes como fracción.

Este formato además de la particularidad que puede representar un número decimal en formato BCD empaquetado, dispone de un campo de signo para el exponente y otro para indicar situaciones especiales.

▼ **26-10-2021**

Explicación de números flotantes

▼ **27-10-2021**

Explicación de punto flotante

▼ **28-10-2021**

No hubo clase

▼ **01-11-2021**

MS DOS

Uno de los primeros lenguajes de programación era el ensamblador

UNIVERSIDAD AUTONOMA DE AGUASCALIENTES

Lenguaje ensamblador

Estructura MS-DOS

Profesor: MSE. Cristian Jael Mejia Aguirre

MicroSoft Disk Operating System
Sistema operativo de disco de Microsoft

MS-DOS es el nombre de uno de los sistemas operativos para sistemas informáticos basados en una arquitectura x86 y diseñados por la empresa norteamericana de software Microsoft. Se le conocía popularmente como DOS.

MicroSoft Disk Operating System
Sistema operativo de disco de Microsoft

MS-DOS es un sistema operativo sin interfaz gráfica que confiaba puramente en una interfaz de línea de comandos para funcionar, con un total de 109 comandos con los que movernos por el sistema y utilizar sus capacidades.

Versiones

Entre las versiones o evoluciones del Sistema operativo DOS, se encuentran:

PC DOS 1.0 - 1981
PC DOS 1.1 - 1981
PC DOS 1.25 - 1982
PC DOS 2.0 - 1983
PC DOS 2.11 - 1983
PC DOS 3.0 – 1984
PC DOS 3.1 – 1984
PC DOS 3.3 – 1987
PC DOS 4.0 - 1988
PC DOS 4.01 – 1989
PC DOS 5.0 - 1991
PC DOS 5.0a - 1992
MS-DOS 6.0 - 1992
MS-DOS 6.21 - 1993
MS-DOS 7.1 - 1997
MS-DOS 8.0 -1999
PC DOS 2000 - 2000

Tipo de sistema operativo

DOS es un sistema operativo monousuario (solo puede ser utilizado por una persona de cada vez) y monotarea (solo se puede ejecutar un programa a la vez). La comunicación del usuario con MS-DOS se produce de dos modos: el modo interactivo y el modo por lotes.

Estructura de MS-DOS

Este sistema tiene un núcleo monolítico que es una arquitectura de núcleo donde todo el núcleo se ejecuta en el espacio de kernel en modo de supervisión. En común con otras arquitecturas (micronúcleo, núcleo híbrido), el núcleo define una capa de alto nivel de abstracción sobre el hardware del equipo, con un grupo de llamadas al sistema para implementar los servicios del sistema operativo, como la competencia, la administración de procesos y la gestión de memoria en uno o más módulos.

Aunque cada módulo de mantenimiento de dichas operaciones sea separado de una forma general, es muy difícil hacer el código de integración entre todos estos módulos, y, una vez que todos los módulos se ejecutan en el mismo espacio de direcciones, un error en un módulo puede derribar todo el sistema.

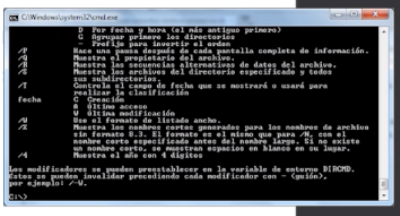
Comandos básicos

Los comandos de MS-DOS y que actualmente pueden ser utilizados desde la línea de comandos en sistemas operativos Windows. Existen dos tipos de comandos: internos y externos.

Presentación

Comandos internos

Los comandos internos o residentes son aquellos que se transfieren a la memoria en el momento de cargarse el Sistema Operativo y se pueden ejecutar sin necesidad de tener el DOS presente en la unidad por defecto desde el cual se puede ejecutar el mandato.



Comandos externos

Los comandos externos en contraposición con los comandos internos se almacenan en archivos de comandos denominados transitorios o externos, y para ejecutarse necesitan de estos archivos, además los comandos externos tienen nombre propio y se pueden copiar de un disco a otro.

Comandos internos

CD o CHDIR - Cambia el directorio actual.

CD nombre_directorio Cambia al directorio jerárquicamente inferior.

CD *PATH - Especifica trayectorias, en las cuales el sistema operativo busca archivos ejecutables. Es un comando que se suele escribir en el Config.sys y en archivos de procesos por lotes.

CLS - Limpia todos los comandos y toda la información que hay en pantalla, excepto la letra de la unidad usada (Por ejemplo C:\)

COPY - Copiar un archivo de un directorio a otro

COPY CON Realizar archivos extensión .bat

DIR - Lista los directorios y archivos de la unidad o directorio actual.

FOR - Repite un comando

PROMPT- Cambia la línea de visualización de la orden.

MD - Crea un nuevo directorio.

RD o RMDIR- Elimina un directorio.

REM - Permite insertar comentarios en archivos de proceso por lotes.

REN o RENAME - Renombra archivos y directorios.

SET - Asigna valores a variables de entorno.

TIME - Visualiza o cambia la hora del reloj interno.

TYPE - Muestra el contenido de un fichero. Se utiliza, principalmente, para ver contenidos de ficheros en formato texto.

VER - Muestra la versión del Sistema Operativo.

VOL - Muestra la etiqueta del disco duro y su volumen (si lo tiene).

MEM - Muestra la cantidad de memoria RAM, la cantidad ocupada y la libre.

el Mejia Aguirre

▼ **02-11-2021**

No hubo clase. Asueto

▼ **03-11-2021**

BIOS Basic Input/Output System

El sistema básico de entrada-salida o BIOS es un estándar que define la interfaz de firmware para computadoras. También es conocido como BIOS del sistema, ROM BIOS2 y BIOS de PC.

El firmware del BIOS es instalado dentro de la PC y es el primer programa que se ejecuta cuando se enciende la computadora.

El propósito fundamental del BIOS es iniciar, y probar el hardware del sistema y cargar un gestor de arranque o un sistema operativo desde un dispositivo de almacenamiento de datos. Además, el BIOS provee una capa de abstracción para el hardware, por ejemplo, que consiste en una vía para que los programas de aplicaciones y los sistemas operativos interactúen con el teclado, el monitor y otros dispositivos de entrada/salida.

Interrupción

Una interrupción es el rompimiento en la secuencia de un programa para ejecutar un programa especial llamando una rutina de servicio cuya característica principal es que al finalizar regresa al punto donde se interrumpió el programa. Existen dos clases de interrupciones:

- Interrupciones por software: Son aquellas programadas por el usuario, es decir, el usuario decide cuando y donde ejecutarlas, generalmente son usadas para realizar entrada y salida.

- Interrupciones por hardware: Son aquellas que son provocadas por dispositivos externos al procesador su característica principal es que no son programadas, esto es, pueden ocurrir en cualquier momento en el programa.

Interrupciones

Existen dos clases de interrupciones por HW:

- Interrupciones por hardware enmascarables: Aquellas en las que el usuario decide si quiere o no ser interrumpido.

- Interrupciones por hardware no enmascarables (NMI): Aquellas que siempre interrumpen al programa.

Asociado al concepto de interrupción se tiene un área de memoria llamada vector de interrupciones; la cual contiene las direcciones de las rutinas de servicio de cada interrupción.

Tabla de interrupciones

Interrupción		Uso
Hex	Decimal	
00h	0	Generada por la CPU cuando se intenta hacer una división por cero
01h	1	Utilizada par ir paso a paso por los programas (como DEBUG)
02h	2	Interrupción no enmascarable
03h	3	Utilizada para establecer puntos de ruptura en programas (igual que con DEBUG)
04h	4	Generada cuando operaciones aritméticas dan operaciones de desbordamientos
05h	5	Invoca la rutina de servicio de imprimir pantalla de la ROM BIOS
06h	6	Reservada para DOS
07h	7	Reservada para DOS
08h	8	Generada por el tic-tac del reloj del hardware
09h	9	Generada por acción del teclado
0Ah a 0Dh	10 a 13	Reservadas para BIOS
0Eh	14	Señala atención al diskette (por ejemplo, para señalar operación completada)
0Fh	15	Utilizada para controlar la impresora
10h	16	Invoca servicios de vídeo de la ROM BIOS
11h	17	Invoca el servicio de lista de equipamiento de ROM BIOS
12h	18	Invoca servicio de tamaño de memoria de la ROM BIOS
13h	19	Invoca servicios de disco de la ROM BIOS
14h	20	Invoca servicios de comunicaciones de la ROM BIOS
15h	21	Invoca servicios del sistema de la ROM BIOS
16h	22	Invoca los servicios estándar del teclado de la ROM BIOS

Tabla de interrupciones

32h		
33h	51	Funciones del driver del ratón
34h a 3Eh	52 a 62	Reservado para DOS
3Fh	63	Gestor Overlay
40h	64	Reasignación Disquete BIOS
41h	65	Apunta a la tabla de parámetros del disco duro
42h	66	Gestor Vídeo Reasignado
43h	67	Apunta a los caracteres gráficos de vídeo (EGA, PS/2)
44h	68	API red Novell
45h	69	Reservado
46h	70	Parámetros del disco duro
47h a 49h	71 a 73	Reservadas para BIOS
4Ah	74	Alarma usuario
4Bh a 5Fh	75	Reservadas para BIOS
60h a 66h	A 102	Reservadas a programas
67h	103	Invoca al gestor de memoria expandida LIM
68h a 69h	104 a 105	Reservadas para BIOS
70h	106	Reloj tiempo real

Tabla de interrupciones

71h a 74h	107 a 110	Reservadas para BIOS
75h	111	Redirigido a interrupción NMI
76h a 79h	112 a 114	Reservadas
7Ah	115	Software Novell (API)
7Bh a 7Fh	116 a 120	No usadas
80h a F0h	121 a	Reservado para BASIC
F1h a FFh	A 255	Reservadas a programas

▼ **04-11-2021**

No hubo clase

▼ **08-11-2021**

Práctica 7: Operaciones con flotantes

Práctica 8: INT 21H

Práctica 9: Interfaz con lenguaje C sin parámetros

Práctica 10: Interfaz con lenguaje C sin parámetros

Examen tercer parcial

Proyecto final en equipo

Un men debe enviar por teams al profe el equipo

Presentar el proyecto

▼ **09-11-2021**

???

▼ **10-11-2021**

Lenguaje C

C es un lenguaje de programación de propósito general originalmente desarrollado los Laboratorios Bell, como evolución del anterior lenguaje B.

Al igual que B, es un lenguaje orientado a la implementación de sistemas operativos, concretamente Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas y aplicaciones.

La primera estandarización del lenguaje C fue en ANSI, con el estándar X3.159-1989. El lenguaje que define este estándar fue conocido vulgarmente como ANSI C. Posteriormente, en 1990, fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia por lo que, si los programas creados lo siguen, el código es portable entre plataformas y/o arquitecturas.

Se trata de un lenguaje de tipos de datos estáticos, de medio nivel, que dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a bajo nivel. Los compiladores ofrecen extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

Inline Assembler (C)

El ensamblador alineado permite incrustar instrucciones de lenguaje ensamblador directamente en los programas de origen C sin realizar pasos adicionales de ensamblado y vínculo. El ensamblador alineado se compila en el compilador, por lo tanto, no se necesita un ensamblador independiente.

Dado que el ensamblador alineado no requiere pasos de ensamblado y vínculo independientes y de vínculo, es más aconsejable que un ensamblador independiente. El código de ensamblado alineado puede utilizar cualquier nombre de variable o de función de C que esté en el ámbito, por lo que resulta fácil integrarlo con el código C del programa. Dado que el código de ensamblado se puede combinar con instrucciones de C, puede realizar tareas complicadas o imposibles solo en C.

La palabra clave **__asm** invoca el ensamblador alineado y puede aparecer siempre que una instrucción de C sea válida. No puede aparecer por sí sola. Debe ir seguida de una instrucción de ensamblado, un grupo de instrucciones entre llaves o, como mínimo, un par de llaves vacío. El término "bloque **__asm** " aquí hace referencia a cualquier instrucción o grupo de instrucciones, incluido o no entre llaves.

El código siguiente es un bloque __asm simple incluido entre llaves. (El código es una secuencia de prólogo de función personalizada).

```
__asm
{
  push ebp
  mov  ebp, esp
  sub  esp, __LOCAL_SIZE
}
```

Como alternativa, puede colocar __asm delante de cada instrucción de ensamblado:

```
__asm push ebp
__asm mov  ebp, esp
__asm sub  esp, __LOCAL_SIZE
```

▼ **11-11-2021**

Actividad en aula virtual

▼ **15-11-2021**

No hubo clase. Asueto

▼ **16-11-2021**

Ejemplo de programación ASM en C

▼ **17-11-2021**

No hubo clase

▼ **18-11-2021**

No hubo clase

▼ **22-11-2021**

No hubo clase

▼ **23-11-2021**

???

▼ **24-11-2021**

▼ **25-11-2021**