Programación en Lenguaje Ensamblador (x86): Construcciones básicas de los lenguajes de alto nivel.

HISTORIAL DE REVISIONES

NÚMERO	FECHA	MODIFICACIONES	NOMBRE
v1.0.0	2018-11-01		CA

Índice

1. Temario

1. Programación en lenguaje Ensamblador x86
    a. x86 :Representación de Datos, Representación de Instrucciones, Modos de direccionamiento.
    b. Sentencias de asignación
    c. Sentencias condicionales
    d. Bucles
    e. LLamadas y retorno de funcion o subrutina

2. Introducción

2.1. Objetivos

- Analizar la arquitectura del repertorio de las instrucciones máquina (Formato de instrucciones, formato de datos, operaciones y direccionamiento de operandos) de la arquitectura x86-64 para su utilización en el desarrollo práctico de programas en lenguaje ensamblador GNU_asm_x86 (gas).
- Capacidad para desarrollar pequeños programas en lenguaje ensamblador x86, siendo función de las prácticas de laboratorio su puesta en práctica.

2.2. Requisitos

- Requisitos:
    - Von Neumann Architecture: Arquitectura de una Computadora, Máquina IAS.
    - Programación en lenguaje ensamblador IAS
    - Representacion de datos
    - Operaciones Aritméticas y Lógicas
    - Representación de las Instrucciones

3. Procesadores Intel con arquitectura x86

3.1. Nomenclatura

3.1.1. General

- Los procesadores intel reciben nombre por todos conocidos: Pentium II, Pentium III, Corei3, Corei5, Corei7, etc
- La arquitectura de las computadoras que utilizan dichos procesadores responden a una arquitectura común
    - Arquitectura x86 en el caso de la arquitectura de 32 bits
    - Arquitectura x86-64 en el caso de la arquitectura de 64 bits.
- Procesadores con arquitectura x86 de 32 bits

```
*** 1978 y 1979 Intel 8086 y 8088. Primeros microprocesadores de la arquitectura x86.
    1980 Intel 8087. Primer coprocesador numérico de la arquitectura x86, inicio de la  ↩
        serie x87.
    1980 NEC V20 y V30. Clones de procesadores 8088 y 8086, respectivamente, fabricados  ↩
        por NEC.
    1982 Intel 80186 y 80188. Mejoras del 8086 y 8088.
    1982 Intel 80286. Aparece el modo protegido, tiene capacidad para multitarea.
*** 1985 Intel 80386. Primer microprocesador x86 de 32 bits.
    1989 Intel 80486. Incorpora el coprocesador numérico en el propio circuito integrado.
    1993 Intel Pentium. Mejor desempeño, arquitectura superescalar.
    1995 Pentium Pro. Ejecución fuera de orden y Ejecución especulativa
    1996 Amd k5. Rival directo del Intel Pentium.
    1997 Intel Pentium II. Mejora la velocidad en código de 16 Bits, incorpora MMX
    1998 AMD K6-2. Competidor directo del Intel Pentium II, introducción de 3DNow!
    1999 Intel Pentium III. Introducción de las instrucciones SSE
    2000 Intel Pentium 4. NetBurst. Mejora en las instrucciones SSE
    2005 Intel Pentium D. EM64T. Bit NX, Intel Viiv
```

- Procesadores con arquitectura x86-64 de 64 bits

```
*** 2003 AMD Opteron. Primer microprocesador x86 de 64 bits, con el conjunto de  ↩
    instrucciones AMD64)
    2003 AMD Athlon.
    2006 Intel Core 2. Introducción de microarquitectura Intel P8. Menor consumo,  ↩
        múltiples núcleos, soporte de virtualización en     hardware    incluyendo x86-64  ↩
        y SSSE3.
    2008 Core i7
    2009 Core i5
    2010 Core i3
```

- significado del código de los nombres de procesadores intel serie Core: El primer dígito del código indica la generación (en el 2017 salió la 8ª generación)
- Intel Serie Core:
  - https://www.intel.com/content/www/us/en/products/processors/core/view-all.html
- Intel Serie Core X : familias i9, i7 ,i5
  - Intel Serie X:
  - https://ark.intel.com/products/series/123588/Intel-Core-X-series-Processors
- https://es.wikipedia.org/wiki/Anexo:Procesadores_AMD
- COMPETENCIA INTEL-AMD año 2018 en computadoras de sobremesa.
  - AMD Ryzen 2nd Generation - INTEL Core i7-8086K

3.1.2. linux i386/amd64

- En el entorno Linux a la arquitectura de intel x86 de 32 bits la denomina **i386** y a la arquitectura de amd x86-64 la denomina **amd64**.

3.1.3. procesador o arquitectura

- Al hablar de la arquitectura de un procesador nos referimos a la Instruction Set Architecture (ISA) del procesador. Por lo que no nos referiremos a un procesador específico sino a un conjunto de procesadores todos ellos con la misma ISA ejecutanto el mismo repertorio de instrucciones máquina. Hablaremos de un procesador i386 ó de un procesador amd64 para referirnos a la **ISA compatible** con dicho procesador.

4. Estructura de la Computadora

4.1. CPU

- Componentes básicos de la CPU
  - ALU
  - Unidad de Control
  - Registros internos
- Función de la CPU
  - Llevar a cabo el ciclo de instrucción de las instrucciones almacenadas en la memoria principal

4.2. Memoria

- Jerarquía de Memoria: Registros internos a la CPU y Memoria principal (DRAM) externa a la CPU

4.3. Memoria Principal

- Memoria DRAM: Dynamic Random Access Memory
- Almacena la secuencia de instrucciones máquina binarias y los datos en formato binario.
- Espacio de direcciones lineal: Notación hexacecimal
- Direccionamiento: bytes : notación hexadecimal

4.4. Registros internos a la CPU

4.4.1. introducción

- Registos accesibles NO accesibles por el programador en la arquitectura amd64
  - PC: Contador del Programa : x86 lo denomina RIP: 64 bits
  - IR: Registro de instrucción : 64 bits
  - MBR: Registro buffer de Memoria : 64 bits $\rightarrow$ WORD SIZE : 64
  - MAR: Registro de direcciones de Memoria: 40 bits
    - Capacidad de Memoria: 2^{40} : 1TB
- Para el caso de la arquitectura i386 sustituir 64 bits por 32 bits y el registro MAR también es de 32 bits.

4.4.2. Registros visibles al programador

63-0	31-0	15-0	15-8	7-0
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si		sil
rdi	edi	di		dil
rbp	ebp	bp		bpl

rsp	esp	sp		spl
r8	r8d	r8w		r8b
r9	r9d	r9w		r9b
r10	r10d	r10w		r10b
r11	r11d	r11w		r11b
r12	r12d	r12w		r12b
r13	r13d	r13w		r13b
r14	r14d	r14w		r14b
r15	r15d	r15w		r15b

4.4.3. Compatibilidad 32-64

- En la nominación de los registros de la arquitectura de 64 bits sustituir R por E y obtenemos la nominación de la arquitectura de 32 bits.

64 bits	32 bits
RIP	EIP
RAX	EAX
RFLAG	EFLAG
.....	.....

- Hay excepciones

4.4.4. Control Flag Register

- Registro de STATUS: La ejecución de una instrucción, activa unos bits denominados banderines que indican consecuencias de la operación realizada. Ejemplo: banderín de overflow : indica que la operación aritmética realizada ha resultado en un desbordamiento del resultado de dicha operación.
- wikipedia
- Unicamente nos fijamos en los flags OSZAPC.

Cuadro 1: RFLAG Register

Flag	**Bit**	**Name**
CF	0	Carry flag
PF	2	Parity flag
AF	4	Adjust flag
ZF	6	Zero flag
SF	7	Sign flag
OF	11	Overflow flag

- Carry flag:
  - se activa si la llevada afecta a una posición de bit mayor que del ancho de palabra (word size) de la ALU en una operación aritmética de números enteros sin signo o con signo
- Overflow flag:
  - se activa si teniendo en cuenta el bit de mayor peso MSB (aunque esté fuera el word size) indicase error en la operación aritmética con números enteros con signo. Si no se tiene en cuenta el MSB fuera del word size, la operación es correcta.

- Parity Even flag:
  - indica si el número de bits del byte LSB del resultado de la última operación ha sido par.
- Sign flag:
  - se activa si el resultado de la última operación ha sido negativo.
- Adjust flag:
  - se activa si hay llevada en el nibble LSB del resultado de la última operación
- Ejemplos

```
  11111111
+ 00000001
____________
 100000000 -> hay acarreo y también overflow ya que si consideramos el MSB=1 de la  ↩
    posición 8ª el resultado sería erróneo, es decir, si realizasemos una    extensión de ↩
     signo el resultado sería erroneo y por lo tanto hay que rechazar el dígito fuera de  ↩
    rango con lo que la operación es correcta. El bit de signo es el de la posición más  ↩
    alta del "word size" (posición 7ª)


 A :    11110000
-B :   +11101100
____________________
A-B:   111011100 -> hay acarreo pero no overflow ya que el MSB=1 en una posición fuera del ↩
    "word size" no afecta al signo del resultado, ya que las posiciones del MSB (8ª) y la ↩
    anterior (7ª) del bit de signo tienen el mismo dígito de valor 1


 A :    10000000
-B :   +10000000
____________________
A-B:   100000000 -> hay acarreo. Hay también overflow ya que el bit MSB de valor 1 es  ↩
   diferente de la posición anterior ( 7ª, bit de signo) de valor 0.  Los dos sumandos  ↩
   son negativos (bit de signo posición 7ª) y el bit de signo del resultado (bit posición ↩
    7ª) es positivo luego el resultado es erróneo.
```

4.4.5. Otros Registros

./images/instrucciones_representacion/registers_1200x800.png

- Los registros fp, mmx y xmm se utilizan para ejecutar instrucciones complejas como la tangente que operan con números reales en coma flotante o instrucciones que ejecutan operaciones sobre múltiples datos enteros (Single Instruction Multiple Data) (P.ej producto escalar).
- Más información en el apéndice FPU_x87

5. Lenguaje Intel versus Lenguaje AT&T

5.1. Lenguajes ensamblador de la arquitectura i386/amd64

- El lenguaje en código máquina del repertorio de instrucciones de la arquitectura AMD64 es único pero no así el lenguaje ensamblador correspondiente a dicha arquitectura.
- En la asignatura "Estructura de Computadores" se utiliza la sintaxis **AT&T** de la compañía telefónica americana AT&T.

5.2. Sintaxis de las instrucciones en el lenguaje INTEL

- El formato de las instrucciones en lenguaje ensamblador se conoce como *sintaxis* de las instrucciones.
- SINTAXIS ASM: Etiqueta-Código de Operación- Operando1- Operando2- Comentario
- x86-64
- x86

Cuadro 2: Sintaxis Intel

label:	op_mnemonic	operand_destination	,	operand_source	#comment

- Ejemplo:
  - `bucle: sub rsp,16 ;comienzo del bucle`
  - `suma: add eax,esi ;sumar`
  - + mov ax,[resultado] ;salvar resultado+

---

nota
La sintaxis del lenguaje ensamblador depende del traductor del proceso de ensamblaje (assembler) utilizado, en este caso, se utiliza el assembler NASM.

- Ejemplo sum1toN.asm de programa en lenguaje ensamblador intel y assembler "NetWide Asm" (nasm).
- Tutorial completo NASM tutorialspoint
- Apuntes de la Universidad de Bristol
- Apuntes del Dr. Paul Carter

5.2.1. GNU Assembly (Gas)

- Lenguaje desarrollado por la empresa de telefonía AT&T
- Assembler gas (GNU as)
  - arquitecturas: i386, amd64, mips, 68000, etc
  - Sintaxis: Etiqueta-Código de Operación- Operando1- Operando2- Comentario

Cuadro 3: Sintaxis AT&T

label:	op_mnemonic	operand_source	,	operand_destination	;comment

- Ejemplo:
  - `bucle: subq $16,%rsp ;comienzo del bucle`
  - `suma: addl %esi,%eax ;sumar`
  - + movw %ax,resultado ;salvar resultado+

- ETIQUETA
  - Se especifica en la primera columna. Tiene el sufifjo **:**
- CODIGO DE OPERACION: Se utilizan símbolos *mnemónicos* que ayudan a interpretar intuitivamente la operación. Pej: ADD sumar, MOV mover, SUB restar, ...
- OPERANDO FUENTE Y/O DESTINO
  - dato alfanumérico: representación alfanumérica → 16
    - direccionamiento *inmediato*: prefijo $
  - dirección de memoria externa: etiqueta → resultado
    - direccionamiento *directo*
  - registros internos de la CPU: %rax, %rbx, %rsp, %esi,.. → %rsp, %esi
    - El prefijo % significa que el nombre hace referencia a un registro
  - tamaño del dato operando: sufijos de los mnemónicos: q(quad):8 bytes, l(long):4 bytes, w(word):2 bytes, b(byte):1 byte.

---

nota
La sintaxis del lenguaje ensamblador depende del traductor del proceso de ensamblaje (assembler) utilizado, en este caso, se utiliza el assembler GAS.

5.3. Traductores del proceso de ensamblaje

- Existen dos lenguajes ensamblador para la ISA i386/amd64, dos sintaxis diferentes, que deben de ser traducidos por diferentes traductores de ensamblador produciendo un módulo binario en el mismo lenguaje máquina.
- los traductores de ensamblador "NASM" (Netwide Asm), "FASM", "MASM", "TASM", and "YASM" traducen módulos fuente que utilizan la sintaxis del lenguaje ensamblador **intel** a modulos binarios para las arquitecturas i386/amd64
- El traductor "as" de la fundación GNU traduce módulos fuente que utilizan las sintaxis del lenguaje de ensamblador **AT&T** a módulos binarios para las arquitecturas i386/amd64
- La traducción entre la representación del módulo fuente en lenguaje ensamblador almacenado en un fichero del disco duro y la representación del programa en lenguaje máquina se da en el proceso de ensamblaje:
  - Módulo Fuente en lenguaje ensamblador → Traductor Ensamblador → Módulo Objeto Reubicable → Linker → Módulo Objeto Ejecutable Binario→ Cargador → Proceso
    - Ejemplo: hola_mundo.s → **as** → hola_mundo.o → **ld** → hola_mundo (formato ELF) → **loader** → hola_mundo (memoria principal) → creación proceso (hola_mundo en ejecución). "as", "ld" y "loader" son las herramientas GNU necesarias para la creación de un proceso a partir del módulo en lenguaje ensamblador.
    - el linker **ld** mezcla el módulo objeto con módulos del entorno del sistema operativo y con módulos de las librerías.
  - Apéndice B del libro de texto.

5.4. Código Máquina

5.4.1. Almacenamiento en Memoria

- Una vez realizado el proceso de traducción del módulo fuente en lenguaje ensamblador se genera un módulo objeto en lenguaje binario que se almacena en el disco duro en forma de fichero.
- El fichero que contiene el módulo objeto ejecutable en lenguaje binario es necesario cargarlo en la memoria principal. Esta tarea la realiza el **cargador** del sistema operativo.
- Cada dirección de memoria apunta a 1 byte.
- La dirección más baja apunta a todo el objeto: instrucción o dato.
- Ejemplo:
  - **4001a4: 48 83 ec 10**
    - En la posición **0x4001A4** está el byte **48**
    - En la posición **0x4001A4+1** está el byte **83**
    - En la posición **0x4001A4+2** está el byte **EC**
    - En la posición **0x4001A4+3** está el byte **10**
    - En la posición de memoria principal `0x4001A4` está almacenada la instrucción de 4 Bytes 48 83 ec 10.

5.4.2. Interpretación del Código Máquina

- Ejemplo:
  - instrucción máquina arquitectura amd64.
    - **4001a4: 48 83 ec 10 → subq $16, %rsp**
  - Interpretación del programador:
    - lenguaje ensamblador AT&T de la arquitectura x86.
    - Descripción de la instrucción en lenguaje **RTL**: *RSP ← RSP - 16*
    - En la posición de memoria principal 0x4001A4 está almacenada la instrucción `subq $16,%rsp`
    - subq indica la operación de resta con datos enteros de 64 bits (sufijo q). Resta del operando destino el operando fuente.
    - El operando fuente tiene valor decimal 16, 0x10 en hexadecimal y el direccionamiento de este operando es inmediato, es decir, su valor es 16 y está ubicado en la propia instrucción.
    - El operando destino está almacenado en el registro interno de la CPU denominado RSP
    - La referencia a la Próxima Instrucción la realiza no la instrucción sino la CPU realizando la opeación PC←PC+tamaño de la instrucción en bytes.
  - ¿Cómo interpretar una instrucción máquina en lenguaje binario ? Es necesario consultar el **Manual de Referencia de la Arquitectura ISA de la máquina x86** y tener conocimientos de los modos de direccionamiento. Este ejemplo se desarrolla al final de este Tema en el apartado 8) *Formato de Instrucción de la arquitectura ISA x86*

5.5. Assembler "as"

5.5.1. Directivas

- Directivas del traductor ensamblador "as" utilizado por el sistema GNU/linux para el lenguaje ensamblador AT&T.
  - Al assembler de GNU también se el conoce como "gas".
  - binutils → as assembler: manual oficial
  - gas ref card
    - .word reserva 2 bytes en amd64.
  - oracle
  - opciones para x86 y x86-64

5.5.2. Manual

- binutils as: manual oficial
- Linux Assembly howto: referencias a diferentes assemblers
- MIT

6. Representación de los datos en lenguaje ensamblador (ASM) para la arquitectura i386/amd64

6.1. Tipos de Datos

6.1.1. Números y Caracteres

- Número sin signo (naturales): codificación binario natural
- Números enteros con signo: entero codificados en complemento a 2
- Números reales reales codificados en formato IEEE-754 de simple o doble precisión
- Caracteres alfanuméricos: código ASCII

6.1.2. Directivas de la Sección de Datos

importante
Recomendable leerse los seis primeros apartados por lo menos y sobre todo lo referente a las directivas dependientes de la arquitectura x86

Cuadro 4: Directivas básicas

Directivas	**descripción**
.global o .globl etiqueta	variables globales
.section .data	sección de las variables locales estáticas inicializadas
.section .text	sección de las instrucciones
.section .bss	sección de las variables sin inicializar
.section .rodata	sección de las variables de sólo lectura
.type name , type description	tipo de variable, p.ej @function
Et: .common 100	reserva 100 bytes sin inicializar y puede ser referenciado globalmente
Et: .lcomm bucle, 100	reserva 100bytes referenciados con el símbolo local bucle. Sin inicializar.
Et: .space 100	reserva 100 bytes inicializados a cero
Et: .space 100, 3	reserva 100 bytes inicializados a 3
Et: .string "Hola"	añade el byte 0 al final de la cadena
Et: .asciz "Hola"	añade el byte 0 al final de la cadena
Et: .ascii "Hola"	no añade le caracter NULL de final de cadena
Et: .byte 3,7,-10,0b1010,0xFF,0777	tamaño 1Byte y formatos decimal,decimal,decimal,binario,hexadecimal,octal
Et: .2byte 3,7,-10,0b1010,0xFF,0777	tamaño 2Bytes
Et: .word 3,7,-10,0b1010,0xFF,0777	tamaño 2Bytes
Et: .short 3,7,-10,0b1010,0xFF,0777	tamaño 2B
Et: .4byte 3,7,-10,0b1010,0xFF,0777	tamaño 4B
Et: .long 3,7,-10,0b1010,0xFF,0777	tamaño 4B
Et: .int 3,7,-10,0b1010,0xFF,0777	tamaño 4B
Et: .8byte 3,7,-10,0b1010,0xFF,0777	tamaño 8B
Et: .quad 3,7,-10,0b1010,0xFF,0777	tamaño 8B
Et: .octa 3,7,-10,0b1010,0xFF,0777	formato octal
Et: .double 3.14159, 2 E-6	precisión doble
Et: .float 2E-6, 3.14159	precisión simple
Et: .single 2E-6	precisión simple

Cuadro 4: (continued)

Directivas	**descripción**
.include "file"	incluye el fichero . Obligatorias las comillas.
.equ SUCCESS, 0	macro que asocia el símbolo SUCCESS al número 0
.macro macname macargs	define el comienzo de una macro de nombre macname y argumentos macargs
.endmacro	define el final de una macro
.align n	las instrucciones o datos posteriores empezarán en una dirección multiplo de n bytes.
.end	fin del ensamblaje

- Et: Etiqueta

6.2. Tamaño del operando x86

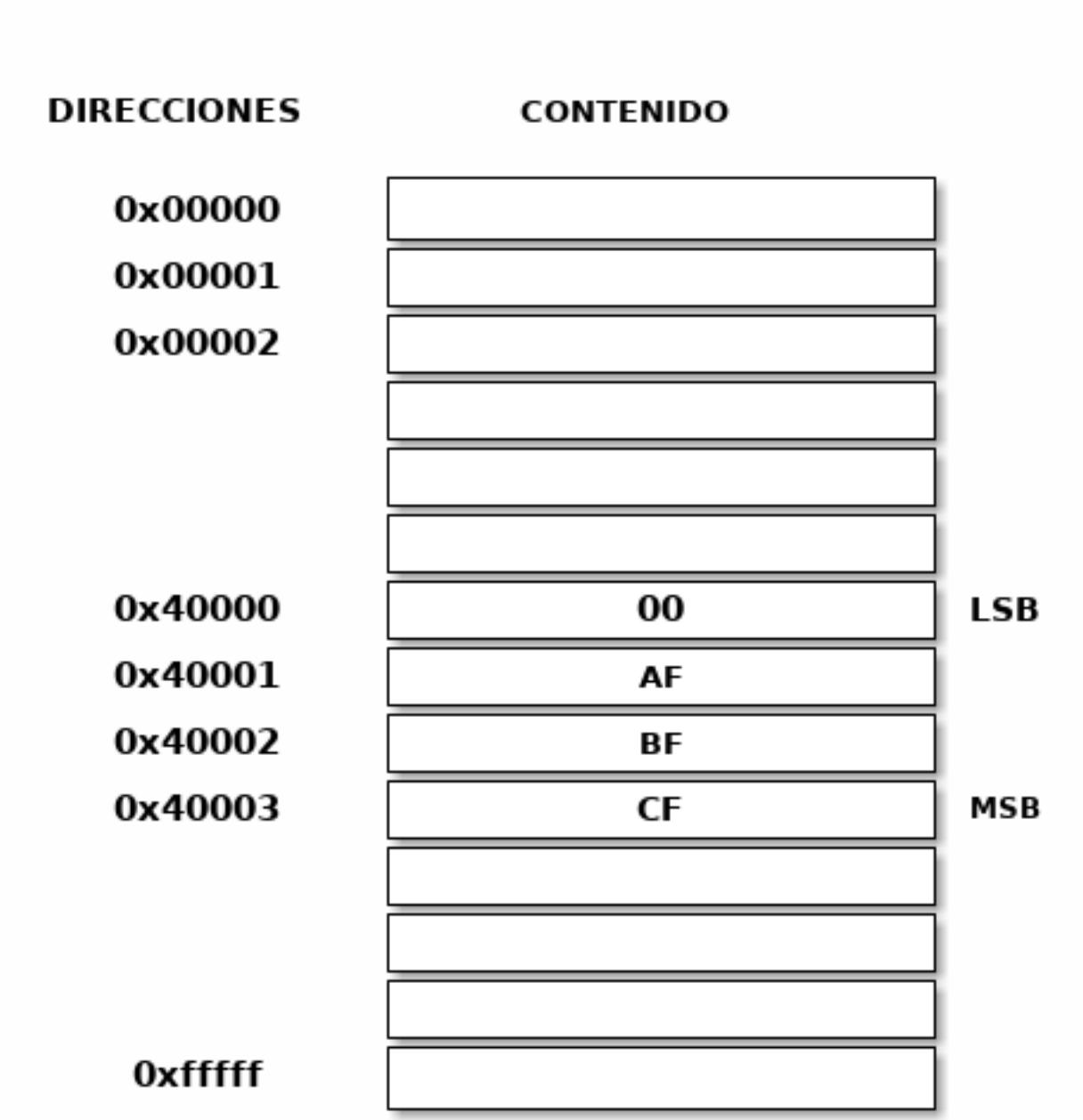
- Tamaño del operando: sufijos de los MNEMÓNICOS.

```
q (quad) 8bytes
l (long) 4bytes
w (word) 2bytes
b (byte) 1byte
```

  - Ejemplos:
    - movq %rax,resultado
    - movl %eax,resultado
    - movw %ax,resultado
    - movb %ah,resultado

6.3. Alineamiento de Bytes: Big-Little Endian

- Los bytes de un dato de varios bytes se pueden almacenar en memoria en sentido MSB-_- LSB ó MSB-_-LSB
- Alineamiento *Little Endian*: El byte de menor peso LSB se almacena en la posición de memoria más baja
- Ejemplo **0x40000: 00 AF BF CF**
  - En la posición de memoria principal 0x40000 está almacenado el dato de 4 bytes: `00 AF BF CF`
    - Los bytes se guardan en dirección de memoria ascendente. Cuando se escribe en horizontal, ascendente significa de izda a dcha.
    - En la posición **0x40000** está el byte 00 → **LSB** (Least Significant Byte)
    - En la posición **0x40001** está el byte AF
    - En la posición **0x40002** está el byte BF
    - En la posición **0x40003** está el byte CF → **MSB** (Most Significant Byte)



- El byte de menor peso se almacena en la posición de memoria más baja. La posición más baja de las cuatro es la 0x4000 donde se almacena el 00, luego este es el byte de menor peso. El dato almacenado en formato little-endian es el **0xCFBFAF00**.
- La arquitectura i386/amd64 utiliza LITTLE ENDIAN

- Tipos de información que siguen el formato little endian.
  - Para las instrucciones el formato es por campos por lo que no tiene sentido hablar de posiciones de mayor o menor peso de la instrucción por lo que no siguen el formato little endian.
  - Las cadenas de caracteres (strings) no representan un valor y por lo tanto no siguen el formato little endian.
  - Los números enteros se almacenan siguiendo el formato little endian.
  - Los números reales se almacenan siguiendo el formato little endian
  - Las direcciones de memoria se almacenan siguiendo la organización Little Endian.

- Formato Big Endian
  - El orden de almacenamiento es el inverso al little endian, es decir, el byte LSB del dato se almacena en la dirección de memoria mayor de la región que ocupa el dato.

7. Operandos: Modos de Direccionamiento

7.1. Localización

- Ejemplo:
  - `bucle: SUBQ $16,%rsp ;comienzo del bucle`
    - Operando fuente: $ indica direccionamiento INMEDIATO .El operando está en la propia instrucción → Operando=16
    - Operando destino: % indica REGISTRO. El operando está en el registro RSP
  - `suma: ADDW (%ESI),resultado ;fin de operación`
    - Operando fuente: () indica INDIRECCION y % registro .El registro ESI continene la dirección de memoria donde está el operando
    - Operando destino: "resultado" es una etiqueta. Direccionamiento ABSOLUTO. El operando está en la dirección de memoria "resultado".

7.2. Modos de Direccionamiento

- Manual del assembler, apartado directivas dependientes de la arquitectura x86
  - https://sourceware.org/binutils/docs-2.26/as/i386_002dDependent.html#i386_002dDependent:

importante
RECOMENDABLE leerse los seis primeros apartados por lo menos

- Direccionamientos:

INMEDIATO:	El valor del operando está ubicado inmediatamente después del código de operación de la instruccion. Unicamente se especifica el operando fuente.
	sintaxis: el valor del operando se indica con el prefijo **$** .
	ejemplo: **movl $0xabcd1234,%ebx.** El operando fuente es el valor 0xABCD1234
REGISTRO:	El valor del operando está localizado en un registro de la CPU.
	sintaxis: Nombre del registro con el prefijo**%.**
	ejemplo: **movl%eax,%ebx.** El operando fuente es el REGISTRO EAX y el destino es el REGISTRO EBX
DIRECTO:	La dirección efectiva apuntando al operando almacenado en la Memoria Principal es la dirección absoluta referenciada por la etiqueta especificada en el campo de operando. El programador utiliza el direccionamiento directo pero el compilador lo transforma en un direccionamiento relativo al contador de programa. Ver direccionamiento con desplazamiento.
	sintaxis: una etiqueta definida por el programador
	ejemplo: **je somePlace** . Salto a la dirección marcada por la etiqueta somePlace si el resultado de la operación anterior activa el flag ZF=1 del registro RFLAG.
INDEXADO:	El valor del operando está localizado en memoria. La dirección efectiva apuntando a Memoria es la SUMA del valor del registro_base MAS scale POR el valor en el registro_índice, MAS el offset. *EA=Offset+R_Base+R_índice*Scale*

	sintaxis: lista de valores separados por coma y entre paréntesis (base_register, index_register, scale) y precedido por un offset.
	ejemplo: **movl $0x6789cdef, -16(%edx,%eax, 4)** . La dirección efectiva del destino es EDX + EAX*4 - 16.
INDIRECTO:	Si el modo general de indexación lo particularizamos en (base_register) entonces la dirección del operando no se obtiene mediante una indexación sino que la dirección efectiva es el contenido de rdx y por lo tanto se accede al operando indirectamente.
	sintaxis: (base_register)
	ejemplo: **movl $0x6789cdef, (%edx)** . La dirección efectiva del destino es EDX. EDX es un puntero.
RELATIVO: registro base más un offset:	El valor del operando está ubicado en memoria. La dirección efectiva del operando es la suma del valor contenido en un registro base más un valor de offset.
	sintaxis: registro entre paréntesis y el offset inmediatamente antes del paréntesis.
	ejemplo: **movl $0xaabbccdd, -12(%eax)** . La dirección efectiva del operando destino es EAX-12

7.2.1. Ejemplos

Cuadro 5: Modos de Direccionamiento de los Operandos

Direccionamiento Operando	Valor Operando	Nombre del Modo
$0	Valor Cero	Inmediato
%rax	RAX	Registro
loop_exit	M[loop_exit]	Directo
data_items(,%rdi,4)	M[data_item + 4*RDI]	Indexado
(%rbx)	M[RBX]	Indirecto
(%rbx,%rdi,4)	M[RBX + 4*RDI]	Indirecto Indexado

- M[loop_exit]: directo ya que loop_exit es una dirección de memoria externa y M indica la memoria externa.
- M[RBX]: indirecto ya que RBX es una dirección de memoria interna y M indica memoria externa: A la mem. externa se accede a través de la mem. interna.

8. Repertorio de Instrucciones: Operaciones

8.1. Manuales de referencia

8.1.1. Lenguaje Intel

- Manuales oficiales
  - Intel: Vol 2
  - AMD: apartado Manuals : vol 3
  - binutils:
    - **as: traductor assembler**: opciones de la línea de comandos, directivas, tipos de datos, etc

  - ld: linker
  - objdump, readelf, ...

- Manuales no oficiales:

  - manual Intel quick: **recomendado**
  - intel descriptivo i386
    - Repertorio ISA y Formato de Instrucción
  - AT&T Solaris Manual amd64-i386 : lenguaje y traductor assembler.
  - Saltos Condicionales

8.1.2. lenguaje AT&T

- Oracle Solaris ASM

  - En este documento a la sintaxis AT&T la denomina "Oracle Solaris".

8.2. TRANSFERENCIA

Nombre	Comentario	Código	Operación	O D I T S Z A P C
MOV	Mover (copiar)	MOV Fuente,Dest	Dest:=Fuente	
XCHG	Intercambiar	XCHG Op1,Op2	Op1:=Op2 , Op2:=Op1	
STC	Set the carry (Carry = 1)	STC	CF:=1	1
CLC	Clear Carry (Carry = 0)	CLC	CF:=0	0
CMC	Complementar Carry	CMC	CF:=Ø	±
STD	Setear dirección	STD	DF:=1(interpreta strings de arriba hacia abajo)	1
CLD	Limpiar dirección	CLD	DF:=0(interpreta strings de abajo hacia arriba)	0
STI	Flag de Interrupción en 1	STI	IF:=1	1
CLI	Flag de Interrupción en 0	CLI	IF:=0	0
PUSH	Apilar en la pila	PUSH Fuente	DEC SP, [SP]:=Fuente	
PUSHF	Apila los flags	PUSHF	O, D, I, T, S, Z, A, P, C 286+: También NT,IOPL	
PUSHA	Apila los registros generales	PUSHA	AX, CX, DX, BX, SP, BP, SI, DI	
POP	Desapila de la pila	POP Dest	Destino:=[SP], INC SP	
POPF	Desapila a los flags	POPF	O,D,I,T,S,Z,A,P,C 286+: También NT,IOPL	± ± ± ± ± ± ± ± ±
POPA	Desapila a los reg. general.	POPA	DI, SI, BP, SP, BX, DX, CX, AX	
CBW	Convertir Byte a Word	CBW	AX:=AL (con signo)	
CWD	Convertir Word a Doble	CWD	DX:AX:=AX (con signo)	
CWDE	Conv. Word a Doble Exten.	CWDE 386	EAX:=AX (con signo)	
IN	Entrada	IN Dest,Puerto	AL/AX/EAX := byte/word/double del puerto esp.	
OUT	Salida	OUT Puer,Fuente	Byte/word/double del puerto := AL/AX/EAX	

- Flags: $\pm$ =Afectado por esta instrucción, **?** =Indefinido luego de esta instrucción

8.3. ARITMÉTICOS

Nombre	Comentario	Código	Operación	O D I T S Z A P C
ADD	Suma	ADD Fuente,Dest	Dest:=Dest+ Fuente	± ± ± ± ± ±
ADC	Suma con acarreo	ADC Fuente,Dest	Dest:=Dest+ Fuente +CF	± ± ± ± ± ±
SUB	Resta	SUB Fuente,Dest	Dest:=Dest- Fuente	± ± ± ± ± ±
SBB	Resta con acarreo	SBB Fuente,Dest	Dest:=Dest-(Fuente +CF)	± ± ± ± ± ±
DIV	División (sin signo)	DIV Op	Op=byte: AL:=AX / Op AH:=Resto	? ? ? ? ? ?
DIV	División (sin signo)	DIV Op	Op=word: AX:=DX:AX / Op DX:=Resto	? ? ? ? ? ?
DIV	386 División (sin signo)	DIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Resto	? ? ? ? ? ?
IDIV	División entera con signo	IDIV Op	Op=byte: AL:=AX / Op AH:=Resto	? ? ? ? ? ?
IDIV	División entera con signo	IDIV Op	Op=word: AX:=DX:AX / Op DX:=Resto	? ? ? ? ? ?
IDIV	386 División entera con signo	IDIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Resto	? ? ? ? ? ?
MUL	Multiplicación (sin signo)	MUL Op	Op=byte: AX:=AL*Op si AH=0 #	± ? ? ? ? ±
MUL	Multiplicación (sin signo)	MUL Op	Op=word: DX:AX:=AX*Op si DX=0 #	± ? ? ? ? ±
MUL	386 Multiplicación (sin signo)	MUL Op	Op=double: EDX:EAX:=EAX*Op si EDX=0 #	± ? ? ? ? ±
IMUL	i Multiplic. entera con signo	IMUL Op	Op=byte: AX:=AL*Op si AL es suficiente #	± ? ? ? ? ±
IMUL	Multiplic. entera con signo	IMUL Op	Op=word: DX:AX:=AX*Op si AX es suficiente #	± ? ? ? ? ±
IMUL	386 Multiplic. entera con signo	IMUL Op	Op=double: EDX:EAX:=EAX*Op si EAX es sufi. #	± ? ? ? ? ±
INC	Incrementar	INC Op	Op:=Op+1 (El Carry no resulta afectado !)	± ± ± ± ±
DEC	Decrementar	DEC Op	Op:=Op-1 (El Carry no resulta afectado !)	± ± ± ± ±
CMP	Comparar	CMP	Op1,Op2 Op1-Op2	± ± ± ± ± ±
SAL	Desplazam. aritm. a la izq.	SAL	Op,Cantidad	i ± ± ? ± ±
SAR	Desplazam. aritm. a la der.	SAR	Op,Cantidad	i ± ± ? ± ±
RCL	Rotar a la izq. c/acarreo	RCL Op,Cantidad		i ±
RCR	Rotar a la derecha c/acarreo	RCR Op,Cantidad		i ±
ROL	Rotar a la izquierda	ROL Op,Cantidad		i ±
ROR	Rotar a la derecha	ROR Op,Cantidad		

- **i**:para más información ver especificaciones de la intrucción,
- **#**:entonces CF:=0, OF:=0 sino CF:=1, OF:=1

8.4. LÓGICOS

Nombre	Comentario	Código	Operación	O D I T S Z A P C
NEG	Negación (complemento a 2)	NEG Op	Op:=0-Op si Op=0 entonces CF:=0 sino CF:=1	± ± ± ± ± ±
NOT	Invertir cada bit	NOT Op	Op:=Ø~Op (invierte cada bit)	
AND	*Y* (And) lógico	AND Fuente,Dest	Dest:=Dest ^ Fuente	0 ± ± ? ± 0
OR	*O* (Or) lógico	OR Fuente,Dest	Dest:=Dest v Fuente	0 ± ± ? ± 0
XOR	*O* (Or) exclusivo	XOR Fuente,Dest	Dest:=Dest (xor) Fuente	0 ± ± ? ± 0
SHL	Desplazam. lógico a la izq.	SHL Op,Cantidad		i ± ± ? ± ±
SHR	Desplazam. lógico a la der.	SHR Op,Cantidad		i ± ± ? ± ±

8.5. MISCELÁNEOS

Nombre	Comentario	Código	Operación	O D I T S Z A P C
NOP	Hacer nada	NOP	No hace operación alguna	
LEA	Cargar direciión Efectiva	LEA Fuente,Dest	Dest := dirección fuente	
INT	Interrupción	INT Num	Interrumpe el proceso actual y salta al vector Num	0 0

8.6. SALTOS (generales)

- wiki x86 assembly

```
Nombre  Comentario                    Código          Operación
CALL    Llamado a subrutina           CALL Proc
JMP     Saltar                        JMP Dest
JE      Saltar si es igual            JE Dest         (= JZ)
JZ      Saltar si es cero             JZ Dest         (= JE)
JCXZ    Saltar si CX es cero          JCXZ Dest
JP      Saltar si hay paridad         JP Dest         (= JPE)
JPE     Saltar si hay paridad par     JPE Dest        (= JP)
JPO     Saltar si hay paridad impar   JPO Dest        (= JNP)
JNE     Saltar si no es igual         JNE Dest        (= JNZ)
JNZ     Saltar si no es cero          JNZ Dest        (= JNE)
JECXZ   Saltar si ECX es cero         JECXZ Dest 386
JNP     Saltar si no hay paridad      JNP Dest        (= JPO)
RET     Retorno de subrutina          RET
```

8.7. SALTOS Sin Signo (Cardinal) SALTOS Con Signo (Integer)

```
Nombre  Comentario                    Código          Operación
JA      Saltar si es superior         JA Dest         (= JNBE)
JAE     Saltar si es superior o igual JAE Dest        (= JNB = JNC)
JB      Saltar si es inferior         JB Dest         (= JNAE = JC)
JBE     Saltar si es inferior o igual JBE Dest        (= JNA)
JNA     Saltar si no es superior      JNA Dest        (= JBE)
JNAE    Saltar si no es super. o igual JNAE Dest      (= JB = JC)
JNB     Saltar si no es inferior      JNB Dest        (= JAE = JNC)
JNBE    Saltar si no es infer. o igual JNBE Dest      (= JA)
JC      Saltar si hay carry JC Dest   JO  Dest        Saltar si hay Overflow
```

```
JNC     Saltar si no hay carry          JNC Dest
JNO     Saltar si no hay Overflow       JNO Dest
JS      Saltar si hay signo (=negativo) JS Dest
JG      Saltar si es mayor              JG Dest         (= JNLE)
JGE     Saltar si es mayor o igual      JGE Dest        (= JNL)
JL      Saltar si es menor              JL Dest         (= JNGE)
JLE     Saltar si es menor o igual      JLE Dest        (= JNG)
JNG     Saltar si no es mayor           JNG Dest        (= JLE)
JNGE    Saltar si no es mayor o igual   JNGE Dest       (= JL)
JNL     Saltar si no es inferior        JNL Dest        (= JGE)
JNLE    Saltar si no es menor o igual   JNLE Dest       (= JG)
```

8.8. FLAGS (ODITSZAPC)

```
O: Overflow resultado de operac. sin signo es muy grande o pequeño.
D: Dirección
I: Interrupción Indica si pueden ocurrir interrupciones o no.
T: Trampa Paso, por paso para debugging
S: Signo Signo del resultado. Razonable sólo para enteros. 1=neg. 0=pos.
Z: Cero Resultado de la operación es cero. 1=Cero
A: Carru Aux. Similar al Carry, pero restringido para el nibble bajo únicamente
P: Paridad 1=el resultado tiene cantidad par de bits en uno
C: Carry resultado de operac. sin signo es muy grande o inferior a cero
```

8.9. Sufijos

- Sufijos de los mnemónicos del código de operación:
  - *q* : quad: operando de 8 bytes: cuádruple palabra
  - *l* : long: operando de 4 bytes: doble palabra
  - *w* : word: operando de 2 bytes: palabra
  - *b* : byte: operando de 1 byte
- Si el mnemónico de operación no lleva sufijo el tamaño por defecto del operando es *long*

8.10. Códigos de Operación

- MOV

```
MOV -- Move Data

Opcode   Instruction       Clocks        Description

88  /r   MOV r/m8,r8       2/2           Move byte register to r/m byte
89  /r   MOV r/m16,r16     2/2           Move word register to r/m word
89  /r   MOV r/m32,r32     2/2           Move dword register to r/m dword
8A  /r   MOV r8,r/m8       2/4           Move r/m byte to byte register
8B  /r   MOV r16,r/m16     2/4           Move r/m word to word register
8B  /r   MOV r32,r/m32     2/4           Move r/m dword to dword register
8C  /r   MOV r/m16,Sreg    2/2           Move segment register to r/m word
8D  /r   MOV Sreg,r/m16    2/5,pm=18/19  Move r/m word to segment register
A0       MOV AL,moffs8     4             Move byte at (seg:offset) to AL
A1       MOV AX,moffs16    4             Move word at (seg:offset) to AX
A1       MOV EAX,moffs32   4             Move dword at (seg:offset) to EAX
A2       MOV moffs8,AL     2             Move AL to (seg:offset)
A3       MOV moffs16,AX    2             Move AX to (seg:offset)
```

```
A3        MOV moffs32,EAX   2             Move EAX to (seg:offset)
B0 + rb   MOV reg8,imm8     2             Move immediate byte to register
B8 + rw   MOV reg16,imm16   2             Move immediate word to register
B8 + rd   MOV reg32,imm32   2             Move immediate dword to register
C6        MOV r/m8,imm8     2/2           Move immediate byte to r/m byte
C7        MOV r/m16,imm16   2/2           Move immediate word to r/m word
C7        MOV r/m32,imm32   2/2           Move immediate dword to r/m dword
```

- dword :double word: 32 bits

atención
la sintaxis del lenguage ASM no es AT&T sino Intel → mnemónico operando_destino, operando fuente

---

Programas Ejemplo

9. Mnemónicos Básicos (Explicados)

9.1. Operaciones aritméticas

- `mul`: multiplicación de números naturales, sin signo

```
Performs an unsigned multiplication of the first operand (destination operand) and the ↩
    second operand (source operand) and stores the result in the destination operand. The ↩
    destination operand is an implied operand located in register AL, AX or EAX (depending ↩
     on the size of the operand); the source operand is located in a general-purpose ↩
    register or a memory location
```

- `mul`: multiplicación de números enteros con signo
  - Puede tener 1,2 o 3 operandos

```
Performs a signed multiplication of two operands. This instruction has three forms, ↩
    depending on the number of operands.

    One-operand form -- This form is identical to that used by the MUL instruction. Here ↩
        , the source operand (in a general-purpose register or memory location) is ↩
        multiplied by the value in the AL, AX, EAX, or RAX register (depending on the ↩
        operand size) and the product (twice the size of the input operand) is stored in ↩
         the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.
```

  - +imull Etiqueta : R[%edx]:R[%eax] ← M[Etiqueta] × R[%eax]

- `div`: división de números naturales, sin signo
- `idiv`:

```
Divides the (signed) value in the AX, DX:AX, or EDX:EAX (dividend) by the source operand ( ↩
    divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The ↩
    source operand can be a general-purpose register or a memory location. The action of ↩
    this instruction depends on the operand size (dividend/divisor).

Non-integral results are truncated (chopped) towards 0
```

  - idivl

- EAX←Cociente{[EDX:EAX]/M[Op_fuente]} , EDX←Resto{[EDX:EAX]/M[Op_fuente]}

- Extensión de signo
  - movsbw src,Reg → Mov Sign Byte to Word
  - movsbl src,Reg → Mov Sign Byte to Long
  - movswl rc,Reg → Mov Sign Word to Long
- Cambio de tamaño
  - movzbw src,Reg → Mov Byte to Word
  - movzbl src,Reg → Mov Byte to Long
  - movzwl src,Reg → Mov Byte to Long

9.2. Procesamiento Condicional

9.2.1. Boolean & Comparación

- NOT
  - no flags
- AND
  - Clear CF,OF
  - Modifica SF,ZF,PF
- OR
  - Clear CF,OF
  - Modifica SF,ZF,PF
- XOR
  - Clear CF,OF
  - Modifica SF,ZF,PF
- TEST
  - Clear CF,OF
  - Modifica SF,ZF,PF
- CMP
  - Modifica CF,OF,SF,ZF,PF,AF

9.3. Saltos

9.3.1. Indirectos

- Símbolo *para indicar indirección en los saltos y diferenciarlos del direccionamiento relativo. En cambio en los movimientos MOV no hace falta el símbolo* ya que no hay saltos relativos.

```
jmp bucle   -> salto relativo a EIP
jmp *bucle
jmp *eax
jmp *(eax)
jmp *(mem)
jmp *table(%ebx,%esi,4)
```

9.4. Desplazamiento y rotación

- `sar`,`sal` : Shift Arithmetic Right, Shift Arithmetic Left.
  - desplazamiento aritmético: El digito entrante por la izda o dcha es el bit de signo.

```
Shifts the bits in the first operand (destination operand) to the left or right by the ↩
    number of bits specified in the second operand (count operand). Bits shifted beyond ↩
    the destination operand boundary are first shifted into the CF flag, then discarded. ↩
     At the end of the shift operation, the CF flag contains the last bit shifted out of ↩
     the destination operand.
```

  - `sarl $31, %edx` : desplazamiento de 31 bits a la dcha y el bit entrante es el bit de signo el operando en EDX.
- `shr`,`shl`
- desplazmiento lógico: entran ceros
  - Ejemplos de multiplicación y división
- ROL,ROR
  - el bit que sale fuera se copia en CF
  - Aplicación: conversión endianess

9.5. Cambiar el Endianess

```
## Cambio del endianess en EAX. Previamente guarda el original de EAX y al final restaura ↩
    EAX
swapbytes:
       xchg (%ebx), %eax
       bswap %eax
       xchg (%ebx), %eax
```

10. Formato de Instrucción: ISA Intel x86-64

- Apéndice Formato Instrucción

11. Subrutinas

11.1. Introducción

- Las subrutinas en lenguaje ensamblador son el equivalente a las funciones en el lenguaje de programación en C, por lo que es necesario repasar el concepto de función en el lenguaje C.
- Referencias a las subrutinas son la práctica 5 y el capítulo 14 del Apéndice.

11.2. Lenguaje C: Sentencia Función

11.2.1. Introducción

El objetivo de las funciones es descomponer el programa en módulos de código para dotar al programa de una estructura organizada que facilite el desarrollo del programa y su mantenimiento. La librería standard "libc" son colecciones de funciones básicas desarrolladas en el lenguaje C que son reutilizadas por la mayoría de los programas. De esta manera el programador no tiene que inventar la rueda. Por lo tanto en un programa coexisten funciones desarrolladas por el propio usuario en lenguaje C y funciones de librerías accesibles en código binario.

11.2.2. Declaración

- La declaración de una función en lenguaje C se denomina **prototipo**. Ejemplo de prototipo: `int sumMtoN(short sumando1, short sumando2)` donde
  - Nombre: el nombre de la función es *sumMtoN*
  - Argumentos: el nombre del primer argumento es *sumando1* y es del tipo short, el nombre del 2º argumento es *sumando2* y es del tipo shrot.
  - Tipo del valor de retorno: el tipo del valor de retorno es int.

11.2.3. Definición

- La definición de la función *sumMtoN* consiste en desarrollar el algoritmo mediante sentencias de C, es decir, el cuerpo de la función:

```
int sumMtoN(short sumando1, short sumando2)   {
      //sumando2 > sumando1
      short i;
      int resultado=0; // variable local a la función
      i=sumando2;
        while (i >= sumando1) {
          resultado += i ;
          i--;
        }
      printf("\n\t Subrutina sumMtoN \n");
      return resultado;
    }
```

  - resultado es la variable que contiene el valor de retorno

11.2.4. LLamada y Retorno

- La función *main()* llama a la función *sumMtoN()* la cual después de ser ejecutada devuelve el resultado de la suma.

```
/*
   Programa: sumMtoN.c
   Compilación: gcc -g -ggdb3 -o sumMtoN sumMtoN.c
                -ggdb3 : inserta en la tabla de símbolos del depurador información de  ↩
                   macros
*/

// Prototipos de las funciones
#include <stdio.h>  // Declaración de la función printf()
#include <stdlib.h>  // Declaración de la función exit()

//Macros
#define SUCCESS   0

//Prototipos: declaración de la función sumMtoN()
int sumMtoN(short sumando1, short sumando2);

// Definición de la Función Principal main()
void main(void) {
       //Inicialización de los argumentos M y N de la función sumMtoN()
       short M=1;
       short N=1;
       // Llamada a la función
       printf("El resultado de la suma es %d \n", sumMtoN(M,N));
       // La evaluación de sumMtoN consiste en:  llamar a la función y capturar el valor  ↩
           de retorno.
       exit(SUCCESS);
     }

// Definición de las Funciones
int sumMtoN(short sumando1, short sumando2)   {
      //sumando2 > sumando1
      short i;
      int resultado=0; // variable local a la función
      i=sumando2;
        while (i >= sumando1) {
          resultado += i ;
          i--;
        }
      printf("\n\t Subrutina sumMtoN \n");
      return resultado;
    }
```

- printf → sumMtoN : printf imprime el resultado de **evaluar** la función *sumMtoN()* . La evaluación consiste en obtener el **valor de retorno** de la ejecución de la función *sumMtoN()*

11.3. Anidamiento de Funciones

- LLamada: init() → main() → sumMtoN() → printf() → write()
- El sistema operativo llama a la función principal ,del programador, que a su vez llama a la función sumMtoN() ,del programador, la cual a su vez llama a la función printf() , de la librería libc, la cual a su vez llama al sistema operativo para que ejecute la función write(), del sistema.
- Retorno: write() → printf() → sumMtoN() → main () → exit()

11.4. Pila/Frame

- Ver concepto de pila en el Apéndice F.
- La pila es una *sección* del programa en la memoria principal como lo son la sección de datos y la sección de instrucciones.
- Los argumentos M y N de la subrutina *sumMtoN* se pasan a través de la pila.
- Partición de la pila en frames: Cada rutina y subrutina tienen su *segmento* de pila que se denomina **frame**.
  - La rutina *main* tiene su frame y la subrutina *sumMtoN* su propio frame.
  - Los frames se apilan según se anidan las llamadas a subrutinas.
  - Dinamismo: En un momento dado de la ejecución del programa el último frame generado es el frame activo.
    - La parte baja del frame activo esta referenciada por el puntero EBP y el top del frame por el puntero ESP.

11.5. Definición de la subrutina

- Nombre: *sumMtoN*
- El nombre de la subrutina es la etiqueta que apunta a la primera instrucción de la subrutina.
- La subrutina finaliza con la instrucción ret.
- La subrutina está estructurada en 3 partes:
  - Prólogo:
    I. Salvar los registros que van a ser modificados por el cuerpo de la subrutina.
    II. Activar el nuevo frame inicializando los punteros `EBP` y `ESP`.
  - Cuerpo:
    I. Capturar los argumentos y procesarlos
  - Epílogo:
    I. Salvar el valor de retorno en el registro `EAX`
    II. Recuperar el valor de los registros salvados en el Prólogo
    III. Activar el frame de la función que ha realizado la llamada actualizando `EBP` y `ESP` con sus antiguos valores.
    IV. Retorno a la función que ha realizado la llamada.
- Código

```
# Comienzo de la subrutina
sumMtoN:
# Prólogo
                push  %ebp      # salvo el bottom del frame de la función llamante en la  ↩
                    parte baja del nuevo frame
                mov   %esp,%ebp # configuro el puntero %ebp apuntando a la parte baja del  ↩
                    nuevo frame
                push   xxx      # Si fuera necesario: salvar registros que se utilizarán  ↩
                    en el Cuerpo de la subrutina
                push   xxx
# Cuerpo
                mov 8(%ebp),%rbx        #capturo el 1° argumento
                mov 16(%ebp),%rcx       #capturo el 2° argumento
                xxx xxx
                xxx xxx
# Epílogo
                mov resultado,%rax      #inicializo el valor de retorno
                pop   xxx               #recuperar registros que se salvaron en el prólogo
                pop   xxx
                mov %ebp,%esp
                pop %ebp
                ret
```

11.6. Registros a Preservar

11.6.1. Refs

- ABI x86-32
- Convenio de LLamada MicroSoft

11.6.2. Rutina llamante

La rutina que realiza la llamada (caller routine) está obligada a preservar los siguientes registros si los está utilizando:

- EAX-ECX-EDX

Es decir, dichos registros pueden ser utilizados libremente por la subrutina llamada. En caso de no ser utilizados por la subrutina no sería necesario preservarlos. En caso de ser utilizados se copiarían en la pila antes de realizar la llamada a la subrutina y serían recuperados al finalizar la subrutina.

11.6.3. Subrutina llamada

La subrutina llamada (callee routine) está obligada a preservar los siguientes registros:

- EBX-ESP-EBP-ESI-EDI y X87CW

Es decir, dichos registros al finalizar la subrutina de mantener el mismo valor que antes de la llamada. En caso de no utilizarlos no sería necesario preservarlos

11.6.4. Arquitectura amd64

Caller routine: The registers RAX, RCX, RDX, R8, R9, R10, R11 are considered volatile and must be considered destroyed on function calls (unless otherwise safety-provable by analysis such as whole program optimization).

Callee routine: The registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15 are considered nonvolatile and must be saved and restored by a function that uses them.

11.7. Argumentos de la subrutina

- Los argumentos deben de transferirse a través de la pila y antes de realizar la llamada.
- Los argumentos se apilan uno detrás de otro comenzando por el último argumento y finalizando con el primer argumento.
- Se apilan mediante la instrucción `push argumento` donde el operando es el argumento a transferir.

```
push N
push M
```

11.8. Llamada a la subrutina

- La rutina llamante *main* llama a la subrutina *sumMtoN* mediante la instrucción `call sumMtoN`. Por lo que la rutina *main* queda interrumpida hasta que finalice la ejecución de la subrutina *sumMtoN*.
- La instrucción `call` se ejecuta en dos fases:

    a. Apila la dirección de retorno: en la rutina *main* siguiente instrucción a `call sumMtoN`: **ESP** ← **ESP-4** y **M[ESP]** ← **PC**

b. Salta a la etiqueta *sumMtoN*: **PC ← sumMtoN**

- básicamente la instrucción call es una salto con retorno a la dirección donde fue interrumpida la rutina llamante.

```
push N
push M
call sumMtoN
```

11.9. Retorno de la subrutina

- La última instrucción de la subrutina es **RET** cuya ejecución por la Unidad de Control de la CPU realiza las siguientes órdenes:

  a. **PC ← M[ESP]** : extrae de la pila la dirección de retorno guardada por la instrucción **CALL** y la carga en el Contador de Programa, por lo que se ejecutará el ciclo de instrucción de la instrucción posterior a **call sumMtoN**

  b. Actualiza el stack pointer: **ESP ← ESP + 4**

- Es necesario que en el epílogo de la subrutina, antes de la ejecución de RET el stack pointer apunte a la dirección de la pila donde está almacenada la dirección de retorno.

11.10. Estado de la pila

11.10.1. Previo al salto de la llamada a la subrutina

- Estado de la pila ejecutando la *rutina main* justo antes de ejecutar la instrucción `call sumMtoN`:

  - El frame activo de la pila es el correspodiente a main.
  - Los últimos datos apilados en el *frame main* son los argumentos de *sumMtoN*

```
        push N
        push M
        call sumMtoN
```

- Contenido de los registros RIP,EBP,ESP:

  - RIP:
  - EBP:
  - ESP:

11.10.2. Posterior al salto de la llamada a la subrutina

- Estado de la pila ejecutando la *subrutina sumMtoN* justo después de ejecutar el salto `call sumMtoN`:

  - El frame activo de la pila es el correspodiente a main.
  - El último dato apilado en el frame main es la *dirección de retorno* a main desde sumMtoN

- Contenido de los registros RIP,EBP,ESP:

  - RIP:
  - EBP:
  - ESP:

11.10.3. Creación del nuevo frame *sumMtoN*

- Estado de la pila después de ejecutar:

```
sumMtoN:
        push %ebp
        mov  %esp,%ebp
```

- Contenido de los registros RIP,EBP,ESP:
  - RIP:
  - EBP:
  - ESP:

11.10.4. Previo al salto de retorno

- Estado de la pila ejecutando la *subrutina sumMtoN* justo antes de ejecutar la instrucción `ret`:
  - El frame activo de la pila es el correspodiente a sumMtoN.
  - El puntero del top *ESP* del frame sumMtoN apunta a la dirección de pila que contiene la *dirección de retorno*
- Contenido de los registros RIP,EBP,ESP:
  - RIP:
  - EBP:
  - ESP:

11.10.5. Posterior al salto de retorno

- Estado de la pila ejecutando la *subrutina sumMtoN* justo después de ejecutar la instrucción `ret`:
  - La ejecución de `ret` ha realizado las siguientes operaciones:
    - `pop %irp`
  - El frame activo de la pila es el correspodiente a main.
- Contenido de los registros RIP,EBP,ESP:
  - RIP:
  - EBP:
  - ESP:

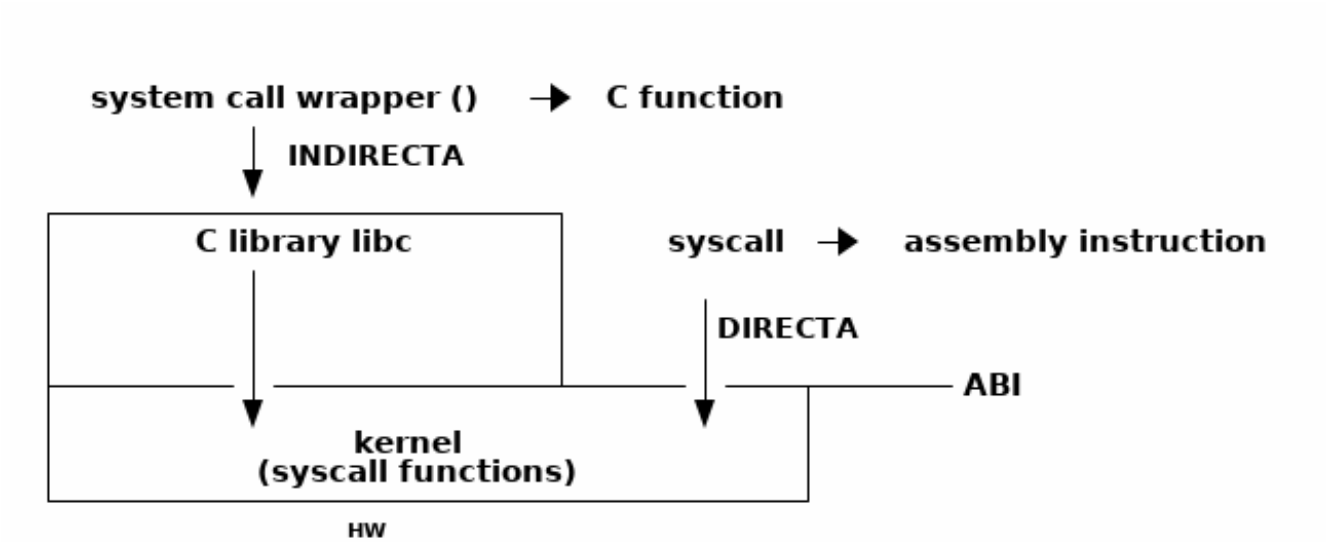
12. Llamadas al Sistema Operativo

12.1. Introducción

- Se conoce con el nombre de *llamadas al sistema* a las Llamadas que realizar el programa de usuario a subrutinas del Kernel del Sistema Operativo.
- Para realizar funciones privilegiadas del sistema operativo como el acceso a los dispositivos i/o de la computadora es necesario que los programas de usuario llamen al kernel para que sea éste quien realice la operación de una manera segura y eficaz. De esta forma se evita que el programador de aplicaciones acceda al hardware y al mismo tiempo se facilita la programación.
- Ejemplos de llamadas

- **exit** : el kernel suspende la ejecución del programa eliminando el proceso
- **read** : el kernel lee los datos de un fichero accediendo al disco duro
- **write**: el kernel escribe en un fichero
- **open** : el kernel abre un fichero
- **close**: el kernel cierra el proceso
- más ejemplos de llamada en el listado `man 2 syscalls`

- La llamada a los servicios del kernel denominados *syscalls* se puede realizar de dos formas: **directa** o **indirecta**
  - Directa: desde ASM mediante la instrucción `syscall`
  - Indirecta: desde C o ASM mediante funciones de la librería `libc`: wrappers de las llamadas directas
- API/ABI



- Ejemplo arquitectura i386
  - Los 6 primeros argumentos se pasan a través de los registros: [EBX-ECX-EDX-ESI-EDI-EBP] previamente a la instrucción de la llamada `int 0x80`

```
* printf() -> write(int fd, const void *buf, size_t count) -> [EBX-ECX-EDX-ESI-EDI-EBP, ↩
    int 0x80] -> kernel syscall write
* API      ->        wrapper function                    ->                ABI  ↩
                    ->  kernel syscall
```

- Ejemplo arquitectura amd64
  - Los 6 primeros argumentos se pasan a través de los registros: [RAX-RDI-RSI-RDX-R10-R8-R9] previamente a la instrucción de la llamada `syscall`

```
* printf() -> write(int fd, const void *buf, size_t count) -> [RAX-RDI-RSI-RDX-R10-R8-R9 ↩
    ,syscall] -> kernel syscall write
* API      ->        wrapper function                    ->                ABI  ↩
                    ->  kernel syscall
```

12.2. Ejemplos

- Ver en el Apéndice

13. Bibliografias

- Listado Programación Ensamblador.
  - Apuntes completos [?] de programación en lenguaje AT&T con diversidad de aspectos.