

Estructura de Computadores (240306)

Cándido Aramburu

2022-09-05

Table of Contents

I Arquitectura del Repertorio de Instrucciones (ISA): computadora von Neumann, datos, instrucciones, programación	1
1. Introducción a la Estructura de los Computadores	2
1.1. Introducción	2
1.2. Arquitectura de una máquina	2
1.3. Arquitectura desde la perspectiva HW	3
1.4. Lenguajes de Programación y Lenguaje Máquina	7
1.5. Interface Software/Hardware	9
1.6. Temario	10
1.6.1. Bibliografía Basica	11
1.6.2. Bibliografía Complementaria	13
1.7. Profesorado	13
1.8. Grado Informática	14
1.9. Calendarios	14
1.10. Ejercicios mediante resolución de problemas	16
1.11. Prácticas	17
1.11.1. Memorias	17
1.12. Recursos Informáticos	17
1.12.1. UPNA	17
1.12.2. Estaciones de Trabajo: 32 y 64 bits	18
1.12.3. Registrarse	19
1.13. Grupos de Prácticas	19
1.14. Metodología	20
1.14.1. Distribución de créditos	21
1.14.2. Distribución de créditos de las Prácticas	21
1.15. Evaluación	21
1.16. Exámenes	22
2. Arquitectura Von Neumann	23
2.1. Arquitectura Von Neumann	23
2.1.1. Temario	23
2.1.2. Contexto Histórico	23
2.2. Institute Advanced Machine (IAS) : Arquitectura	25
2.2.1. Referencia	25
2.2.2. Ejemplo del Programa sum1toN	25
2.3. Estructura de la computadora IAS	26
2.3.1. Módulos	26
2.3.2. Unidad Central de Proceso (CPU)	28
2.3.3. Memorias	29

2.3.4. Bus	31
2.3.5. Input Output (I/O)	32
2.3.6. Animación del Ciclo de Instrucción	32
2.4. ISA: Arquitectura del Repertorio de Instrucciones de la máquina IAS	33
2.4.1. Formato de los datos e Instrucciones de la Computadora IAS	33
2.4.2. Repertorio ISA	34
2.4.3. Interfaz ISA	37
2.5. Programación en el Lenguaje Ensamblador IAS	38
2.5.1. Estrategia del Desarrollo de un Programa en Lenguaje Ensamblador	38
2.5.2. Ejemplo 1: sum1toN.ias	39
2.5.3. Ejemplos de Programas en Lenguaje IASSim	44
2.6. Operación de la Máquina IAS: Ruta de Datos	44
2.7. Conclusiones	45
3. Representación de los Datos	46
3.1. Temario	46
3.2. Objetivo	46
3.3. Datos e Instrucciones: Codificación Binaria	46
3.4. Bit, Byte, Palabra	46
3.5. Números Enteros	47
3.5.1. Base Decimal	47
3.5.2. Base Binaria	47
3.5.3. Conversión Decimal-Binaria	48
3.5.4. Base Octal	48
3.5.5. Calculadora	49
3.5.6. Python	49
3.5.7. Enteros con Signo	49
3.6. Números Reales	52
3.6.1. Coma Fija	52
3.6.2. Coma Flotante	53
3.7. Character Type	56
3.7.1. ASCII	56
3.7.2. Python	58
3.7.3. Unicode UTF-8	58
3.8. ISO-8859-1	61
3.8.1. Programación en C	61
3.8.2. Otros	61
4. Operaciones Aritmeticas y Logicas	62
4.1. Temario	62
4.2. Objetivo	62
4.3. Introduccion	62
4.4. Aritmetica Binaria	62

4.4.1. Suma en módulo 2 (binaria) en binario puro (Nº NATURALES)	62
4.4.2. Resta en módulo 2 (binaria) en binario puro	64
4.4.3. Suma/Resta en módulo 2 (binaria) en complemento a 2	64
4.4.4. Suma en Módulo 16 (Hexadecimal)	66
4.4.5. Resta en Módulo 16 (Hexadecimal)	67
4.4.6. Tipos de variables en C	68
4.5. Operaciones Logicas	68
4.5.1. Operadores BITWISE	68
4.6. Multiplicación	69
4.7. Programación	69
4.7.1. funciones matemáticas	69
4.7.2. Aplicación	70
4.8. Hardware	70
4.8.1. Circuitos Digitales	70
4.8.2. Unidad Aritmetico Lógica (ALU)	70
4.8.3. Registro de flags EFLAG	71
4.8.4. Float Point Unit-FPU	73
5. Representación de las Instrucciones	74
5.1. Temario	74
5.1.1. Bibliografía	74
5.2. Objetivos	74
5.2.1. Requisitos	74
5.3. Introducción	74
5.3.1. Lenguaje máquina y el formato binario	74
5.3.2. El lenguaje máquina y el lenguaje ensamblador	75
5.3.3. Lenguajes de Alto Nivel	75
5.4. Arquitectura	76
5.4.1. Contexto	76
5.4.2. Instruction Set Architecture (ISA)	77
5.5. Procesadores Intel con arquitectura x86	77
5.5.1. Nomenclatura	77
5.6. Estructura de la Computadora	79
5.6.1. Arquitectura	79
5.6.2. CPU	79
5.6.3. Memoria	81
5.7. Elementos de una Instrucción Máquina	83
5.7.1. Direcciones implícitas	84
5.7.2. Tipos de Arquitecturas de Operando: Ejemplos	84
5.8. Representación de las instrucciones en lenguaje ensamblador (ASM) para computadoras en general	86
5.8.1. Lenguaje Máquina Binario	86

5.9. Operandos: Modos de Direccionamiento	86
5.9.1. Localización	86
5.9.2. Direcciones referenciadas durante el ciclo de instrucción	87
5.9.3. Direccionamiento para un lenguaje general	87
5.10. Operaciones	89
5.10.1. Códigos de Operación	89
5.10.2. Tipos de Operaciones	89
6. Programación en Lenguaje Ensamblador (x86): Construcciones básicas de los lenguajes de alto nivel.	91
6.1. Temario	91
6.2. Introducción	91
6.2.1. Objetivos	91
6.2.2. Requisitos	91
6.3. Estructura de la Computadora	91
6.3.1. CPU	91
6.3.2. Memoria	92
6.3.3. Memoria Principal	92
6.3.4. Registros internos a la CPU	92
6.4. Lenguaje Intel versus Lenguaje AT&T	95
6.4.1. Lenguajes ensamblador de la arquitectura i386/amd64	95
6.4.2. Sintaxis de las instrucciones en el lenguaje INTEL	95
6.4.3. Traductores del proceso de ensamblaje	97
6.4.4. Código Máquina	98
6.4.5. Assembler "as"	99
6.5. Representación de los datos en lenguaje ensamblador (ASM) para la arquitectura i386/amd64	100
6.5.1. Tipos de Datos	100
6.5.2. Tamaño del operando x86	101
6.5.3. Alineamiento de Bytes: Big-LittleEndian	102
6.6. Operandos: Modos de Direccionamiento	104
6.6.1. Localización	104
6.6.2. Modos de Direccionamiento	104
6.7. Repertorio de Instrucciones: Operaciones	106
6.7.1. Manuales de referencia	106
6.7.2. TRANSFERENCIA	106
6.7.3. ARITMÉTICOS	108
6.7.4. LÓGICOS	109
6.7.5. MISCELÁNEOS	109
6.7.6. SALTOS (generales)	110
6.7.7. SALTOS Sin Signo (Cardinal) SALTOS Con Signo (Integer)	110
6.7.8. FLAGS (ODITSZAPC)	111

6.7.9. Sufijos	111
6.7.10. Códigos de Operación	111
6.8. Mnemónicos Básicos (Explicados)	112
6.8.1. Operaciones aritméticas	112
6.8.2. Procesamiento Condicional	113
6.8.3. Saltos	114
6.8.4. Desplazamiento y rotación	114
6.8.5. Cambiar el Endianess	114
6.9. Formato de Instrucción: ISA Intel x86-64	115
6.10. Subrutinas	115
6.10.1. Introducción	115
6.10.2. Lenguaje C: Sentencia Función	115
6.10.3. Anidamiento de Funciones	117
6.10.4. Pila/Frame	117
6.10.5. Definición de la subrutina	117
6.10.6. Registros a Preservar	118
6.10.7. Argumentos de la subrutina	119
6.10.8. Llamada a la subrutina	119
6.10.9. Retorno de la subrutina	120
6.10.10. Estado de la pila	120
6.11. Llamadas al Sistema Operativo	121
6.11.1. Introducción	122
6.11.2. Ejemplos	123
6.12. Bibliografias	123
VIII Apéndices	124
7. Programas ensamblador IASSim	125
7.1. Ejemplo 1: sum1toN.ias	125
7.1.1. Lenguaje ensamblador iassim	125
7.1.2. Ejemplo 2: Producto/Cociente	127
7.1.3. Ejemplo 3: Vectores	130
8. Simulador IASSim	135
8.1. Máquina Virtual Java JVM	135
8.2. Simulador IAS	135
8.3. Simulación/Depuración	136
9. Lenguajes Ensamblador	138
9.1. Intel x86 / AMD 64	138
9.1.1. Hola Mundo	138
9.1.2. Programación ensamblador	139
9.1.3. Números Reales	139
9.1.4. Discusión por qué ASM AT&T	140
9.1.5. Miscellaneous	140

9.2. Motorola 68000	141
9.2.1. Hola Mundo	141
9.2.2. ISA	141
9.3. MIPS	143
9.3.1. ISA	143
9.4. ARM	144
9.4.1. Hola Mundo	144
9.4.2. ISA	144
10. Lenguajes de programación para sum1toN	145
10.1. Otros Lenguajes para sum1toN	145
11. RTL Register Transfer Language	151
11.1. Lenguaje RTL	151
11.1.1. Introducción	151
11.1.2. Registros	151
11.1.3. Símbolos	153
11.1.4. Sentencias RTL	154
11.1.5. Ejemplos RTL con expresiones aritmético-lógicas	155
12. Formato de Instrucción: ISA Intel x86-64	156
12.1. Formato de Instrucción: ISA Intel x86-64	156
12.1.1. Ejemplo subq \$16,%rsp	156
12.1.2. Otros x86-32	157
13. FPU x87	158
13.1. FPU x87	158
13.1.1. Resumen	158
13.1.2. Refs	159
14. Pila	160
14.1. Concepto	160
14.2. Anchura	160
14.3. Frame: frame pointer y stack pointer	161
14.4. Instrucciones Ensamblador Push-Pop	162
14.4.1. Anidamiento de llamadas	163
15. Llamadas al Sistema Operativo	164
15.1. Introducción	164
15.2. Manuales de las llamadas	165
15.3. Llamada INDIRECTA	165
15.4. LLamada DIRECTA	165
15.4.1. Argumentos de la llamada directa	166
15.4.2. Códigos de la llamada directa	166
15.5. Ejemplos: lenguaje C	166
15.6. Ejemplos: ASM INDIRECTO	167
15.7. Ejemplos: ASM DIRECTO	167

15.8. Línea de Comandos	168
15.8.1. Procedimiento	168
15.8.2. Stack Initialization	169
15.8.3. Rutina principal con Retorno	170
15.8.4. Ejercicios: suma_linea_com.s ,maximum_linea_com.s	171
16. Lenguaje de Programación C	173
16.1. Introducción	173
16.2. Casting	173
16.2.1. Concepto	173
16.2.2. Ejemplo	173
16.3. Puntero	173
16.3.1. Referencias	173
16.3.2. Introducción	173
16.3.3. Concepto	174
16.3.4. Módulo Ilustrativo	178
16.3.5. Declaración	179
16.3.6. Operador Dirección	180
16.3.7. Operador Indirección o Dereferencia	180
16.3.8. Ejemplo	180
16.3.9. Aplicaciones de los punteros	181
16.3.10. String Literal	182
16.3.11. Puntero a Puntero	182
16.3.12. String Variable	183
16.3.13. Funciones	183
17. Apéndice Prácticas	184
17.1. Prácticas	184
17.1.1. Documentación: guiones, bibliografía, apuntes	184
17.1.2. Plataforma de Desarrollo	184
17.1.3. Documento Memoria: Contenido y Formato	186
17.1.4. Evaluación	187
17.1.5. Programación	187
17.1.6. Compilación	188
17.1.7. Errores Comunes	191
18. Arquitectura amd64	192
18.1. Módulo fuente: sum1toN.s	192
19. Exámenes de Cursos Anteriores	193
19.1. Año 2018	193
19.1.1. Noviembre	193
19.2. Año 2017	195
20. Miaulario: Videoconferencia	205
20.1. Introducción	205

20.2. Instalación de Zoom	205
20.3. Guía de usuario Zoom	205
20.3.1. Configuración	205
20.4. Sesión de videoconferencia	205
IX Bibliografía	206
X Glosario	208
XI Colofón	209

I Arquitectura del Repertorio de Instrucciones (ISA): computadora von Neumann, datos, instrucciones, programación.

Chapter 1. Introducción a la Estructura de los Computadores

1.1. Introducción

- El objetivo de la asignatura Estructura de Computadores (240306) del Grado de Ingeniería Informática [<http://www.unavarra.es/ets-industrialesytelecos/estudios/grado/grado-en-ingeneria-informatica/presentacion?submenu=yes>] de la Universidad Pública de Navarra es ser un curso introductorio universitario a la arquitectura de los computadores, estudiando sus componentes básicos (procesador, memoria y módulo de entrada/salida) así como la programación de bajo nivel en lenguaje ensamblador x86 mediante la utilización de herramientas de desarrollo software como el compilador, depurador, etc.

1.2. Arquitectura de una máquina

- Arquitectura: Organización, Estructura: Qué, Cómo, Implementación (tecnología)

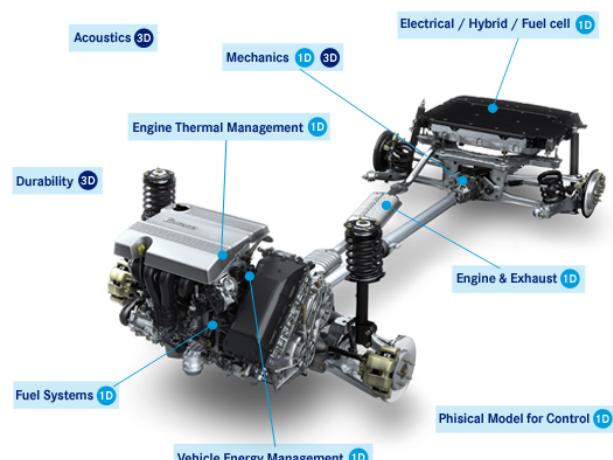


Figure 1. Estructura del Automóvil

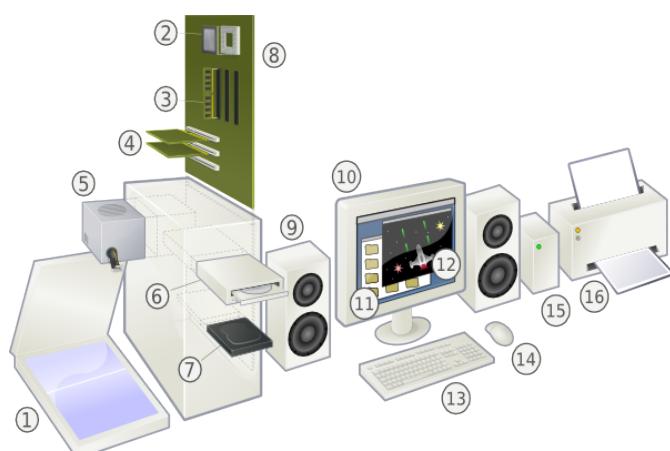


Figure 2. Personal Computer

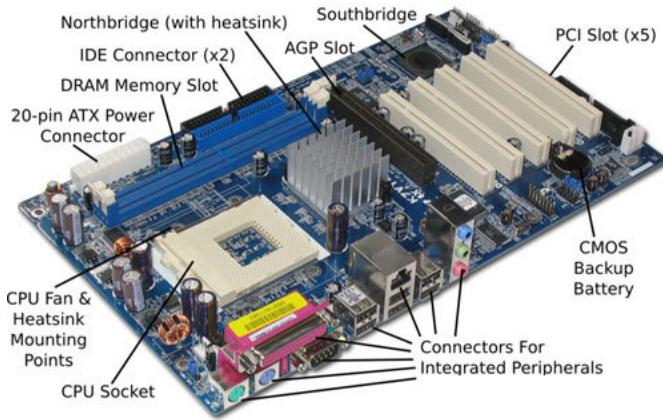


Figure 3. Placa Base

1.3. Arquitectura desde la perspectiva HW

- 4 Módulos Básicos: CPU-MEMORIA-CONTROLADORES Entrada/Salida [Periféricos]-BUSES



Figure 4. CPU Intel Core i7 4^a Generación

Note, as well as the different number of pins, the different spacing of the slots in the connector-edge

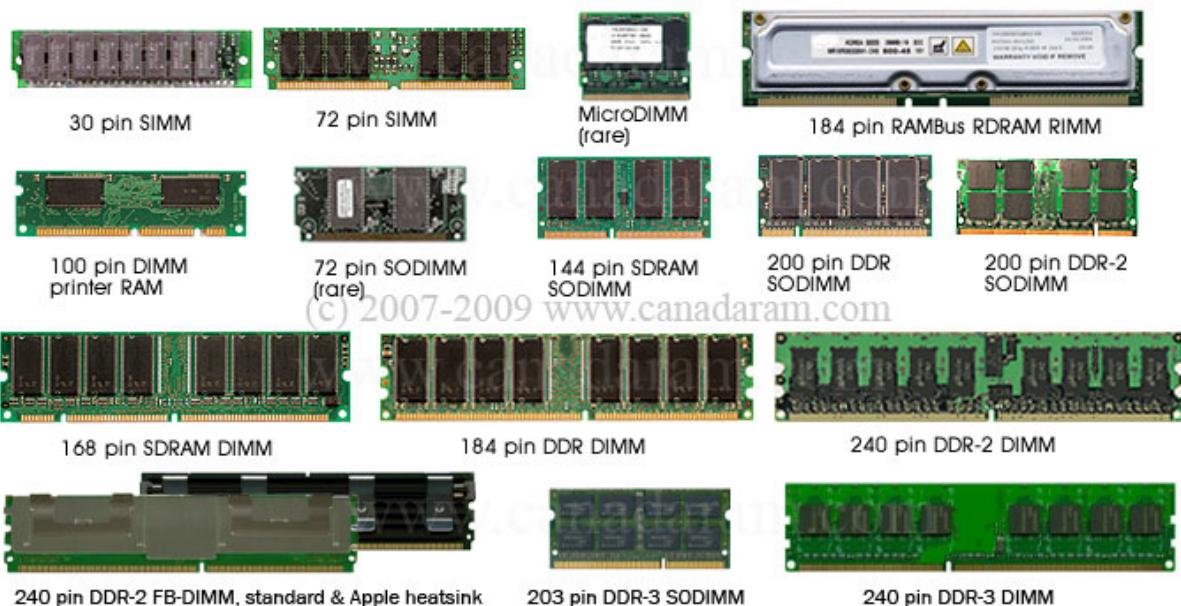


Figure 5. Memoria DRAM



Figure 6. Periféricos: Memoria de Semiconductor Solid State Drive (SSD)



Figure 7. Periféricos: Disco Duro Seagate



Figure 8. Periféricos: Memoria M2M

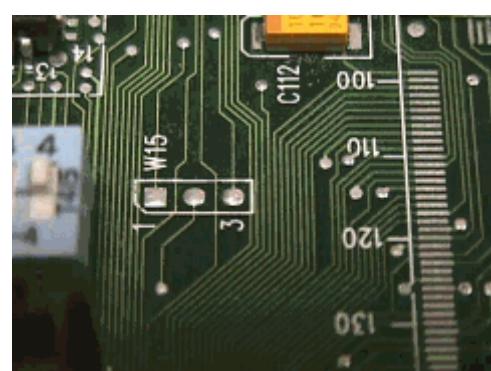


Figure 9. Bus de la Placa Base

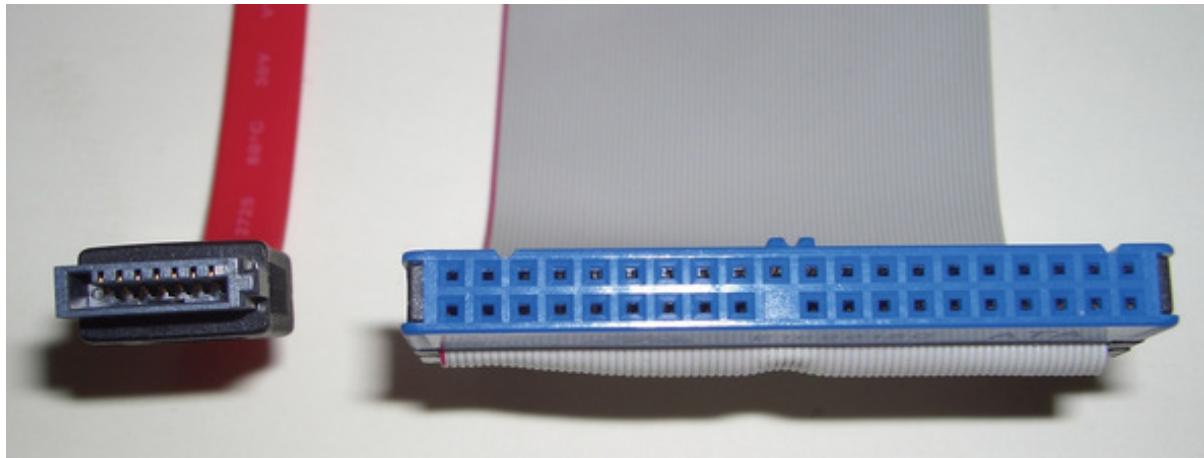


Figure 10. Bus Cableado

1.4. Lenguajes de Programación y Lenguaje Máquina

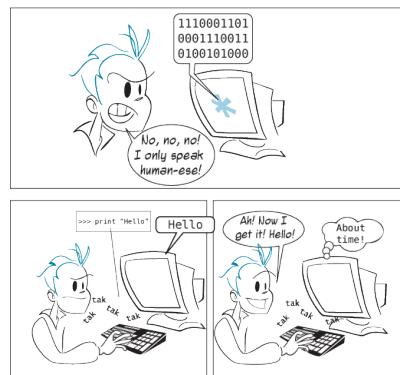


Figure 11. Lenguaje Máquina: Binario

- Lenguaje Pascal

```
program Hello_world;
begin
  writeln('hello world')
end.
```

- L. Máquina-Binario Intel x86

```
# [2] begin
0000000004001a0 <PASCALMAIN>:
4001a0: 01010101
4001a1: 01001000 10001001 11100101
4001a4: 01001000 10000011 11101100 00010000
4001a8: 01001000 10001001 01011101 11111000
4001ac: 11101000 10110111 00111110 00000001 00000000
# [3] writeln('hello world')
4001b1: 11101000 11001010 10010011 00000001 00000000
4001b6: 01001000 10001001 10100011
4001b9: 01001000 10001001 11011110
4001bc: 01001000 10111010 11000000 11110110 01100001 00000000 00000000
```

```

4001c3: 00000000 00000000 00000000
4001c6: 10111111 00000000 00000000 00000000 00000000
4001cb: 11101000 01111000 10010110 00000001 00000000
4001d0: 11101000 00010011 00111101 00000001 00000000
4001d5: 01001000 10001001 11011111
4001d8: 11101000 01110011 10010101 00000001 00000000
4001dd: 11101000 00000110 00111101 00000001 00000000
4001e2: 11101000 10011001 01000010 00000001 00000000
4001e7: 01001000 10001011 01011101 11111000
4001eb: 10101001
4001ec: 10010011

```

- Lenguaje Máquina (Código Hexadecimal) vs Lenguaje Ensamblador ASM x86

```

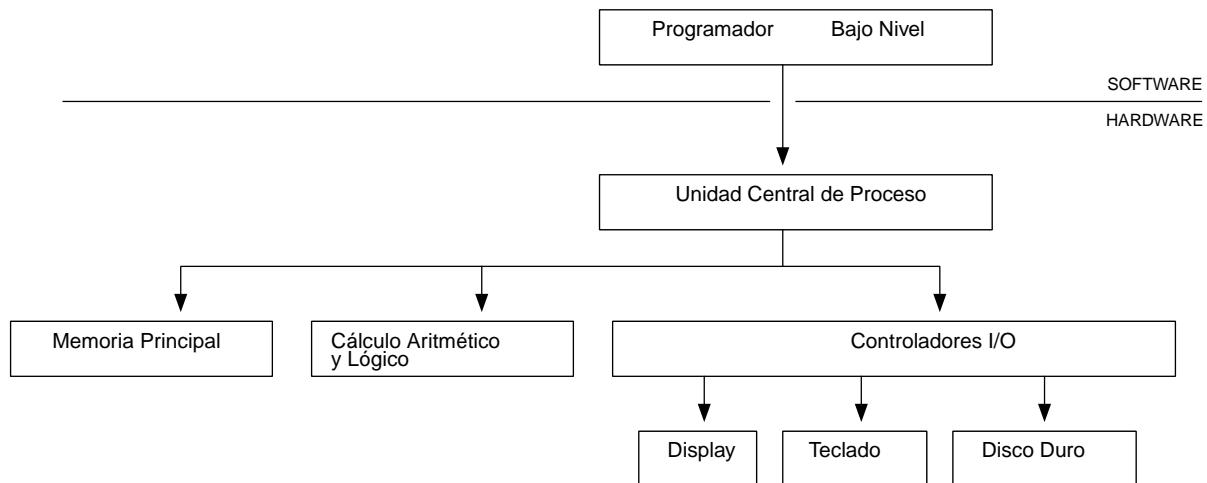
# [2] begin
0000000004001a0 <PASCALMAIN>:
4001a0: 55                      push   %rbp
4001a1: 48 89 e5                mov    %rsp,%rbp
4001a4: 48 83 ec 10             sub    $0x10,%rsp
4001a8: 48 89 5d f8             mov    %rbx,-0x8(%rbp)
4001ac: e8 b7 3e 01 00          callq  414068 <FPC_INITIALIZEUNITS>
# [3] writeln('hello world')
4001b1: e8 ca 93 01 00          callq  419580 <fpc_get_output>
4001b6: 48 89 c3                mov    %rax,%rbx
4001b9: 48 89 de                mov    %rbx,%rsi
4001bc: 48 ba c0 f6 61 00 00    movabs $0x61f6c0,%rdx
4001c3: 00 00 00
4001c6: bf 00 00 00 00          mov    $0x0,%edi
4001cb: e8 78 96 01 00          callq  419848 <FPC_WRITE_TEXT_SHORTSTR>
4001d0: e8 13 3d 01 00          callq  413ee8 <FPC_IOCHECK>
4001d5: 48 89 df                mov    %rbx,%rdi
4001d8: e8 73 95 01 00          callq  419750 <fpc writeln_end>
4001dd: e8 06 3d 01 00          callq  413ee8 <FPC_IOCHECK>
4001e2: e8 99 42 01 00          callq  414480 <FPC_DO_EXIT>
4001e7: 48 8b 5d f8             mov    -0x8(%rbp),%rbx
4001eb: c9                      leaveq
4001ec: c3                      retq

```

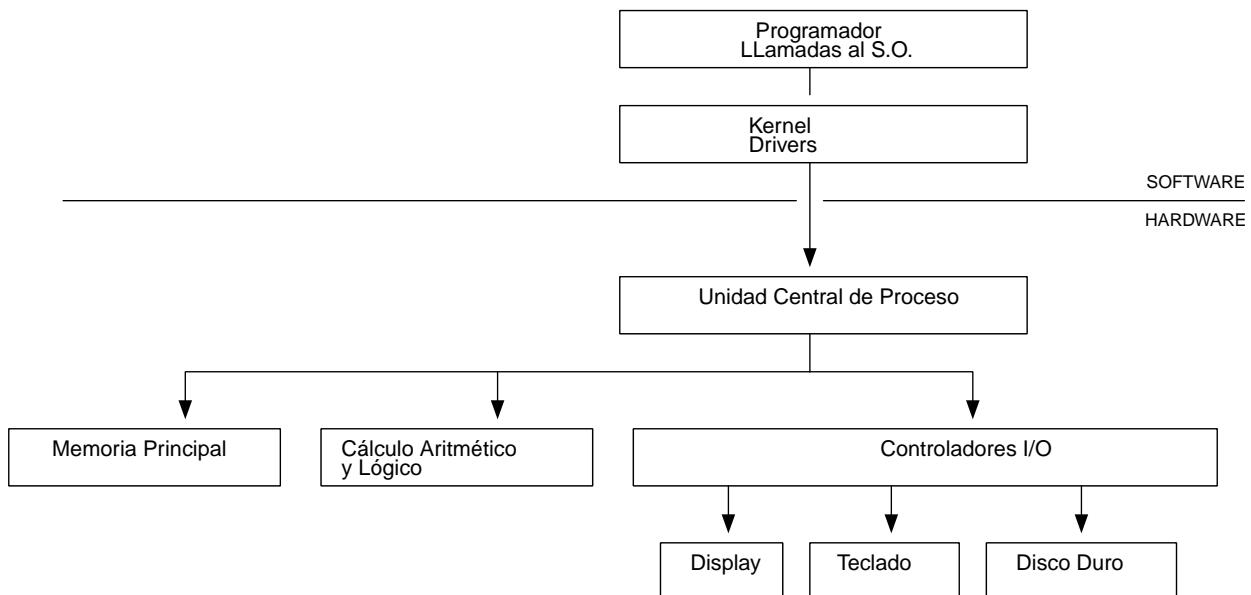
- Lenguaje de programación humano (texto) vs Lenguaje de programación máquina (binario)
- Traducir → Compilar
- Interpretar
- Lenguajes de programación: javascript, ruby, python, java, bash, C, assembly
 - Niveles de abstracción: humano/máquina

1.5. Interface Software/Hardware

- Interacción directa del Programador con el hardware Máquina → Programación en un lenguaje de bajo nivel (ensamblador, C)



- Programación de bajo nivel
 - Módulo fuente: Lenguaje C ó Lenguaje ensamblador
 - Módulo objeto: Lenguaje máquina → Compilación, Ensamblaje y Enlazado del Módulo Fuente.
 - Sistemas Bare Metal: No hay Sistema Operativo. Programamos directamente sobre el Hardware.
- Interacción indirecta del programador con el hardware de la máquina. El programador interactúa con el Sistema Operativo (S.O.)
 - Interacción del Kernel (núcleo, pej linux) del Sistema Operativo (S.O.) con la máquina (pej intel x86)



- Sistemas con Sistema Operativo (pej GNU/linux): A la hora de programar recurrimos a librerías que acceden al hardware a través del sistema operativo. Pej la función printf(), de la librería libc, al compilarse se traduce en la función write() del sistema operativo la cual llama al driver (en el sistema operativo) de la tarjeta gráfica de la pantalla. El Sistema operativo consigue "abstraer" el HW físico de la máquina y facilita enormemente la programación al no tener que conocer el funcionamiento físico de la computadora.

1.6. Temario

- [Web Estructura de Ordenadores \[http://www.unavarra.es/ficha-asignaturaDOA?languageId=100000&codPlan=240&codAsig=240306\]](http://www.unavarra.es/ficha-asignaturaDOA?languageId=100000&codPlan=240&codAsig=240306)

Temario

- 1 - Introducción
- 2 - Arquitectura de Von Neumann
 - 2.1 CPU
 - 2.2 Memoria
 - 2.3 Entrada / Salida
- 3 - Representación de datos
 - 3.1 Bit, Byte y Palabra
 - 3.2 Caracteres, enteros y reales
- 4 - Aritmética y lógica
 - 4.1 Operaciones aritméticas y lógicas sobre enteros en binario
 - 4.2 Redondeo y propagación de error en números reales
- 5 - Representación de instrucciones
 - 5.1 Lenguaje máquina, lenguaje ensamblador y lenguajes de alto nivel
 - 5.2 Formato de instrucción
 - 5.3 Tipos de instrucción y modos de direccionamiento
- 6 - Programación en lenguaje ensamblador de construcciones básicas de los lenguajes de alto nivel

- 6.1 Sentencias de asignación
- 6.2 Sentencias condicionales
- 6.3 Bucles
- 6.4 Llamadas y retorno de función o subrutina
- 7 - Arquitectura y organización de la CPU
 - 7.1 Conjunto de instrucciones
 - 7.2 Arquitecturas CISC, RISC y VLIW
 - 7.3 Fases de ejecución de una instrucción
 - 7.4 Camino de datos
- 8 - Sistema de entrada / salida
 - 8.1 Sincronización por encuesta
 - 8.2 Sincronización por interrupción
 - 8.3 Vector de interrupciones
 - 8.4 Acceso directo a memoria DMA
 - 8.5 Programación en lenguaje ensamblador de rutinas de entrada/salida
- 9 - Organización de la memoria
 - 9.1 Jerarquía de memoria
 - 9.2 Latencia y ancho de banda
 - 9.3 Memoria cache
 - 9.4 Memoria virtual

1.6.1. Bibliografía Basica

- Teoría
 - [William Stalling \[http://williamstallings.com/ComputerOrganization/\]](http://williamstallings.com/ComputerOrganization/)



Figure 12. William Stalling Book

Organización y arquitectura de computadores .William Stallings

Computer Organization and Architecture: Designing for Performance.
William Stallings
9^a Ed Upper Saddle River (NJ) : Prentice Hall, [2013]
ISBN 0-273-76919-7 . 2012

- COA 7^a [<http://williamstallings.com/COA/COA7e.html>]
- 11^o Ed 2019 [<https://www.pearson.com/us/higher-education/program/Stallings-Pearson-e-Text-for-Computer-Organization-and-Architecture-Access-Code-Card-11th-Edition/PGM2043621.html>]
- Prácticas:

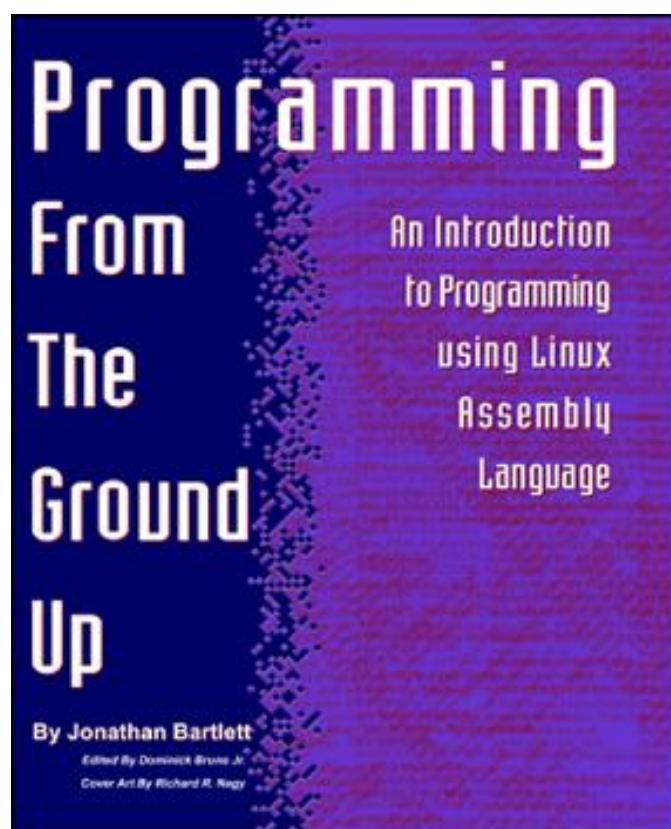


Figure 13. Programming Assembly

Programming from the Ground Up
by Jonathan Bartlett Edited by Dominick Bruno, Jr.
Copyright © 2003 by Jonathan Bartlett
ISBN 0-9752838-4-7
Published by Bartlett Publishing in Broken Arrow, Oklahoma

- Programming from the Ground Up by Jonathan Bartlett. Programación en Lenguaje Ensamblador AT&T para la arquitectura x86.
 - PGU book online [<http://programminggroundup.blogspot.com.es/2007/01/programming-from-ground-up.html>]

- [PGU book home](http://savannah.nongnu.org/projects/pgubook/) [<http://savannah.nongnu.org/projects/pgubook/>]
- [pdf](http://download.savannah.gnu.org/releases/pgubook/) [<http://download.savannah.gnu.org/releases/pgubook/>]
- **El ensamblador... pero si es muy fácil** [<https://upcommons.upc.edu/handle/2117/115067>]: IA-32 (i386) (sintaxis AT&T)
 - Manuales GNU:
- Documentación de los Manuales: La mayoría de las aplicaciones y herramientas software de la fundación GNU y Linux disponen de Manuales bien on-line o bien localmente en la propia máquina. Manual del compilador gcc: `$man gcc`



Es una habilidad a adquirir por el profesional informático el acceder y utilizar dichos manuales así como disponer de hojas de referencia de acceso rápido durante las sesiones de trabajo.

1.6.2. Bibliografía Complementaria

- Ir al capítulo de la [bibliografía](#).

1.7. Profesorado

- Cándido Aramburu Mayoz.
 - Doctor Ingeniero Telecomunicación. Profesor Titular de Universidad.
 - Edificio los Tejos, planta 2^a. Despacho 2028.
 - Tutorías semestre Otoño : Viernes de 8:00 a 14:00
 - correo electrónico interno: a través del servidor Miaulario.
 - [PDI Profesorado](http://www.unavarra.es/ficha-docente?rangoLetras=a-z&uid=364) [<http://www.unavarra.es/ficha-docente?rangoLetras=a-z&uid=364>]
- Andrés Garde Gurpegui
 - Técnico Superior de la Dirección General de Informática y Telecomunicaciones del Gobierno de Navarra.
 - Profesor Asociado (Prácticas de Laboratorio)
 - Edificio de Los Tejos, planta 2^a, Sala de Asociados del departamento de INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE COMUNICACIÓN.
 - andres.garde@unavarra.es
 - [PDI Profesorado](http://www.unavarra.es/pdi?uid=6578) [<http://www.unavarra.es/pdi?uid=6578>]
- Carlos Juan de Dios Ursua
 - Ingeniería Eléctrica y Electrónica
 - Master en Robótica y Automatización
 - Profesor Asociado (Teoría Estructura Computadores)
 - Edificio de Los Tejos, planta 2^a, Sala de Asociados del departamento de INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE COMUNICACIÓN.

- carlos.juandedios@unavarra.es

1.8. Grado Informática

- **Grado Informática** [https://www.unavarra.es/sites/grados/informatica-y-telecomunicacion/ingenieria-informatica/asignaturas-y-profesorado.html]
 - 200 - Escuela Técnica Superior de Ingeniería Industrial, Informática y de Telecomunicación
 - 240 - Graduado o Graduada en Ingeniería Informática por la Universidad Pública de Navarra

1.9. Calendarios

- **Calendario administrativo** [https://www.unavarra.es/sites/estudios/calendarios.html]
 - Teoría: Grupo1 GG Jueves 15:00 A318 ; Grupo2 GG Viernes 17:00 A207 ; Grupo91 GG Lunes 19:00 A135
- Prácticas
 - **Calendario Aulas Informática 1º Cuatrimestre** [https://www2.unavarra.es/gesadj/servicioInformatico/usuario/aulas/Horario1C.htm]
 - **Calendario Aulas Informática 2º Cuatrimestre** [https://www2.unavarra.es/gesadj/servicioInformatico/usuario/aulas/Horario2C.htm]
- Fiestas

12 de octubre miércoles (Fiesta Nacional de España).
 01 de noviembre martes (Festividad de Todos los Santos).
 03 de diciembre martes (San Francisco Javier, Día de Navarra).
 06 de diciembre jueves (Día de la Constitución).

- **Horarios y Aulas** [https://www.unavarra.es/sites/grados/informatica-y-telecomunicacion/ingenieria-informatica/horarios-y-aulas.html#cCentralUPNA]

- Teoría

	Lunes	Martes	Miércoles	Jueves	Viernes
15:00	T91-A135		PG1G2-A303	T1-A125	
17:00	T91-A327		PG2-A306	T2-A234	
19:00	PG1-A336				

- Prácticas

A336 PG1 17:00	A303 PG1G2 15:00
P327 P91 19:00	A306 PG2 17:00
3/10	5/10
17/10	19/10

24/10	
	2/11
7/11	16/11
21/11	23/11
28/11	30/11

- * P91: 3/10, 17/10, 24/10, 7/11, 21/11, 28/11
- * PG1: 3/10, 17/10, 24/10, 7/11, 21/11, 28/11
- * PG2: 5/10, 19/10, 2/11, 16/11, 23/11, 30/11
- * PG1G2: 5/10, 19/10, 2/11, 16/11, 23/11, 30/11

- * T1: Aula A125
- * T2: Aula A234
- * T91: Aula A135
- * PG1G2: Miércoles a las 15:00 en E-ISM/A303 (40 puestos) -> Prácticas de los Grupos de teoría G1 y G2
- * PG2: Miércoles a las 17:00 en A306 (36 puestos) -> Prácticas del Grupo de teoría G2
- * PG1: Lunes a las 19:00 en A336 (36 puestos) -> Prácticas del Grupo de teoría G1
- * P91: Lunes a las 17:00 en A327 (20 puestos) -> Prácticas del Grupo de teoría Euskera
- * Prof Andrés Garde: 3 Grupos de castellano P1,P2,P3
- * Prof Carlos Juan de Dios: Grupo de castellanos T2
- * Prof Cándido Aramburu: Grupo de euskera T91-P91 y grupo de castellano T1
- * Aula E-ISM: 34 puestos

	A	B	C	D	E	F
1	2022 / 2023 SEPTIEMBRE	LUNES	MARTES	MIÉRCOLES	JUEVES	VIERNES
2		29	30	31	1	2
3		5	6	7	Sesión 1 (T1, T2*) G1 (A125) 15:00 G2 (A234) 17:00	8 9
4		12	13	14	Sesión 2 (T2) 15	16
5		19	20	19	Sesión 3 (T3*) G1 (?) 15:00 G2 (?) 17:00	Sesión 4 (T3/T4*) 22 23
6		26	27	26	Sesión 5 (T5) G1 (?) 15:00 G2 (?) 17:00	Sesión 6 (T5/T6*) 29 30
7		Práctica 1 PG1 (A336) 19:00 3	4	Práctica 1 PG1G2 (A303) 15:00 PG2 (A306) 17:00 5	Sesión 7 (T6) 6	7
8		10	11	12	Sesión 8 (T6) 13	Examen parcial 1 (T1-T5) + Programación papel 14
9		Práctica 2 17	18	Práctica 2 19	Sesión 9 (T6) 20	21
10		Práctica 3 24	25	26	Sesión 10 27	Examen de prácticas 1 (P1-P2) 28
11		31	1	Práctica 3 2	Sesión 11 3	4
12		Práctica 4 7	8	9	Sesión 12 10	11
13		14	15	Práctica 4 16	Sesión 13 17	18
14		Práctica 5 21	22	Práctica 5 23	Sesión 14 24	25

Figure 14. Calendario Resumido

- Semanas del curso :
 - 15 semanas: Duración del curso 2 de Septiembre al 17 de Diciembre
- [calendario exámenes](https://www.unavarra.es/sites/grados/informatica-y-telecomunicacion/ingenieria-informatica/evaluacion.html#cCentralUPNA) [<https://www.unavarra.es/sites/grados/informatica-y-telecomunicacion/ingenieria-informatica/evaluacion.html#cCentralUPNA>]
- [otoño 2023](http://www2.unavarra.es/gesadj/CyD1/ETSIIT/Examenes/Grados/Otono/240_otono.pdf) [http://www2.unavarra.es/gesadj/CyD1/ETSIIT/Examenes/Grados/Otono/240_otono.pdf]

1.10. Ejercicios mediante resolución de problemas

- Realización de ejercicios básicos a lo largo de cada capítulo del temario incluidos en los apuntes.
- Ejercicios tipo examen.

1.11. Prácticas

1.11.1. Memorias

- Cinco Guiones :aulario virtual
 - Prácticas Individuales.
 - Memorias :
 - La entrega de la memoria a través del servidor de miaulario se realizará antes de la siguiente sesión de prácticas en la fecha indicada y publicada por el profesor.
 - La memoria es un documento único en formato PDF.
 - El nombre del fichero debe ser apellido1_apellido2_tituloguionpractica.pdf
 - Contenido de la memoria.
 - El programa descrito en pseudocódigo
 - El código fuente en lenguaje ensamblador debidamente comentados en coherencia con el programa en pseudocódigo.
 - No se valora el estilo de la memoria sino su contenido ya que interesa que sirva como apuntes para las pruebas de evaluación.
-  • Imprescindible tomar notas dentro y fuera del laboratorio
-  • Salvar todo el trabajo en un pendrive o enviarlo por correo
-  • Borrado automático a diario del contenido del disco duro.

1.12. Recursos Informáticos

1.12.1. UPNA

servicio informático

- Servicio Informático UPNA [<https://www.unavarra.es/servicio-informatico/servicios/pdi?languageId=100000>]
 - Reserva Aulas [<http://www2.unavarra.es/gesadj/servicioInformatico/usuario/aulas/GERES.pdf>]
 - Aulas VDI [<https://www.unavarra.es/servicio-informatico/servicios/pdi?opcion=23>]
 - Manual Acceso Remoto VDI [<https://www2.unavarra.es/gesadj/servicioInformatico/usuario/aulas/AccesoRemotoVDI.pdf>]
 - Escritorio Virtual [<https://vdibroker.unavarra.es/>]
- Laboratorio Remoto de Computación
 - acceso general [<https://laboratoriosvirtuales.unavarra.es/>] → ¿?

laboratorio remoto ARM/FPGA

- No se contempla la utilización del laboratorio remoto ARM/FPGA dentro del plan de la asignatura pero se encuentra disponible para quien desee utilizarlo, para lo cual es necesario abrir una cuenta de acceso.
- [acceso fpga](https://laboratoriosvirtuales.unavarra.es/fpga/#/) [<https://laboratoriosvirtuales.unavarra.es/fpga/#/>] → laboratorio remoto FPGA-Raspberry
 - Guacamole login → credenciales UPNA
 - Acceso a la tarjeta DE1-SoC (ARM) → login → Consola shell linux
 - Acceso a la tarjeta Raspberry (ARM) → login → Consola shell linux
 - Acceso a la tarjeta Xilinx (ARM) → login → Consola shell linux
- [Empresa Colaboradora Labsland](https://labsland.com/es/) [<https://labsland.com/es/>]: Instancias DE1-SoC. Interfaz Web.
 - Se instalará su acceso a través de miaulario mediante el botón EDA.
 - Herramientas de diseño dispositivos digitales

1.12.2. Estaciones de Trabajo: 32 y 64 bits

Arquitectura



Si la estación de trabajo particular de un alumno es Ubuntu (64 bits) sobre un procesador x86-64 no hay ningún problema para la ejecución de programas con una arquitectura de x86-64, en cambio, los programas a desarrollar en la asignatura utilizan una arquitectura de 32 bits que para poder compilar y ejecutar dichos programas es necesario

Qué instalar

- Es necesario disponer de un PC con una CPU cuya arquitectura sea x86-64 ó amd64 y un Sistema Operativo [linux](https://linuxfoundation.org/) [<https://linuxfoundation.org/>]/[https://www.gnu.org/software/\[GNU\]](https://www.gnu.org/software/[GNU]) mediante la distribución distribución [Ubuntu](https://ubuntu.com/download) [<https://ubuntu.com/download>]. La distribución Ubuntu que sea reciente con una versión superior a la 18. Se recomienda la distribución Ubuntu ya que es la distribución instalada en el laboratorio de informática donde se realizan las prácticas.
 - El Sistema operativo puede estar instalado de forma nativa o virtual (VMware, Virtualbox, [Virtio](https://libvirt.org/) [<https://libvirt.org/>], etc).
- Para la realización de las prácticas es necesaria una instalación mínima con las herramientas de programación de bajo nivel: **binutils**, Compilador **gcc**, Debugger **gdb**, Editor **vim**
- En Ubuntu:
 - realizar la instalación de los paquetes mediante el comando `sudo apt-get install binutils gcc gcc-multilib gdb vim`
 - comprobar la instalación de los paquetes mediante el comando `dpkg -l binutils gcc gcc-multilib gdb vim` que el estado de cada uno de los programas es *ii*
 - comprobar las versiones: `gcc --version && as --version && ld --version && gdb --version && vim --version`

- Para la edición de los programas en los lenguajes C y ensamblador ASM se podrá utilizar además del editor Vim, el editor preferido (nano,sublime,gedit,kate,emacs,etc..) y/o el IDE preferido (Visual Studio Code, Eclipse, etc..)

1.12.3. Registrarse

miaulario

- [Miaulario](https://miaulario.unavarra.es/portal) [<https://miaulario.unavarra.es/portal>]

Github

- Es necesario tener una cuenta abierta en la plataforma de repositorios [Github](https://github.com/) [<https://github.com/>]

Google

- [Google account](https://support.google.com/accounts/answer/27441) [<https://support.google.com/accounts/answer/27441>]
- [Navegador Google Chrome](https://www.google.com/chrome/) [<https://www.google.com/chrome/>]: permite seleccionar el idioma

1.13. Grupos de Prácticas

- 3 Grupos de prácticas en castellano: PG1,PG2,PG1G2

PG1	PG2	PG1G2
Alén Urra, Saúl	Elcid Beperet, Iker	Goikoetxea Macua, Jon
Alonso Gómez, Ana	Fernandez Illera, Iker	Goñi Lara, Iker
Altamirano Trujillo, Ruth N.	Fernández Picorelli, Marina	Isturitz Sesma, Eneko
Álvarez Alonso, Markel	Flamarique Arellano, Aritz	Orradre Berdusán,
Joaquín		
Andreu Mangado, Alvaro	Fortun Iñurrieta, Lucas N.	Pascal Alegria, Xabier
Apesteguía Vázquez, Alejandro	Gallo Ansa, Borja	Portuondo Varona, Helen
Arriazu Muñoz, Jon	Garatea Larrayoz, Iñaki	Ripoll Baños, Izar
Ayechu Garriz, Santiago	Garcia Vilas, Jorge Daniel	Romero Sádaba, Irene
Azcona Furtado, Lucía	Gil Gil, Santiago	Ruiz de Gopegui Rubio,
Paula		
Aznarez Gil, Iñigo	Gómez Ciganda, Iker Javier	Sola Bienzobas, Fermín
Baigorrotegui Gil, Oier	Granda Saritama, Anthony D.	Solaegui Garralda,
Beñat		
Bayona Restrepo, Karol Julian	Hualde Romero, Israel	Taberna Maceira, Alain
Bellera Alsina, Alberto	Iribarren Ruiz, Beñat	Zamboran Maldonado,
Andrea		
Calleja Pascual, Vidal	Juanotena Ezkurra, Joseba	Martínez Sesma, Álvaro
Cascan Ustarroz, Eduardo	Juárez Jiménez, Adrián	Meléndez Uriz,
Alejandro		
Cerezo Uriz, Iñaki	Labat García, Pablo	Mellado Ilundain, Iñaki
Chacón Flores, Malena Paola	Labiano Garcia, Martin J.	Molina Puyuelo,
Alejandro		
Cordido Pérez, Elena	Larrayoz Díaz, Asier	Monreal Ayanz, Ander
Couceiro Eizaguirre, Javier	Larrayoz Urra, Carlota	Moral Garcia, Haizea

Cuello Tejero, Jorge de la Ossa Goñi, Miguel Díaz Ochoa, Alex Misael Dobromirova Karailieva, Z. Ezponda Igea, Eduardo Felipe Goicoechea Elío, María	Latasa Sancha, Iván Liébana Revuelta, Diego Longás Saragüeta, Endika J.	Oroz Azcárate, Ángel Pérez Álvarez, Ángel Redrejo Fernández,
Sergio	Lumbreras Corredor, Imanol Martínez Arpón, Alain	Saiz Larraz, Eder Santos Garzon, Jaider
Caleb	Martínez de Goñi, Unai	Sanz Sanz, Iván Sola Alba, Alejandro Tellechea Zamanillo,
		Turrillas Remiro, Ethan Urroz Velasco, Ibai Vadillo Navarro, Asier Yarhui Sarate, Yerson
		Zheng , Yushan

1.14. Metodología

- Teoría, Ejercicios, Prácticas y Exámenes.

Table 1. metodología

Metodología - Actividad	Horas Presenciales	Horas no presenciales
A-1 Clases magistrales	24	
A-2 Estudio autónomo		30
A-3 Sesiones prácticas	16	
A-4 Programación / experimentación u otros trabajos en ordenador / laboratorio		20
A-5 Resolución de problemas, ejercicios y otras actividades de aplicación		12
A-6 Aprendizaje basado en problemas y/o casos	14	
A-7 Elaboración de trabajos y/o proyectos y escritura de memorias		11
A-8 Lectura de Guiones, preparación de presentaciones de trabajos, proyectos, etc...		15
A-9 Actividades de Evaluación		6

Metodología - Actividad	Horas Presenciales	Horas no presenciales
A-10 Tutorías	2	
Total	62	88

- La asignatura se desdobra en 2 partes
 - Parte I: Temas 1-6 (conceptos básicos)
 - Parte II: Temas 7-9 (conceptos avanzados y casos prácticos)

1.14.1. Distribución de créditos

- Distribución de créditos:
 - total créditos: 6 ECTS
 - total horas: 6x25 : 150 horas
 - horas presenciales: 62 horas
 - clases : 24 horas. Durante 12 semanas (2horas/semana) de 15 semanas totales del curso
 - prácticas de laboratorio: 16 horas. Durante 8 semanas (2horas/semana)
 - problemas : 14 horas. Durante 7 semanas (2horas/semana)
 - evaluación: 6 horas. Durante 3 semanas (2horas/semana) de 15 semanas totales del curso
 - tutorías : 2 horas
 - horas no presenciales: 88 horas

1.14.2. Distribución de créditos de las Prácticas

- Cada alumno tendrá 16 horas de prácticas en sesiones de 2 horas: 12 en el laboratorio y 4 para la realización de Memorias.

1.15. Evaluación

- [Web Estructura de Ordenadores](http://www.unavarra.es/ficha-asignaturaDOA?languageId=100000&codPlan=240&codAsig=240306) [http://www.unavarra.es/ficha-asignaturaDOA?languageId=100000&codPlan=240&codAsig=240306]
 - prácticas: 2 pruebas parciales. Primera prueba parcial el 28 de Octubre (práctica 1^a y 2^a). Segunda prueba parcial el 12/01/2023 (prácticas 3^a, 4^a y 5^a). Nota mínima de 4 en cada una de las dos pruebas parciales.
 - teoría: 2 pruebas parciales. Primera prueba parcial el 14 de Octubre . Segunda prueba parcial en la convocatoria final ordinaria el 12/01/2023. Nota mínima de 4 en cada una de las dos pruebas parciales.
 - Distribución del valor de cada una de las pruebas evaluadoras: 15%(asistencia y actitud en clase y laboratorio)+35%(prácticas y trabajos)+35%(conceptos y ejercicios)+15%(programación en papel)



OBLIGATORIEDAD de las prácticas y de las pruebas teóricas:

- Asistencia a las prácticas en el laboratorio: Es **obligatorio** asistir al 87.5% de las horas de prácticas en el laboratorio.
- Entrega de las memorias de prácticas: Es **obligatorio** entregar el 100% de las memorias dentro del plazo establecido en la fecha habilitada en el servidor de miaulario. No se reciben memorias de prácticas ni de tareas fuera del plazo fijado por el servidor de miaulario.
- La obligatoriedad de la asistencia al 87.5% de las horas de prácticas así como la entrega de memorias en el plazo y medio establecido es condición necesaria para poder superar la asignatura.

1.16. Exámenes

- Los exámenes de teoría tendrán preguntas teóricas sobre conceptos vistos en clase y ejercicios con una distribución "aproximada" en el primer parcial del: 20% preguntas sobre conceptos teóricos y un 80% de ejercicios ; y para el segundo parcial: 60% preguntas sobre conceptos teóricos y un 40% de ejercicios.
- El examen de programación en papel se realizará sin ordenador y con las hojas de referencia de las instrucciones en el lenguaje ensamblador, el depurador GDB y las sentencias del lenguaje C.
- El examen de prácticas en el laboratorio se realizará con las memorias de prácticas y las hojas de referencia y sin poder utilizar ninguna información electrónica ni de forma remota (acceso a internet) ni de forma local (pendrive USB,etc).
- Calendario:

1º parcial teoría: Temas 1,2,3,4,5 y 6: 14/10/2022

1º parcial prácticas: Guiones 1 y 2: 28/10/2022

convocatoria ordinaria: 2º parcial (Temas 7,8 y 9): 12/01/2023 08:00

convocatoria recuperación: (Temas 1,2,3,4,5,6,7,8 y 9 y prácticas): 28/01/2023 08:00

Chapter 2. Arquitectura Von Neumann

2.1. Arquitectura Von Neumann

- Calcular la suma $\sum_{i=1}^N i = N(N + 1)/2$

2.1.1. Temario

2. Arquitectura Von Neumann:
 - a. CPU
 - b. Memoria
 - c. Entrada / Salida

2.1.2. Contexto Histórico

Antecedentes

- 1833: Charles Babbage → Diseña la 1^a Computadora mecánica
- 1890: Máquina tabuladora de Herman **Hollerith**. Censo en USA. IBM (1925)
- 1936: Alan Turing → Algoritmia y concepto de máquina de Turing. Máquina código Enigma.
- **Segunda Guerra Mundial 1939-1945**
- 1944: USA, IBM Computadora electromecánica Harvard Mark I
- 1944: Colossus (Colossus Mark I y Colossus Mark 2). Decodificar comunicaciones.

ENIAC

- 1947: En la Universidad de Pensilvania (laboratorio de investigación de balística para la artillería) se construye la **ENIAC** (Electronic Numerical Integrator And Calculator)
 - Ecuaciones diferenciales sobre balística (angle = f (location, tail wind, cross wind, air density, temperature, weight of shell, propellant charge, ...))
 - Computadora electrónica (no mecánica) de **propósito general**.
 - Memoria: Sólo 20 acumuladores → flip-flops hechos con triodos
 - 18,000 tubos electrónicos ó válvulas de vacío
 - Programación manual de los interruptores
 - 100,000 instrucciones por segundo
 - 300 multiplicaciones por segundo
 - 200 kW
 - 13 toneladas y 180 m²

EDVAC

- 1951: En la Universidad de Pensilvania (J. Presper Eckert y John William Mauchly) comienza a operar la **EDVAC** (Electronic Discrete Variable Automatic Computer), concebida por **John von Neumann**, que a diferencia de la ENIAC no era decimal, sino binaria, y tuvo el primer **programa** (no solo los datos) diseñado para ser **almacenado**: STORED PROGRAM COMPUTER → program can be manipulated as data.
 - 500000\$
 - La EDVAC poseía físicamente casi 6000 válvulas termoiónicas y 12 000 diodos de cristal. Consumía 56 kilowatts de potencia. Cubría 45,5 m² de superficie y pesaba 7850 kg.
 - Arquitectura:
 - un lector-grabador de cinta magnética
 - una unidad de control con osciloscopio, una unidad para recibir instrucciones del control
 - la memoria : 2000 word storage "mercury delay lines" → poca fiabilidad
 - una unidad de aritmética de coma flotante en 1958.

IAS

- 1946-1952 : **IAS** (Institute Advanced Studies) mainframe :
 - Evolución de EDVAC: unidad de memoria principal y secundaria tambor magnético.
 - Memoria Selectron: almacenamiento capacitivo → carga electrostática

Posterior

- 1952: **UNIVAC I** (UNIVersal Automatic Computer I) was the first commercial mainframe computer. Evolución de la máquina tabuladora de Hollerith aplicado al procesado del censo en USA.
- 1952: IBM 701, conocido como la "calculadora de Defensa" mientras era desarrollado, fue la primera computadora científica comercial de IBM → primer lenguaje **ENSAMBLADOR**.
- 1964: mainframe (computadora central) **IBM 360** → primer computador con ISA (microprogramación) → compatibilidad
 - tecnología híbrida entre componentes integrados discretos de silicio y otros componentes → no "circuitos" integrados.
 - Basic Operating System/360 (BOS/360), Disk Operating System/360 (DOS/360)

Tecnología de Semiconductor

- 1947: en los Laboratorios Bell, John Bardeen, Walter H. Brattain y William Shockley inventan el **transistor**.
- 1958: Kilby , primer circuito integrado en germanio.
- 1957: Robert Norton Noyce, cofundador de Fairchild Semiconductor, primer circuito integrado planar

- 1968: Robert Norton Noyce y Gordon Moore fundan Intel.
- 1971: Intel 4004 → cpu integrada en silicio → 8 bits

2.2. Institute Advanced Machine (IAS) : Arquitectura

2.2.1. Referencia

- [The Von Neumann Machine \[https://es.wikipedia.org/wiki/M%C3%A1quina_de_von_Neumann\]](https://es.wikipedia.org/wiki/M%C3%A1quina_de_von_Neumann)

2.2.2. Ejemplo del Programa sum1toN

Código binario para calcular $\sum_{i=1}^5 i$

Address	Data	Comments
0	01 005	loop: S(x)->Ac+ n ;load n into AC
0	0F 002	Cc->S(x) pos ;if AC >= 0, jump to pos
1	00 000	halt ;otherwise done
1	00 000	.empty :a 20-bit 0
2	05 007	pos: S(x)->Ah+ sum ;add n to the sum
2	11 007	At->S(x) sum ;put total back at sum
3	01 005	S(x)->Ac+ n ;load n into AC
3	06 006	S(x)->Ah- one ;decrement n
4	11 005	At->S(x) n ;store decremented n
4	0D 000	Cu->S(x) loop ;go back and do it again
5	00 000	n: .data 5 ;will loop 6 times total
5	00 005	
6	00 000	one: .data 1 ;constant for decrementing n
6	00 001	
7	00 000	sum: .data 0 ;where the running/final total is kept
7	00 000	
8	00 000	
8	00 000	

Figure 15. Código Máquina del programa sum1toN en la máquina IAS

Programación Imperativa

- Paradigma:
 - Paradigma imperativo ó estructural : el algoritmo se implementa desarrollando un programa que contiene las ORDENES que ha de ejecutar la máquina
 - A diferencia de la programación declarativa: el algoritmo implementa QUÉ queremos que haga la computadora, no el COMO, no directamente las órdenes que ha de ejecutar.
 - Por ejemplo la operación $\sum_{i=1}^5 i$, se puede describir en python como:

```
sum(range(5,0,-1))
```

Contenido de la Memoria: Datos e Instrucciones

- La computadora IAS se programaba directamente en *lenguaje máquina*, no tenía un lenguaje simbólico como el lenguaje ensamblador.
- Lenguaje Máquina: Código Binario
- Edición del código binario mediante tarjetas perforadas o cintas magnéticas a través de una consola.
- Tipo de información contenido en la memoria: DATOS e INSTRUCCIONES
 - Ejemplo de datos: números enteros +3278,+5,-1,-6592,...
 - Ejemplo de instrucciones:
 - LOAD M(8) : cargar en el registro acumulador el contenido de la posición 8 de memoria
 - ADD M(3) : sumar al registro acumulador el contenido de la posición 3 de la memoria
 - JMP M(100): saltar a la posición 100 de la memoria
 - etc
- Concepto de programa **almacenado** : Instrucciones binarias y Datos binarios almacenados en la **Unidad de Memoria**
 - Fue la gran novedad de la arquitectura Von Neumannns
 - Es necesario CARGAR el módulo binario en la MEMORIA de la computadora para que quede almacenado.
- Programación secuencial: Las instrucciones se ejecutan secuencialmente según están almacenadas en la memoria...mientras no se ejecute una instrucción explícita de salto que rompa la secuencia.

Arquitectura: Instruction Set Architecture (ISA)

- Para poder analizar el programa es necesario no solo conocer el lenguaje binario de la máquina sino conocer su ARQUITECTURA. La arquitectura de una computadora es el WHAT de la máquina, es decir, QUE instrucciones es capaz de ejecutar la máquina, para lo cual es necesario conocer la ARQUITECTURA DEL REPERTORIO DE INSTRUCCIONES (Instruction Set Architecture ISA):
 - el repertorio de instrucciones: operaciones y modo de acceso a los datos
 - jerarquía de memoria: memoria principal y registros
 - formato de instrucciones y datos



la ISA es el primer nivel de **abstracción** del hardware físico de la computadora.

2.3. Estructura de la computadora IAS

2.3.1. Módulos



La Estructura es el HOW de la máquina. De qué hardware disponemos para poder

ejecutar las instrucciones máquina definidas por la arquitectura.

- Hardware con Estructura **Modular**:

- CPU-Memoria-I/O-Bus

- Jerarquía de Memoria: 2 niveles : Memoria Principal (externa a la CPU) y Registros (internos a la CPU)

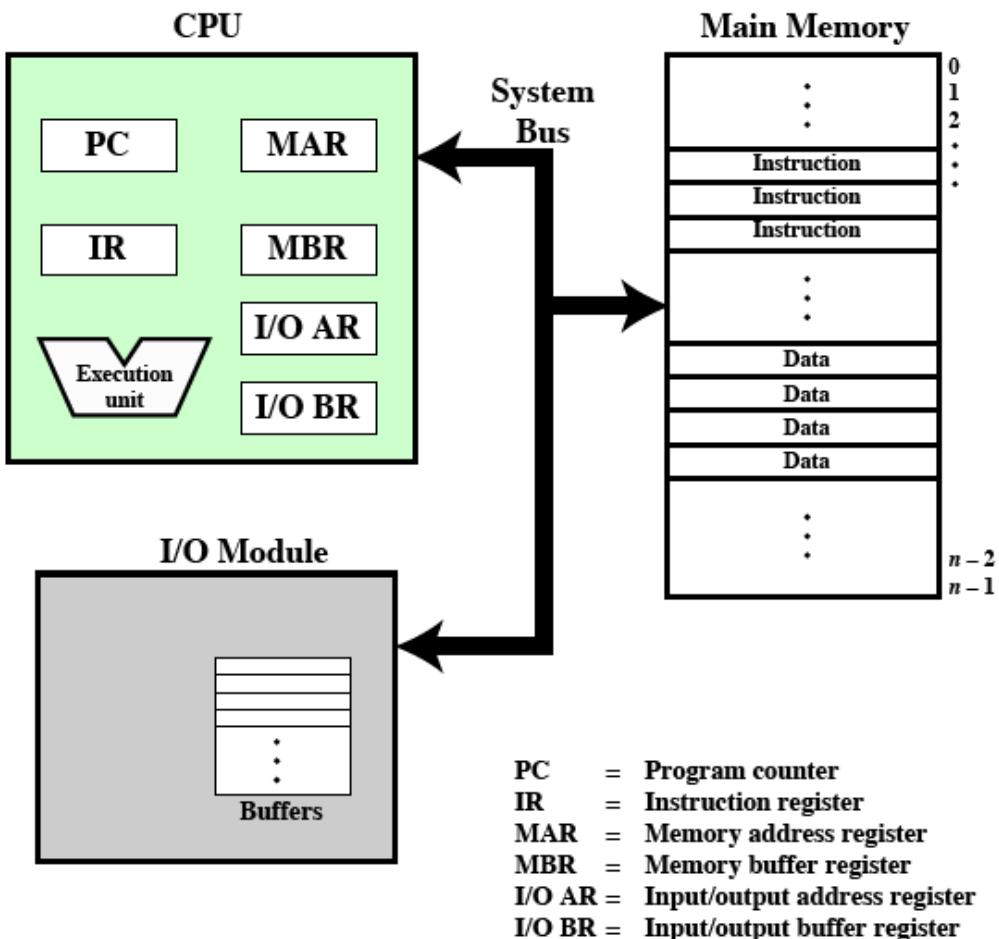


Figure 16. Arquitectura de la máquina IAS

- Arquitectura Interna de la CPU : Microarquitectura

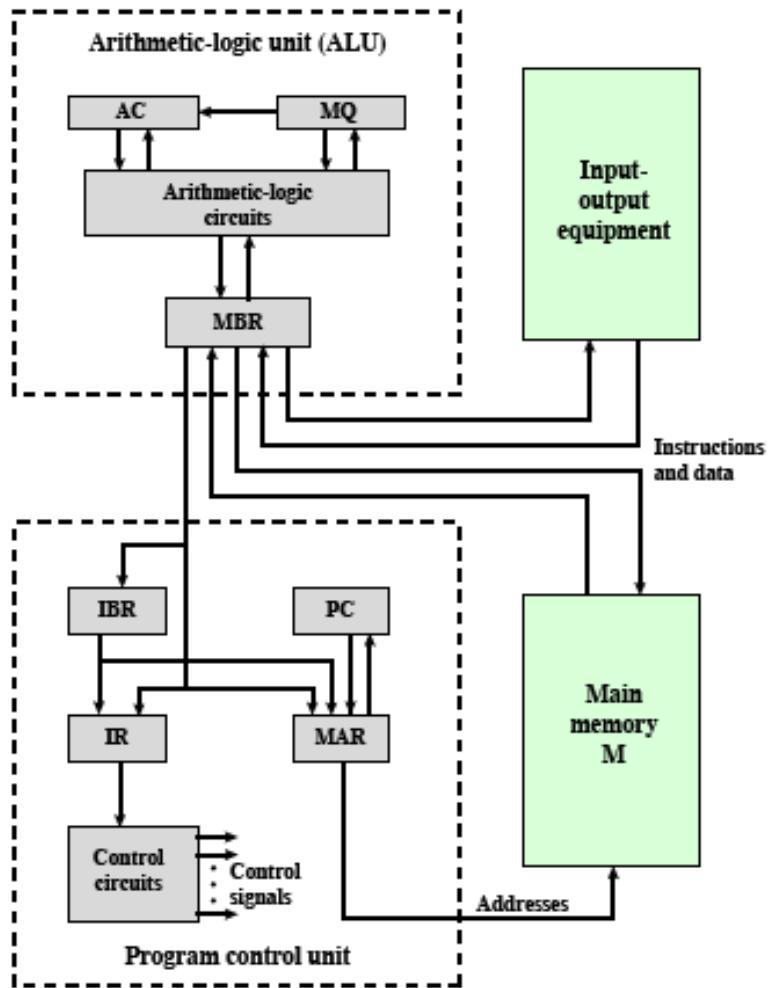


Figure 2.3 Expanded Structure of IAS Computer

Figure 17. Estructura de la máquina IAS

2.3.2. Unidad Central de Proceso (CPU)

- CPU:

- El Funcionamiento de la CPU está dividido 3 FASES: Captura, Interpreta y Ejecuta las instrucciones secuencialmente. A la secuencia de las 3 fases se le conoce con el nombre de **Ciclo de Instrucción**.

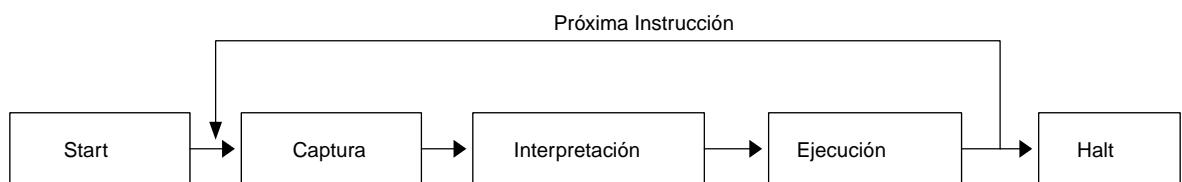


Figure 18. Ciclo de Instrucción

- Cada instrucción máquina de un programa es capturada, interpretada y ejecutada por la CPU y en ese orden. El circuito electrónico digital encargado de controlar que se realice dicha

secuencia es la **Unidad de Control** integrada en la CPU. La Unidad de Control da microordenes mediante señales electrónicas al subcircuito capturador, al subcircuito intérprete y al subcircuito ejecutor para que se lleven a cabo todas las fases del ciclo de instrucción de cada instrucción del programa almacenado en la memoria principal.

- Tres submódulos principales de la CPU:
 - Unidad de cálculo: Unidad Aritmético-Lógica (ALU)
 - Unidad de control: Circuito secuencial que implementa el Ciclo de instrucción dando las órdenes eléctricas a los distintos bloques (ALU, memoria principal, registros, buses, etc) en cada fase hasta completar el ciclo de instrucción.
 - Registros de memoria: En un registro se puede escribir o leer un dato o dos instrucciones.

2.3.3. Memorias

Memoria Principal

Espacio de DIRECCIONES

CONTENIDO

0x00000000	01010101010101010
0x00000001	01010101010101010
0x00000002	01010101010101010
0x00000009	
0x0000000a	
0x0000000f	

Figure 19. Direcccionamiento del contenido de la Memoria Principal

- Debe almacenar el programa a ejecutar en código binario.
- La CPU es el único módulo que tiene acceso a la memoria principal.
- Las instrucciones y datos del programa se almacenan secuencialmente.
- Almacena el programa en dos *secciones*: Sección de Datos y Sección de Instrucciones
- Organizada en Palabras accesibles aleatoriamente. Random Access Memory.
- Dinamismo: Lectura/Escritura de datos e instrucciones
- En la máquina IAS las direcciones de memoria apuntan a palabras de 40 bits que pueden almacenar ó un dato de 40 bits o dos instrucciones de 20 bits cada una.
- Random Access Memory (RAM): direccionable cada posición de memoria.
- Shared Memory: memoria compartida entre datos e instrucciones. También comparten el bus de acceso a memoria.

- Capacidad para $2^{12}=4K$ palabras con 40 bits para cada palabra.
 - $4K \times 40\text{bits} = 4K \times 5\text{Bytes} = 20\text{KBytes}$
 - En cambio la memoria física disponible en esa época era de : 1024 palabras de 40 bits = 5 KBytes (Libro "The Computer from Pascal to von Neumann", Herdman Godstine, pg314, ISBN 0-691-02367-0). Limitación tecnológica.

Registros de la CPU

- Memoria interna a la CPU: 2 tipos de registros: accesibles por el programador y no accesibles por el programador.
- AC y AR/MQ: Acumuladores de la ALU. Multiplier/Quotient .Son los únicos registros accesibles por el programador.
- Registros NO accesibles por el programador: todos los registros de la Unidad de Control: MBR,PC,IR,IBR,MAR
 - MBR: Selectron Register ó Memory Buffer Register *MBR* ó Data Buffer Register *DBR*. Tamaño de 40 bits. Almacena el dato o par de instrucciones leídas de la memoria resultado de la fase de captura del ciclo de instrucción ó almacena el dato a escribir en la memoria resultado de la última fase del ciclo de instrucción.
 - PC: Control Counter: Program Counter (PC) o Instruction Pointer (IP). Tamaño de 12 bits. Apunta a la siguiente instrucción a capturar
 - IR: Control Register: también denominado Instruction Register *IR*. Tamaño de 20 bits. Almacena la instrucción capturada durante el ciclo de instrucción
 - IBR: Instruction Buffer Register: Almacena la segunda instrucción capturada durante el ciclo de instrucción. Tamaño de 20 bits. Observar que esto significa que en la fase de captura se capturan dos instrucciones simultáneamente.
 - MAR: Memory Address Register: current Memory Address. Tamaño de 12 bits. Apunta al operando o instrucción a capturar durante la primera fase del ciclo de instrucción.
- Su tamaño define lo que se conoce como "word size" de la arquitectura. La máquina IAS tiene una arquitectura de 40bits ó un Word de 40 bits

2.3.4. Bus

- Conjunto de hilos o pistas metálicas paralelas para conectar dos dispositivos electrónicos. Todo el mundo ha tenido en sus manos un cable USB el cual contiene un bus USB (Universal Serial Bus).
- Bus del Sistema:
 - Interconexión CPU-Memoria Principal: transferencia de datos e instrucciones.
 - Bus de Datos (40 hilos), Bus de Direcciones (12 hilos) y Bus de Control (Lectura/Escritura) (1 hilo). En total 53 hilos o pistas son necesarios para interconectar la CPU y la Memoria Principal.

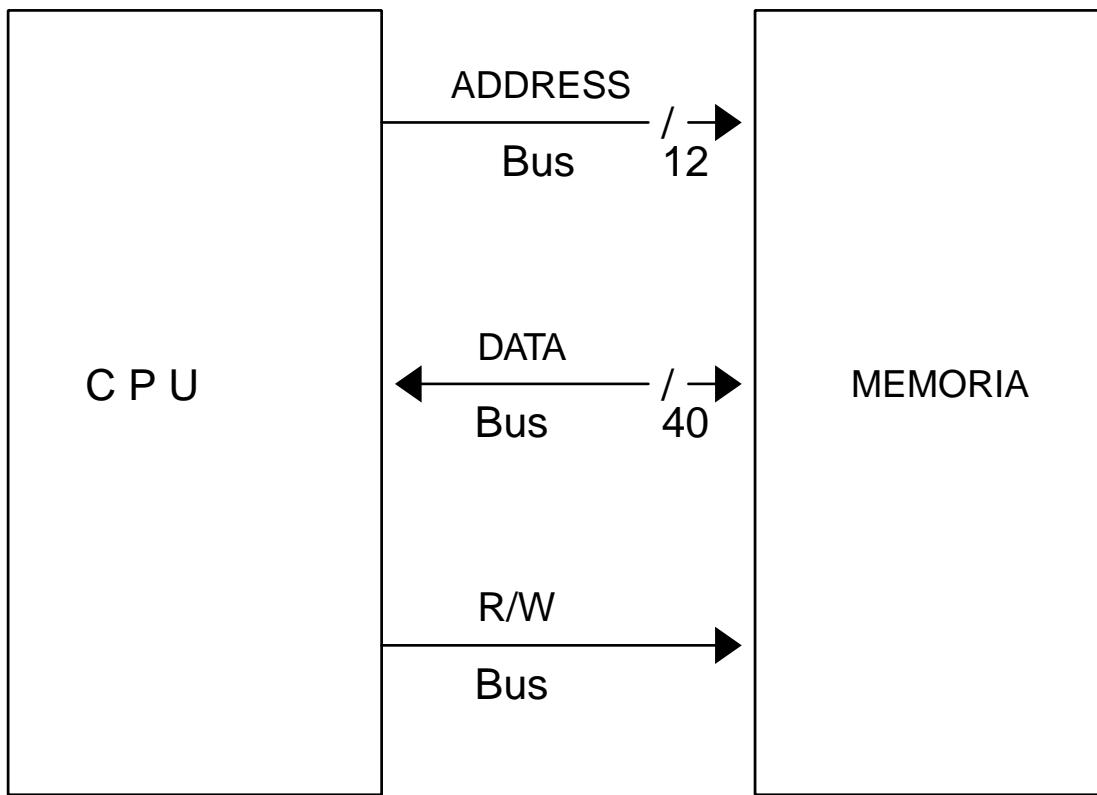


Figure 20. Conexión CPU-Memoria Principal

2.3.5. Input Output (I/O)

- Las entradas y salidas de una computadora son necesarias para poder operar con ellas, bien el programador o bien otras máquinas. Para acceder externamente a la computadora son necesarios los periféricos como teclados, pantallas, etc
 - En la máquina IAS el programa se escribe en tarjetas perforadas (Punch Cards). Tarjetas para Datos y tarjetas para instrucciones. Es necesario cargar los datos en la memoria antes de la ejecución del programa.
 - tarjetas perforadas, consola, tambores magnéticos, cintas magnéticas, cargador de memoria mediante un lector de tarjetas , display mediante tubos de vacío, etc.. → tecnología obsoleta.
 - No tendremos en cuenta el módulo I/O y nos centraremos en los módulos CPU-Memoria Principal.

2.3.6. Animación del Ciclo de Instrucción

- [Animación del ciclo instrucción](https://www.youtube.com/watch?v=04UGopESS6A) [https://www.youtube.com/watch?v=04UGopESS6A]

2.4. ISA: Arquitectura del Repertorio de Instrucciones de la máquina IAS

2.4.1. Formato de los datos e Instrucciones de la Computadora IAS

- Arquitectura de la Memoria
 - Word
 - 40 bits : 1 dato ó 2 instrucciones
 - Datos
 - Números Enteros en formato Complemento a 2.
 - *Data Format*

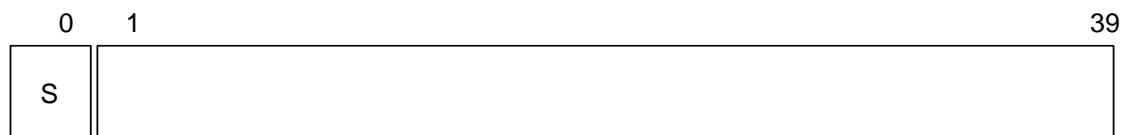


Figure 21. Formato de los datos

- Observar que el bit con la numeración cero es el de la izda.
- Instrucciones
 - Código de Operaciones de 8-bit seguidos de un operando de 12-bit (data address)
 - *Instruction Format*

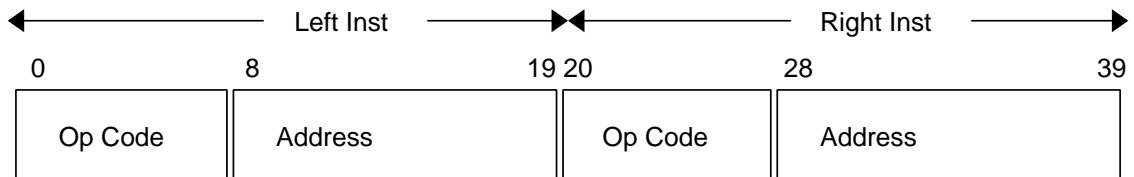


Figure 22. Formato de las instrucciones

- Definimos **un sólo operando** o ninguno en cada instrucción
- *Accumulator Based Architecture*
- Una operación que requiera dos operandos implicitamente hace referencia a un operando almacenado en el *acumulador*
- Observar que el bit con la numeración cero es el de la izda.
- La instrucción de la izda (0-19) se carga en los registros internos de la CPU, el código de operación IR y el campo de operación en MAR .
- La instrucción de la derecha (20-39) se carga en el registro interno de la CPU, IBR .

- Modo de direccionamiento del Operando: Referencia del Operando. Esta arquitectura se diseño con único modo de direccionamiento denominado "Direccionamiento Directo" donde en el campo de operando de la instrucción se especifica la **dirección de memoria del operando**
- Contenido de la Memoria
 - Las direcciones de memoria las visualizamos dobles ya que hacen referencia a la primera a los 20 bits LSB y la segunda a los 20 bits MSB de una palabra de memoria de 40 bits.
 - Observar que en la columna data están las dos secciones: sección de instrucciones y sección de datos
 - En la arquitectura von Neumann datos e instrucciones comparten el mismo espacio de direcciones de memoria.

Address	Data	Comments
0	01 005	loop: S(x)->Ac+ n ;load n into AC
0	0F 002	Cc->S(x) pos ;if AC >= 0, jump to pos
1	00 000	halt ;otherwise done
1	00 000	.empty ;a 20-bit 0
2	05 007	pos: S(x)->Ah+ sum ;add n to the sum
2	11 007	At->S(x) sum ;put total back at sum
3	01 005	S(x)->Ac+ n ;load n into AC
3	06 006	S(x)->Ah- one ;decrement n
4	11 005	At->S(x) n ;store decremented n
4	0D 000	Cu->S(x) loop ;go back and do it again
5	00 000	n: .data 5 ;will loop 6 times total
5	00 005	
6	00 000	one: .data 1 ;constant for decrementing n
6	00 001	
7	00 000	sum: .data 0 ;where the running/final total is kept
7	00 000	
8	00 000	
8	00 000	

Figure 23. Código Maquina sum1toN de la máquina IAS

2.4.2. Repertorio ISA

Lenguaje RTL

- Información sobre el lenguaje de transferencia entre registros (RTL) en el [Apéndice](#)

Repertorio de la máquina IAS

- Instruction Set Architecture (ISA): Definición y características del conjunto de instrucciones. Arquitectura del Repertorio de Instrucciones.
- En la versión original no había código ensamblador, se programaba directamente en lenguaje máquina.
 - En la tabla adjunta, en la segunda columna, los **MNEMONICOS** (LOAD,ADD,SUB,etc) de las operaciones de las instrucciones se corresponden con los diseñados por el libro de texto de William Stalling. En la primera y última columnas las operaciones se simbolizan mediante un lenguaje de transferencia entre registros.

- Selectron es el nombre de la tecnología utilizada para la Memoria Principal.
 - La notación $S(x)$ equivale en notación RTL a $M[x]$
- R es el registro AR que W.Stalling denomina registro MQ.

Table 2. Instruction Set

Instruction name	Op	Description	RTL
$S(x) \rightarrow Ac+$	LOAD M(X)	1 copy the number in Selectron location x into AC	$AC \leftarrow M[x]$
$S(x) \rightarrow Ac-$	LOAD -M(X)	2 same as #1 but copy the negative of the number	$AC \leftarrow \sim M[x] + 1$
$S(x) \rightarrow AcM$	LOAD M(X)	3 same as #1 but copy the absolute value	$AC \leftarrow M[x] $
$S(x) \rightarrow Ac-M$	LOAD - M(X)	4 same as #1 but subtract the absolute value	$AC \leftarrow AC - M[x] $
$S(x) \rightarrow Ah+$	ADD M(X)	5 add the number in Selectron location x into AC	
$S(x) \rightarrow Ah-$	SUB M(X)	6 subtract the number in Selectron location x from AC	
$S(X) \rightarrow AhM$	ADD M(X)	7 same as #5, but add the absolute value	
$S(X) \rightarrow Ah-M$	SUB M(X)	8 same as #7, but subtract the absolute value	
$S(x) \rightarrow R$	LOAD MQ,M(X)	9 copy the number in Selectron location x into AR	
$R \rightarrow A$	LOAD MQ	A copy the number in AR to AC	
$S(x)*R \rightarrow A$	MUL M(X)	B Multiply the number in Selectron location x by the number in AR. Place the left half of the result in AC and the right half in AR.	
$A/S(x) \rightarrow R$	DIV M(X)	C Divide the number in AC by the number in Selectron location x. Place the quotient in AR and the remainder in AC.	
$Cu \rightarrow S(x)$	JUMP M(X,0:19)	D Continue execution at the left-hand instruction of the pair at Selectron location x	
$Cu` \rightarrow S(x)$	JUMP M(X,20:39)	E Continue execution at the right-hand instruction of the pair at Selectron location x	
$Cc \rightarrow S(x)$	JUMP+ M(X,0:19)	F If the number in AC is ≥ 0 , continue as in #D. Otherwise, continue normally.	
$Cc` \rightarrow S(x)$	JUMP+ M(X,20:39)	10 If the number in AC is ≥ 0 , continue as in #E. Otherwise, continue normally.	

Instruction name	Instruction name	Op	Description	RTL
At → S(x)	STOR M(X)	11	Copy the number in AC to Selectron location x	
Ap → S(x)		12	Replace the right-hand 12 bits of the left-hand instruction at Selectron location x by the right-hand 12 bits of the AC	
Ap` → S(x)		13	Same as #12 but modifies the right-hand instruction	
L	LSH	14	Shift the number in AC to the left 1 bit (new bit on the right is 0)	
R	RSH	15	Shift the number in AC to the right 1 bit (leftmost bit is copied)	
halt		0	Halt the program (see paragraph 6.8.5 of the IAS report)	

- Instruction Set (William Stallings)

Table 2.1 The IAS Instruction Set

Instruction Type	Opcode	Symbolic Representation	Description
Data transfer	00001010	LOAD MQ	Transfer contents of register MQ to the accumulator AC
	00001001	LOAD MQ,M(X)	Transfer contents of memory location X to MQ
	00100001	STOR M(X)	Transfer contents of accumulator to memory location X
	00000001	LOAD M(X)	Transfer M(X) to the accumulator
	00000010	LOAD - M(X)	Transfer -M(X) to the accumulator
	00000011	LOAD M(X)	Transfer absolute value of M(X) to the accumulator
	00000100	LOAD - M(X)	Transfer - M(X) to the accumulator
Unconditional branch	00001101	JUMP M(X,0:19)	Take next instruction from left half of M(X)
	00001110	JUMP M(X,20:39)	Take next instruction from right half of M(X)
Conditional branch	00001111	JUMP + M(X,0:19)	If number in the accumulator is nonnegative, take next instruction from left half of M(X)
	00010000	JUMP + M(X,20:39)	If number in the accumulator is nonnegative, take next instruction from right half of M(X)
Arithmetic	00000101	ADD M(X)	Add M(X) to AC; put the result in AC
	00000111	ADD M(X)	Add M(X) to AC; put the result in AC
	00000110	SUB M(X)	Subtract M(X) from AC; put the result in AC
	00001000	SUB M(X)	Subtract M(X) from AC; put the remainder in AC
	00001011	MUL M(X)	Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ
	00001100	DIV M(X)	Divide AC by M(X); put the quotient in MQ and the remainder in AC
	00010100	LSH	Multiply accumulator by 2; that is, shift left one bit position
	00010101	RSH	Divide accumulator by 2; that is, shift right one position
Address modify	00010010	STOR M(X,8:19)	Replace left address field at M(X) by 12 rightmost bits of AC
	00010011	STOR M(X,28:39)	Replace right address field at M(X) by 12 rightmost bits of AC

Figure 24. IAS_Instruction_Set

2.4.3. Interfaz ISA

- La arquitectura del conjunto de instrucciones (ISA) define la **INTERFAZ** entre el Hardware y el Software de la máquina
 - Podemos tener dos CPU totalmente diferentes, p.ej AMD e Intel, pero si tienen la misma ISA serán máquinas compatibles desde el punto de vista del sistema operativo.
 - Concepto de familia: un mismo repertorio de instrucciones puede ser ejecutado por distintas computadoras
- **ISA de distintas máquinas** [http://en.wikipedia.org/wiki/List_of_instruction_sets]

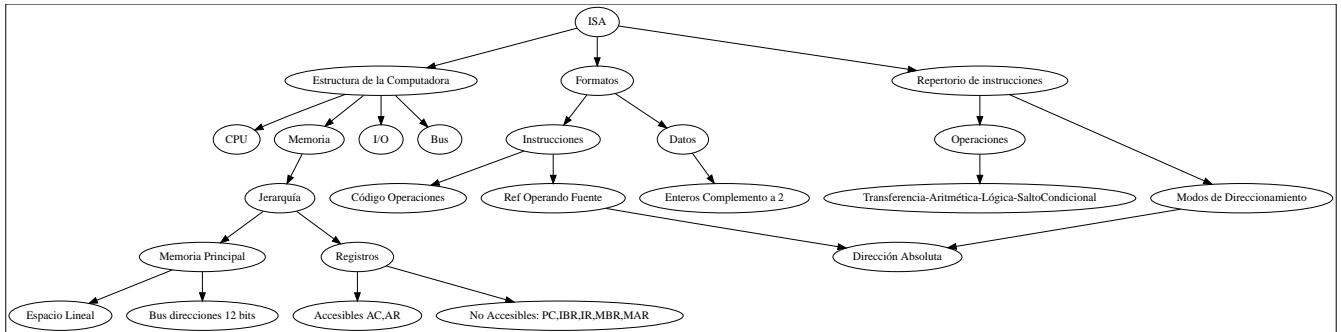


Figure 25. Instruction Set Architecture (ISA)

2.5. Programación en el Lenguaje Ensamblador IAS

2.5.1. Estrategia del Desarrollo de un Programa en Lenguaje Ensamblador

- Una vez entendido el problema que ha de resolverse mediante la programación, no se programa directamente el módulo fuente solución del problema sino que se va resolviendo describiendo el problema y el algoritmo solución en distintos lenguajes y en las siguientes fases:
 - Descripción del algoritmo en lenguaje "pseudocódigo".
 - Descripción del algoritmo mediante un organigrama o diagrama de flujo.
 - Descripción del algoritmo en lenguaje de transferencia entre registros RTL.
 - Descripción del algoritmo en lenguaje ensamblador propio de la computadora

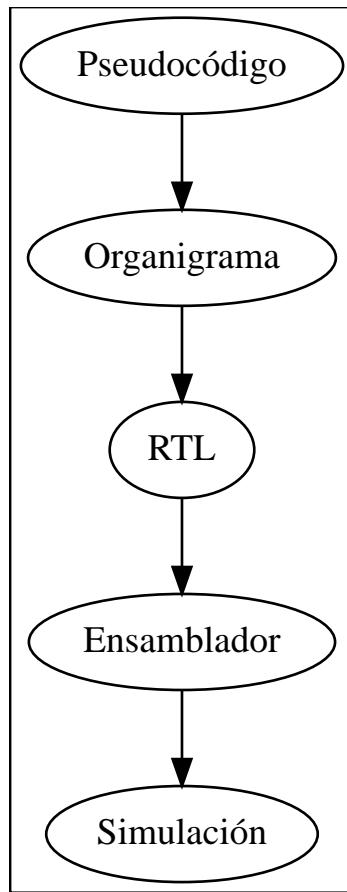


Figure 26. Fases de la Programación

El paso de una descripción en un lenguaje de alto nivel a bajo nivel se realiza en lenguaje RTL teniendo en cuenta la arquitectura de la computadora donde se ejecutará el lenguaje máquina. Cada instrucción de alto nivel habrá que traducirla en un bloque de instrucciones de bajo nivel



2.5.2. Ejemplo 1: sum1toN.ias

Enunciado

- Calcular la suma $\sum_{i=1}^N i = N(N + 1) / 2$

Pseudocódigo

- Descripción del algoritmo mediante expresiones modo texto en lenguaje NATURAL
- VARIABLES:
 - variable suma : almacena los resultados parciales y final
 - variable N : almacena el dato de entrada
 - variable i : almacena el sumando que varía en cada iteración
- Estructura del CODIGO imperativo:
 - La construcción de instrucciones básica es un bucle
 - El bucle cuenta las iteraciones en sentido descendente

- En cada iteración se genera un sumando "i" y se realiza la suma=suma+i
- Se inicializa "i"=N y en cada iteración i=i-1
- Se sale del bucle cuando i=-1

Organigrama

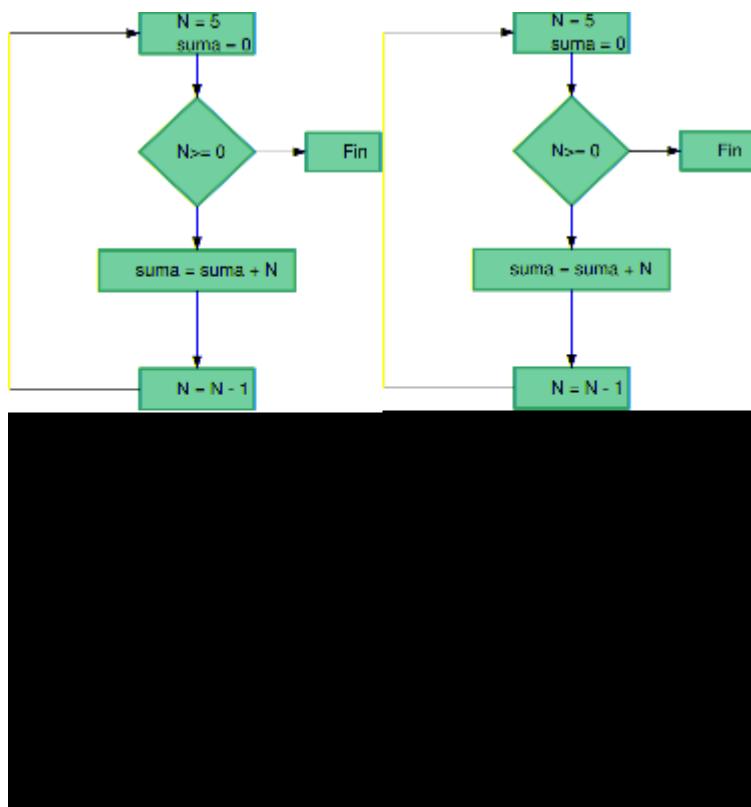


Figure 27. Organigrama: Diagrama de Flujo

RTL

- Descripción RTL para la máquina IAS orientada a acumulador

```

;CABECERA
;Descripcion en lenguaje RTL del algoritmo sum1toN

;SECCION DATOS :
; Declaracion de etiquetas, reserva de memoria externa, inicializacion
; Variables ordinarias
n: M[n] <- 5 ; variable sumando e inicializacion
suma: M[suma] <- 0 ; variable suma parcial y final

;SECCION INSTRUCCIONES
;Arquitectura orientada a Acumulador (AC)
;Registros accesibles : AC
    ; inicio bucle : suma y generacion de sumandos
bucle: AC <- M[n] ; cargar sumando
      AC>=0 : PC <- sumar ; si el sumando < 0 fin del bucle
      ; fin del bucle
      stop
  
```

```

; realizar la suma
sumar: AC <- AC + M[suma]
    M[suma] <- AC
    ; actualizar sumando
    AC <- M[n]
    AC <- AC - 1
    M[n] <- AC
    ; siguiente iteracion
PC <- bucle

```

Lenguaje ensamblador WStalling de la máquina IAS

- Módulo fuente sum1toN.ws

```

; CABECERA
; 1ª version : sum1toN_v1.ias
; Calcula la suma de una secuencia de numeros enteros: suma = 1+2+..+n
; dato de entrada : n
; dato de salida : suma
; Algoritmo : bucle de n iteraciones
;           Los sumandos se van generando en sentido descendente de n a 0
;           Se sale del bucle si el sumando es negativo -> -1
; Estructuras de datos : variables n y suma . Constante uno.
; Lenguaje ensamblador: William Stalling
; Arquitectura de la máquina IAS de Von Neumann

;;;;;;;;;; SECCION DE INSTRUCCIONES
;Arquitectura orientada a Acumulador (AC)
;Registros accesibles : AC
; algoritmo: bucle que genera la secuencia n, n-1, n-2,...0,-1 si n>=0
bucle: LOAD  M(n)      ; AC <- M[n]
        SUB    M(uno)     ; AC <- AC-M[uno]
        STOR   M(suma)    ; M[suma] <- AC
        JMP+   bucle      ; Si AC >= 0, salto a bucle
        ; fin del bucle
        HALT            ; stop

;;;;;;;;;;SECCION DE DATOS
; Declaracion de etiquetas, reserva de memoria externa, inicializacion.
; Variables ordinarias
n: .data 5 ; variable sumando
uno: .data 1 ; cte
suma: .data 0 ; sumas parciales y resultado final

```

- Se ha desarrollado la sección de datos para la reserva de memoria.
- Se ha realizado un BUCLE SENCILLO ya que el bucle es la construcción necesaria en el algoritmo final.

- Se ha realizado la operación RESTA ya que es una operación necesaria en el algoritmo final.
- Se ha COMENTADO el código

Lenguaje ensamblador iassim

El desarrollo del módulo fuente en lenguaje ensamblador NO se realiza de principio a fin sino que se va realizando **POR PASOS**, empezando por un código lo más sencillo posible que será testeado y depurado antes de ir desarrollando hasta llegar al código completo

- **1^a Versión:** módulo fuente *sum1toN_v1.ias*:

- La 1^a versión implementa un bucle cuyo cuerpo únicamente almacena un dato en la variable suma. El dato varía en cada iteración.
- Sintaxis → etiqueta: operacion operando ;comentario → 4 Columnas
- Los símbolos para indicar la operación (Ej. S(x)→Ac+) no son mnemónicos
- No utilizar tildes ni en los comentarios ni en las etiquetas, ya que únicamente se admite código ASCII no extendido.
- Si el número de instrucciones es impar se ha de llenar la palabra de 40 bits de la última instrucción con los 20 bits de menor peso a cero.
- La sección de instrucciones debe de ir previamente a la sección de datos

```

; CABECERA
; 1a version : sum1toN_v1.ias
; Calcula la suma de una secuencia de numeros enteros: suma = 1+2+...+n
; dato de entrada : n
; dato de salida : suma
; Algoritmo : bucle de n iteraciones
;           Los sumandos se van generando en sentido descendente de n a 0
;           Se sale del bucle si el sumando es negativo -> -1
; Estructuras de datos : variables n y suma . Constante uno.
; Lenguaje ensamblador: IASSim
; Arquitectura de la máquina IAS de Von Neumann

;;;;;;;;;; SECCION DE INSTRUCCIONES
;Arquitectura orientada a Acumulador (AC)
;Registros accesibles : AC
; algoritmo: bucle que genera la secuencia n, n-1, n-2,...0,-1 si n>=0
bucle: S(x)->Ac+ n      ; AC <- M[n]
      S(x)->Ah- uno   ; AC <- AC-M[uno]
      At->S(x)  suma    ; M[suma] <- AC
      Cc->S(x)  bucle   ; Si AC >= 0, salto a bucle
      ; fin del bucle
      halt          ; stop
      .empty

;;;;;;;;;;SECCION DE DATOS

```

```

; Declaracion de etiquetas, reserva de memoria externa, inicializacion.
; Variables ordinarias
n: .data 5 ; variable sumando
uno: .data 1 ; cte
suma: .data 0 ; sumas parciales y resultado final

```

- Se ha desarrollado la sección de datos para la reserva de memoria.
- Se ha realizado un BUCLE SENCILLO ya que el bucle es la construcción necesaria en el algoritmo final.
- Se ha realizado la operación RESTA ya que es una operación necesaria en el algoritmo final.
- Se ha COMENTADO el código

Registros

- The IAS machine has 7 registers: Accumulator, Arithmetic Register / Multiplier-Quotient (AR/MQ), Control Counter, Control Register, Function Table Register, Memory Address Register, Selectron Register
 - The Accumulator (AC) and Arithmetic registers (AR/MQ) are the only two programmer-visible registers
 - The Control Counter is what we now call the Program Counter *PC*
 - The Control Register holds the currently executing instruction *IBR*. Unicamente la instrucción de la derecha que se va a ejecutar.
 - The Function Table Register holds the current opcode *IR*
 - The Memory Address Register the current memory address *MAR*
 - Selectron Register the current data value being read from or written to memory → *MBR*

Simulador IASSim

- Instrucciones de instalación y de funcionamiento del Simulador IASSim de la máquina IAS de Von Neumann en el [Apéndice](#)

Notas

- Es necesario que el número de instrucciones sea par. Si es impar se añade la directiva *.empty*.
- Una etiqueta debe de apuntar a la instrucción izda. Si está en la dcha se puede anteponer una instrucción de salto incondicional a dicha etiqueta.
- La sección de datos si está a continuación de la sección de código hay que terminar la sección de código con una instrucción en la dcha y si no la rellenamos con la directiva *.empty*.

Error

- Error al visualizar el valor del registro MAR
 - Al ejecutar la primera instrucción de sum1toN.ias el contenido de MAR es 28, mayor que el rango de direcciones de la memoria principal donde esta cargado el programa.

- El error se da tanto en Windows 7 como en Ubuntu 17.04

2.5.3. Ejemplos de Programas en Lenguaje IASSim

- Más ejemplos en el [Apéndice](#).

2.6. Operación de la Máquina IAS: Ruta de Datos

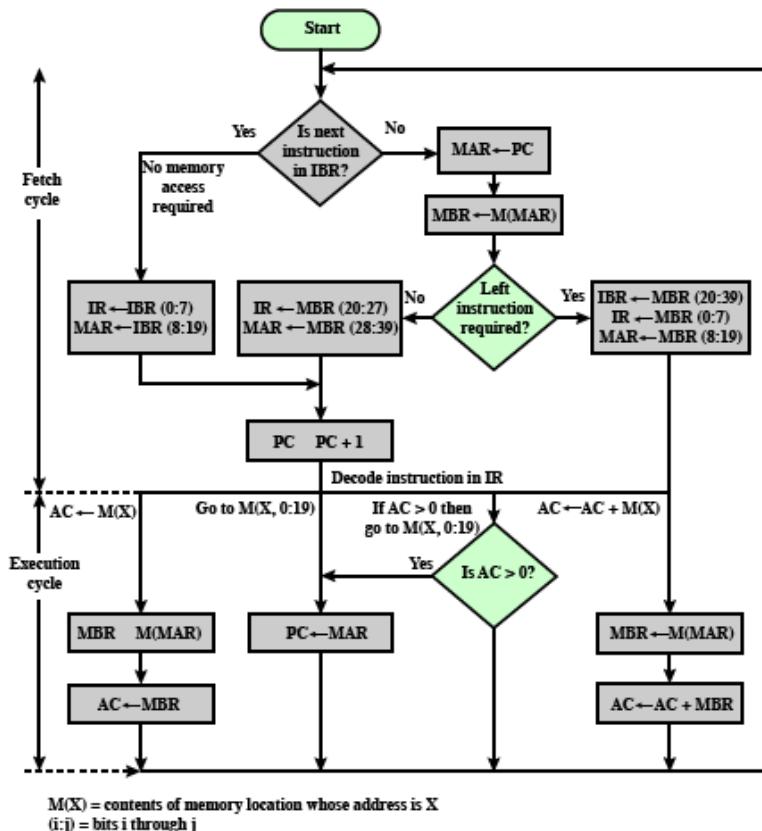


Figure 2.4 Partial Flowchart of IAS Operation

Figure 28. IAS Operation

- Operación de la máquina IAS:
 - El ciclo de instrucción tiene dos FASES
 - La primera fase es común a todas las instrucciones.
- Ejemplos de instrucciones
 - X: referencia del operando
 - $AC \leftarrow M(X)$
 - GOTO $M(X,0:19)$: salto incondicional a la dirección X. X apunta a dos instrucciones. X,0:19 es la referencia de la Instrucción de la izda.
 - If $AC > 0$ goto $M(X,0:19)$: salto condicional
 - $AC \leftarrow AC + M(x)$.

2.7. Conclusiones

1. Para la programación de bajo nivel es necesario conocer las principales características de la arquitectura ISA de la computadora: la Estructura de la computadora (memoria,registros,etc) , Formato de datos (formato complemento a 2,etc) e instrucciones y el Repertorio de Instrucciones (operaciones, modos de direccionamiento, etc ..)
2. La programación en lenguaje ensamblador no se realiza directamente en dicho lenguaje sino que se sigue una estrategia top-down comenzando por una descripción en lenguaje de pseudocódigo, organigrama, lenguaje RTL, etc
3. Es el diseño del repertorio de instrucciones ISA de la computadora el que facilita o dificulta la programación de bajo nivel. Un repertorio excesivamente limitado como la máquina IAS de von Neumann dificulta la realización de expresiones matemáticas tan sencillas como una multiplicación seguida de la división. La secuencia de instrucciones RTL deberá tener en cuenta el facilitar el desarrollo del algoritmo.
4. La programación del algoritmo en lenguaje ensamblador sigue una estrategia ascendente comenzando por una versión incompleta y lo más sencilla posible del programa a desarrollar.
5. Cada versión desarrollada del programa en lenguaje ensamblador ha de ser depurada y verificada mediante un simulador y un depurador que permita la ejecución en modo paso a paso para analizar resultados parciales.

Chapter 3. Representación de los Datos

3.1. Temario

- 3. Representación de datos
 - a. Bit, Byte y Palabra
 - b. Caracteres, enteros y reales

3.2. Objetivo

- Representación de los datos alfanuméricos en el lenguaje máquina, es decir, código binario.
- Libro de texto W.Stalling
 - Parte 3^a, Capítulo 9 : Sistemas Numéricos

3.3. Datos e Instrucciones: Codificación Binaria

- Un programa almacenado en la memoria principal se representa en *lenguaje máquina* y está compuesto por datos e instrucciones . El lenguaje de la máquina es el lenguaje binario formado por los símbolos 0 y 1. Por lo tanto los datos e instrucciones de un programa almacenado en la memoria principal debe de codificarse y representarse mediante estos dos símbolos.
- Los datos tienen un valor numérico que pueden ser procesados por la Unidad Aritmetico Lógica (ALU) para realizar operaciones aritméticas como la suma, resta, etc ó lógicas como las operaciones not,or,etc.
- Los datos son secuencias de 0 y 1 almacenados en la memoria que son capturados por la CPU para ser procesados pej mediante operaciones aritméticas como la suma.
- Las instrucciones son secuencias de 0 y 1 almacenados en la memoria que son capturados por la CPU para ser interpretados y proceder a su ejecución. Pej la instrucción "sumar" dos números enteros.

3.4. Bit, Byte, Palabra

- BIrary digiT (bit) : los dígitos binarios son el 0 y el 1. Son los símbolos que se utilizan para codificar tanto las instrucciones como los datos de un programa almacenado en memoria.
- Byte: Es una secuencia de 8 dígitos binarios. Ejemplo: 00110101
- Palabra: Es una secuencia de dígitos binarios múltiplo de 8, es decir, múltiplo de un byte. En el entorno de arquitectura de computadores se define como el número de bits del bus de datos que conecta la unidad central de proceso (CPU) a la Memoria principal y también suele ser la anchura de los registros de propósito general de memoria interna de la CPU.
 - En linux +sudo lshw -C system | more + → anchura: **64 bits**

lur

```

descripción: Notebook
producto: 20F1S0H400 (LENOVO_MT_20F1_BU_Think_FM_ThinkPad L560)
fabricante: LENOVO
versión: ThinkPad L560
serie: MP15YSW7
anchura: 64 bits
capacidades: smbios-2.8 dmi-2.8 smp vsyscall32
configuración: administrator_password=disabled chassis=notebook family
=ThinkPad L560 power-on_password=disabled sku
=LENOVO_MT_20F1_BU_Think_FM_ThinkPad L560 uuid=4C2F45AA-0A2C-B211-A85C-
B5C56EB5BBAC

```

3.5. Números Enteros

3.5.1. Base Decimal

- Son representados mediante un Sistema Posicional basado en:
 - Número en Base Decimal
 - Representación mediante la combinación de diez Dígitos: 0,1,2,3,...,9
 - Posición → índice
 - Pesos de cada posición → son potencias de la forma $10^{posición}$ → ...,Centenas, decenas, unidades
 - Valor representado = sumatorio de digitos ponderados con su peso posicional
 - Ejemplo: Dada la representación 1197 de un número decimal entero, calcular su valor.

Posición	3	2	1	0
Peso	10^3	10^2	10^1	10^0
	1000	100	10	1
Dígito	1	1	9	7
Ponderación	$1*1000$	$1*100$	$9*10$	$7*1$

Valor $1*1000+1*100+9*10+7*1=\text{mil ciento noventa y siete}.$

La representación uno-uno-nueve-siete tiene el valor mil ciento noventa y siete

3.5.2. Base Binaria

- Número codificado en Base 2
 - Representación mediante la combinación de dos Dígitos: 0,1
 - Posición → índice
 - Pesos de cada posición → son potencias de la forma $2^{posición}$ → ... $2^5,2^4,2^3,2^2,2^1,2^0$...,64,32,16,8,4,2,1

- Valor representado = sumatorio de dígitos ponderados con su peso posicional
- Ejemplo: Dada la representación 1010 de un número binario entero, calcular su valor.

Posición	3	2	1	0
Peso	2^3	2^2	2^1	2^0
	8	4	2	1
Dígitos	1	0	1	0
Ponderación	$1*8$	$0*4$	$1*2$	$0*1$

$$\text{Valor} \quad 1*8+0*4+1*2+0*1 = \text{diez}$$

La representación uno-cero-uno-cero tiene el valor diez

3.5.3. Conversión Decimal-Binaria

- Divisiones sucesivas / 2 → $\text{Dividendo}_1 = 2 * \text{Cociente}_1 + \text{Resto}_1$
- $\text{Cociente}_1 = 2 * \text{Cociente}_2 + \text{Resto}_2 \rightarrow \text{Dividendo}_1 = 2 * (2 * \text{Cociente}_2 + \text{Resto}_2) + \text{Resto}_1 = \text{Resto}_1 * 2^0 + \text{Resto}_2 * 2^1 + \text{Cociente}_2 * 2^2$
- Resto1 es el dígito binario de la posición 0, Resto2 es el dígito binario de la posición 1, Cociente es el dígito binario de la posición 2.
- Regla: los dígitos binarios son todos los restos y el último cociente.
- La división se termina cuando un cociente no es divisible por 2, es decir, el cociente es 1. Este cociente es el MSB.
- Ejemplo: Valor 1197 → Calcular su representación en código binario → Solución: 10010101101

Table 3. Conversión decimal binario

Número	1 ^a Div		2 ^a Div		3 ^a Div		4 ^a Div		5 ^a Div		6 ^a Div	
	Coc	Resto										
1197	598	1	299	0	149	1	74	1	37	0	18	1

Número	7 ^a Div		8 ^a Div		9 ^a Div		10 ^a Div	
	Coc	Resto	Coc	Resto	Coc	Resto	Coc	Resto
1197	9	0	4	1	2	0	1	0

3.5.4. Base Octal

- Base 8
- Dígitos: 0,1,2,3,4,5,6,7
- Pesos: 8 elevado a la posición
- En C se especifica la base con el prefijo 0 → `int 077;`
- Conversión Octal ↔ Binario y viceversa → cada dígito octal se descompone en un binario de 3

bits

- decimal 1197 → Calcular su representación en código octal.
 - solución a) binario 10010101101 → octal 02255
 - solución b) divisiones sucesivas por la base 8.

Base Hexadecimal

- Base 16
- Dígitos: 0-1-2-3-4-5-6-7-8-9-A-B-C-D-E-F
- Pesos: 16 elevado a a la posición
- En C se especifica la base con el prefijo `0x` → `int 0xAF;`
- Hexadecimal ↔ Binario y viceversa → cada dígito hexadecimal se descompone en un binario de 4 bits
- decimal 1197 → Calcular su representación en código hexadecimal
- Solución a) binario 10010101101 → 0x4AD
- Solución b) divisiones sucesivas por la base 16.

3.5.5. Calculadora

- Calculadora en el sistema Linux
 - `candido@lur:~$ echo "obase=2 ; ibase=16; 80AA010F" | bc`
 - 10000000101010100000000100001111
 - `echo "obase=10 ; ibase=16; 80AA010F" | bc` → es obligado poner primero la base del formato de salida
 - 2.158.625.039
 - Intérprete `$ bc`

3.5.6. Python

- <https://docs.python.org/3/tutorial/index.html>
 - `help(builtins)`

```
bin(1197) -> '0b10010101101'  
oct(1197) -> '02255'  
hex(1197) -> '0x4ad'  
int(0x4ad) -> 1197
```

3.5.7. Enteros con Signo

- Se van a estudiar dos formatos: Signo-Magnitud y Complemento a 2, siendo este último el más extendido en las arquitecturas de los computadores.

Signo-Magnitud

- Formato Signo-Magnitud
 - El bit más significativo no tiene valor, indica el signo: el cero para los números positivos y el uno para los negativos.
 - El resto de bits representa el módulo del número entero
 - Ejemplo :
 - Valor +1197 → Representación 010010101101
 - Valor -1197 → Representación 110010101101
 - ¿Cómo se representa el valor cero?

Complemento a 2

- Formato Complemento a 2.
 - Positivos: Igual que el formato signo-magnitud: Bit MSB= 0. Pesos: potencia $2^{\text{posición}}$.
 - More Significant Bit (MSB) → posición más elevada con valor distinto de cero.
 - Less Significant Bit (LSB)
 - Negativos: Transformación del número con la misma magnitud pero positivo mediante la función Complemento a 2.
- La Rueda

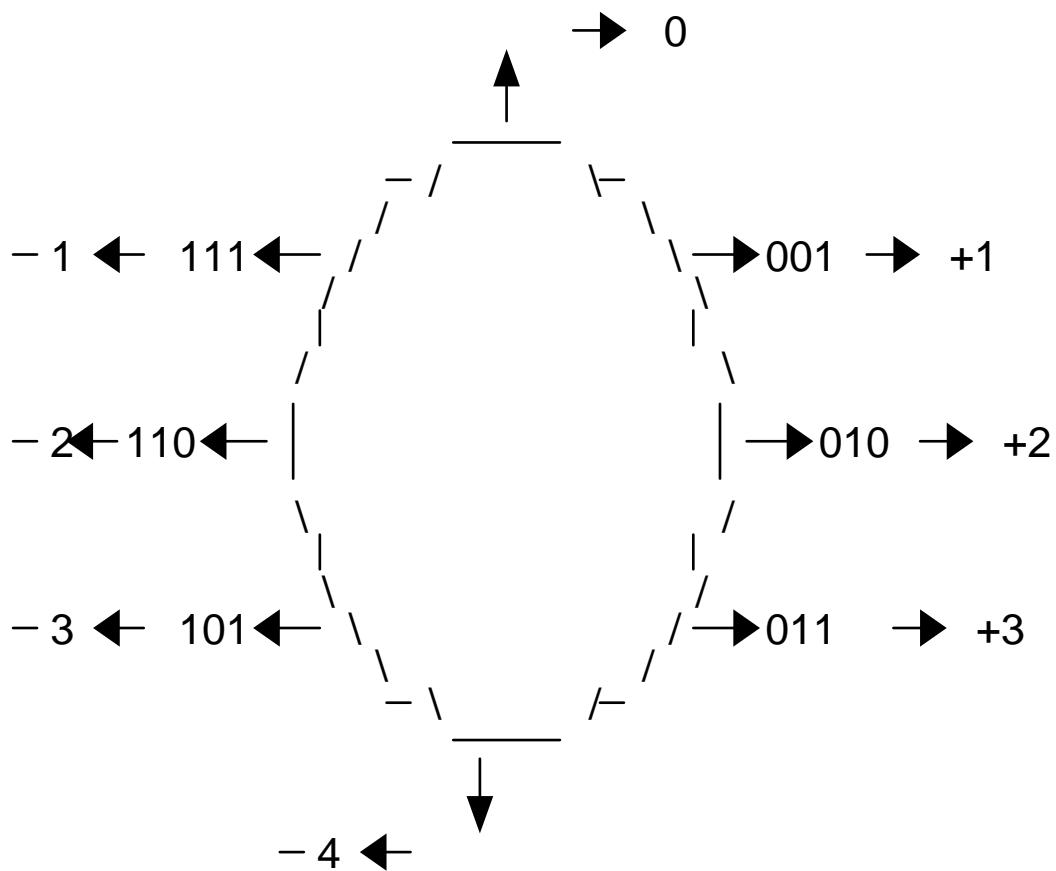


Figure 29. Complemento a 2. Longitud 3 bits.

- N: cantidad de bits del número : 3 bits
- Dividir la circunferencia en el número de combinaciones binarias posibles: $2^N : 2^3$
- Pinto todas las combinaciones binarias en sentido agujas reloj de forma secuencial:
000,001,010,011,
- Pinto los valores de forma alternante: 0, +1, -1, +2, -2,....
- Ejercicio: Representar el número positivo +4 en complemento a 2
- Conclusiones:
 - Asimetría entre el rango positivo y negativo
 - El cero tiene una única representación
 - Los números negativos comienzan por 1
 - El valor -1 se codifica con todos los dígitos unos 11111111111111
 - Extensión de Signo: un 1 por la izda es como en los positivos un cero por la izda: no tiene valor y se puede eliminar la repetición de 1 por la izda dejando el último 1 de los más significativos. 11110111 es equivalente a 10111.
- **Función Complemento a 2:** El complemento a 2 de un número entero equivale a cambiar su signo. La conversión entre números enteros positivos y negativos en complemento a 2 se puede realizar mediante distintos métodos.

- Ejemplos de obtención del complemento a 2 del número entero binario X:
 - a. Método 1: Realizar la operación lógica complemento (negación) de X y sumar 1 → $\sim X + 1$
 - X=0101 tiene valor +5 en complemento a 2
 - ¿El complemento a 2 de 0101? $\sim 0101 + 1 = 1010 + 1 = 1011 = -5 \rightarrow$ El valor del complemento a 2 equivale a cambiar de signo.
 - X=1111 tiene valor -1 en complemento a 2
 - ¿El complemento a 2 de 1111? $\sim 1111 + 1 = 0000 + 1 = 0001 = +1$
 - X=0110011100010101010000 → positivo por tener el bit más significativo (MSB) cero
 - $C_2(X) = 1001100011101010101111 + 1 = 1001100011101010110000 \rightarrow$ negativo por tener el bit más significativo (MSB) uno
 - b. Método 2: Empezando por la posición 0 del código X (bit X_0) copiar todos los dígitos hasta llegar al primer dígito 1 y a partir de ahí negar todos los dígitos hasta el bit más significativo (MSB).
 - X = 0110011100010101010000 → en total 22 bits
 - El primer dígito 1 de X está en la posición 4 → 01100111000101010-10000 → copio los 5 primeros dígitos e invierto los 17 restantes
 - $C_2(X) = 10011000111010101-10000$
 - c. Método 3: Realizar la operación aritmética 0-X
 - X = 0110011100010101010000
 - $0-X = 00000000000000000000000000 - 0110011100010101010000 = 0110011100010101010000$
- Ejemplos
 - Representar el número entero negativo -1197 en signo-magnitud y en complemento a 2
 - $+1197 = 010010101101$ tanto en signo-magnitud como complemento a 2
 - $-1197 = 101101010011$
 - Calcular el rango de los números enteros con 8 bits en complemento a 2
 - Código máximo positivo: 01111111 → Valor = $2^7 - 1$
 - Código mínimo negativo: 10000000
 - $C_2(n=10000000) = 01000000 = 2^7$, luego n=10000000 tiene el valor -2^7
 - Rango $[-2^7, +2^7 - 1]$

3.6. Números Reales

3.6.1. Coma Fija

- Números Reales en Coma Fija:
 - 1234.56789
 - Sistema Posicional

- posición de los dígitos fracción: -1,-2,-3,...
- pesos de los dígitos fracción: 10^{-1} , 10^{-2} , 10^{-3}
- ponderación $1234.56789 = 1*10^3 + 2*10^2 + 3*10^1 + 4*10^0 + 5*10^{-1} + 6*10^{-2} + 7*10^{-3} + 8*10^{-4} + 9*10^{-5}$
- Base Binaria
 - $1010.101 \rightarrow 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3} \rightarrow 10.625$

3.6.2. Coma Flotante

Formato

- Coma Flotante → Notación científica
 - -23.4567E-34 ó $-23.4567*10^{-34}$
 - La **mantisa o significando** es el número que multiplica a la potencia → -23.4567
 - Mantisa **normalizada** : La mantisa tiene como parte entero un número entero de un dígito distinto de cero. → $-2.34567*10^{-33}$
 - parte entera de la mantisa normalizada : 2
 - parte fracción de la mantisa normalizada : 0.34567
 - El **exponente** es el número entero al que se eleva la base de la potencia. Depende del lugar de la coma en la mantisa. En este caso es -33.
 - La **base** es la base de la potencia. En este ejemplo es 10.
- Codificación Binaria
 - Ejemplo: 1234.56789
 - Parte Entera: 1234 → 10011010010
 - Parte Fracción: 0.56789

```

0.56789 * 2 = 1.13578 = 1 + 0.13578 -> 1, bit de la posición -1
0.13578 * 2 = 0.27156 -> 0, bit de la posición -2
0.27156 * 2 = 0.54312 -> 0, bit de la posición -3
0.54312 * 2 = 1.08624 = 1 + 0.08624 -> 1, bit de la posición -4
  
```

- fracción redondeada 0.1001
- fracción sin redondear 0.10010001011000010
- fracción redondeada 0.100100011
- Código Binario coma fija: 10011010010.10010001011000010
- Notación científica: $1.001101001010010001011000010*2^{+10}$ → coma flotante → la parte entera siempre vale 1.

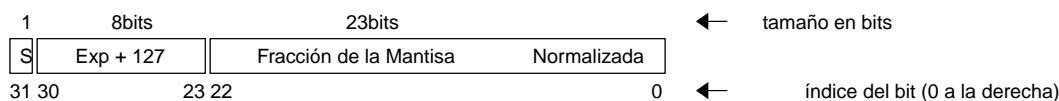
Precisión

- Es el número de dígitos significantes

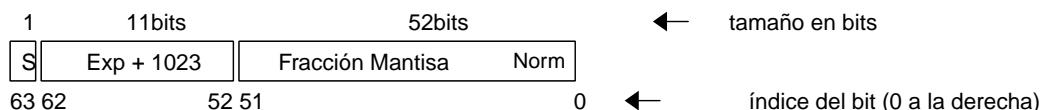
- Se dice que el número q es una aproximación del número p != 0 con una precisión de, al menos, m cifras significativas en la base b, siempre que el error relativo $|p-q|/p \leq 0.5 \cdot b^{-m+1}$
 - Cuando m es el mayor entero para el que se cumple la desigualdad anterior, se dice que q aproxima a p con m cifras significativas.
- Ejemplo
 - $p = 1E0$ y $q = .9999E0 \rightarrow$ Error relativo = $0.1E-3 < 0.5E(-4+1) \rightarrow$ precisión de 4 cifras significativas
 - Una calculadora, A, trabaja en base 2 con mantisa de 22 bits y otra, B, trabaja en base 16 con 6 dígitos de precisión (24 bits). ¿Cuál de las dos es más precisa?

Norma IEEE-Standard 754

- Float
 - Norma IEEE-Standard 754
 - Precisión simple → formato de longitud 32 bits en 3 campos *Signo/Exponente/Fracción de la Mantisa* de longitudes 1/8/23 bits



- Precisión doble → formato de longitud 64 bits en 3 campos *Signo/Exponente/Fracción de la Mantisa* de longitudes 1/11/52 bits



- [conversion manual](http://sandbox.mc.edu/~bennet/cs110/flt/dtob.html) [<http://sandbox.mc.edu/~bennet/cs110/flt/dtob.html>]
- [wiki](http://en.wikipedia.org/wiki/Floating_point) [http://en.wikipedia.org/wiki/Floating_point]
- Binario: Tres campos

Signo	Exponente en Exceso	Fracción de la Mantisa Normalizada
-------	---------------------	------------------------------------

- Valor $(-1)^{\text{Signo}} \times 1.\text{Fracción_Mantisa_Normalizada} \times 2^{\text{Exponente}}$
 - Signo: positivo → bit 0, negativo → bit 1
 - Exponente en exceso: Es el Exponente al que se añade 127 (precisión simple) ó 1023 (precisión doble)
 - Mantisa Normalizada: Es la mantisa tal que su parte entera es 1
 - Fracción de la Mantisa Normalizada: Es la fracción de la mantisa normalizada.

Conversores de Código

- Conversores online:
 - [binary converter](http://www.binaryconvert.com/index.html) [<http://www.binaryconvert.com/index.html>]: tipos char,short,int,float,double
 - [conversor ieee754](http://www.zator.com/Cpp/E2_2_4a1.htm) [http://www.zator.com/Cpp/E2_2_4a1.htm]
 - [IEEE 754 single precision](http://www.h-schmidt.net/FloatConverter/IEEE754.html) [<http://www.h-schmidt.net/FloatConverter/IEEE754.html>]: decimal → binario/hexadecimal y viceversa

Float Point: Representación del Cero, Infinito e Indeterminado

- Cuando el campo del exponente son todo ceros o unos, no se sigue la regla general de un número normalizado

Table 4. Single precision

Números	Exp	Fracción
Ceros	0x00	0
Números desnormalizados	0x00	distinto de 0
Números normalizados	0x01-0xFE	cualquiera
Infinitos	0xFF	0
NaN (Not a Number)	0xFF	distinto de 0

- Cero
 - Por qué el cero se representa en single precision como una secuencia de 32 ceros
 - Por qué cuando el campo del exponente es cero la potencia es 2^{-126} en lugar de 2^{-127} y la mantisa se considera NO normalizada, es decir, 0.fracción en lugar de 1.fracción.
- [Notas Maryland](http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/float.html) [<http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/float.html>]
- Infinito

Referencia

- [numerical analysis](http://people.ds.cam.ac.uk/nmm1/arithmetic/na1.pdf) [<http://people.ds.cam.ac.uk/nmm1/arithmetic/na1.pdf>]: programas ejemplo sencillos
- [IEEE](http://grouper.ieee.org/groups/754/) [<http://grouper.ieee.org/groups/754/>]
- [William Kahan](http://www.cs.berkeley.edu/~wkahan/) [<http://www.cs.berkeley.edu/~wkahan/>]
- [Yale](http://www.cs.yale.edu/homes/aspnes/pinewiki/C%282f%29FloatingPoint.html) [<http://www.cs.yale.edu/homes/aspnes/pinewiki/C%282f%29FloatingPoint.html>]: C programming float.c.
- [Bruce Dawson blog](https://randomascii.wordpress.com/category/floating-point/) [<https://randomascii.wordpress.com/category/floating-point/>]
 - <https://randomascii.wordpress.com/2012/01/11/tricks-with-the-floating-point-format/>
- [wikipedia](https://en.wikipedia.org/wiki/IEEE_floating_point) [https://en.wikipedia.org/wiki/IEEE_floating_point]
 - <http://www.validlab.com/goldberg/paper.pdf>
 - [The pitfalls of verifying floating-point computations](https://hal.archives-ouvertes.fr/hal-00128124v5/document) [<https://hal.archives-ouvertes.fr/hal-00128124v5/document>]

- [el mismo?](http://arxiv.org/pdf/cs/0701192.pdf) [http://arxiv.org/pdf/cs/0701192.pdf]
- [cprogramming](http://www.cprogramming.com/tutorial/floating_point/understanding_floating_point_representation.html) [http://www.cprogramming.com/tutorial/floating_point/understanding_floating_point_representation.html]
- [code tips](http://www.codeproject.com/Articles/29637/Five-Tips-for-Floating-Point-Programming) [http://www.codeproject.com/Articles/29637/Five-Tips-for-Floating-Point-Programming]
- [c review](https://www.cs.princeton.edu/courses/archive/fall09/cos323/precepts/precept2.html) [https://www.cs.princeton.edu/courses/archive/fall09/cos323/precepts/precept2.html]: practicas

3.7. Character Type

3.7.1. ASCII

- Codificación ASCII
 - American Standard Code International Intechange: codificación con 7 bits : rango 0x00-0x7F
 - Tabla de conversión carácter-código_hexadecimal-código binario
 - [man ascii](#)
 - K.N. King,Apéndice E, pg801

Carácter	ASCII hex	Control (Secuencia de Escape)
0	0x30	
1	0x31	
a	0x61	
A	0x41	
+	0x2B	
^J	0x0A	nueva línea (\n)
^M	0x0D	retorno de carro (\r)

- fijarse la relación del código entre J y ^J, entre M y ^M...

C program '\X' escapes are noted.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
-----	-----	-----	------	-----	-----	-----	------

000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH (start of heading)	101	65	41	A
002	2	02	STX (start of text)	102	66	42	B
003	3	03	ETX (end of text)	103	67	43	C
004	4	04	EOT (end of transmission)	104	68	44	D

005	5	05	ENQ (enquiry)	105	69	45	E
006	6	06	ACK (acknowledge)	106	70	46	F
007	7	07	BEL '\a' (bell)	107	71	47	G
010	8	08	BS '\b' (backspace)	110	72	48	H
011	9	09	HT '\t' (horizontal tab)	111	73	49	I
012	10	0A	LF '\n' (new line)	112	74	4A	J
013	11	0B	VT '\v' (vertical tab)	113	75	4B	K
014	12	0C	FF '\f' (form feed)	114	76	4C	L
015	13	0D	CR '\r' (carriage ret)	115	77	4D	M
016	14	0E	SO (shift out)	116	78	4E	N
017	15	0F	SI (shift in)	117	79	4F	O
020	16	10	DLE (data link escape)	120	80	50	P
021	17	11	DC1 (device control 1)	121	81	51	Q
022	18	12	DC2 (device control 2)	122	82	52	R
023	19	13	DC3 (device control 3)	123	83	53	S
024	20	14	DC4 (device control 4)	124	84	54	T
025	21	15	NAK (negative ack.)	125	85	55	U
026	22	16	SYN (synchronous idle)	126	86	56	V
027	23	17	ETB (end of trans. blk)	127	87	57	W
030	24	18	CAN (cancel)	130	88	58	X
031	25	19	EM (end of medium)	131	89	59	Y
032	26	1A	SUB (substitute)	132	90	5A	Z
033	27	1B	ESC (escape)	133	91	5B	[
034	28	1C	FS (file separator)	134	92	5C	\ '\\'
035	29	1D	GS (group separator)	135	93	5D]
036	30	1E	RS (record separator)	136	94	5E	^
037	31	1F	US (unit separator)	137	95	5F	_
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	\$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(150	104	68	h
051	41	29)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w

070	56	38	8		170	120	78	x
071	57	39	9		171	121	79	y
072	58	3A	:		172	122	7A	z
073	59	3B	;		173	123	7B	{
074	60	3C	<		174	124	7C	
075	61	3D	=		175	125	7D	}
076	62	3E	>		176	126	7E	~
077	63	3F	?		177	127	7F	DEL

- ASCII Extendido

- https://en.wikipedia.org/wiki/Extended_ASCII#ISO_8859_and_proprietary_adaptations
- `man iso_8859_1`: latin-1: ascii extendido: 0x80-0xFF
- <http://www.theasciicode.com.ar/ascii-printable-characters/vertical-bar-vbar-vertical-line-vertical-slash-ascii-code-124.html>
 - El linux pulsar ctrl-Shift-u-ascii_code Enter
 - Ejemplo: el código extendido de la ñ es 0xF1 → C-S-u-f1 Enter → C-S-u simultáneo y aparece la u esperando al código, F-1-enter
- [ascii code finder](http://www.mauvecloud.net/charsets/CharCodeFinder.html) [http://www.mauvecloud.net/charsets/CharCodeFinder.html]
- 0
- ~
- ¬
- ñ

3.7.2. Python

- ejemplos de conversión
 - `python`

```
ord('A')
hex(ord('A'))
chr(65)
chr(0x41)
[hex(ord(c)) for c in "Hola"]
[chr(c) for c in [0x48, 0x6f, 0x6c, 0x61, 0x20, 0x4d, 0x75, 0x6e, 0x64, 0x6f]]
```

3.7.3. Unicode UTF-8

- [Unicode Main](https://www.unicode.org/main.html) [https://www.unicode.org/main.html]
- Unicode: Unicode can be implemented by different character encodings. The Unicode standard defines Unicode Transformation Formats (UTF): UTF-8, UTF-16, and UTF-32, and several other encodings. The most commonly used encodings are UTF-8, UTF-16, and the obsolete UCS-2 (a precursor of UTF-16 without full support for Unicode)

- Unicode encoded: <https://www.unicode.org/versions/Unicode14.0.0/ch02.pdf#G25564>
 - Se describe con el prefijo U+ seguido de un número entero (integers from 0 to 0x10FFFF). Al código se le llama **code point**"
- UTF-8:
 - The dominant encoding on the World Wide Web and on most Unix-like operating systems
 - Uses one byte[note 1] (8 bits) for the first 128 code points, and up to 4 bytes for other characters. The first 128 Unicode code points represent the ASCII characters, which means that any ASCII text is also a UTF-8 text.
 - La ñ da como salida 0xc3b1 . El terminal está configurado con salida Unicode UTF8 según la variable de entorno local. Mediante el comando **locale charmap** volvemos con que codificación tenemos la entrada/salida del terminal. Mediante **locale -m** los posibles. Podría haber sido iso-8859-1 en lugar de utf8.
- **localectl --status** → codificación de entrada del teclado

```
System Locale: LANG=eu_ES.UTF-8
                LANGUAGE=eu_ES:eu:en_Gb:en
VC Keymap: n/a
X11 Layout: es
X11 Model: pc105
```

- **utf8** [<http://www.utf8-chartable.de/unicode-utf8-table.pl?number=1024>]:
 - 8-bit Unicode Transformation Format
 - Usa símbolos de longitud variable (de 1 a 4 bytes por carácter Unicode).
 - Esta orientado a la transmisión de palabras de 1 byte.
 - **unicode ñ** [<http://www.fileformat.info/info/unicode/char/f1/index.htm>]
 - la ñ tiene unicode point *U+00F1* ó hex_code_utf8 *0xC3B1*
 - en la wikipedia utf-8 explica cómo pasare de unicode point a hex code.
 - <https://unicode-table.com/es/00F1/>
 - Problema Para copiar los caracteres no US-ASCII de la barra URL de firefox: https://es.wikipedia.org/wiki/Commutaci%C3%B3n_de_circuitos. → C3B3 es el código hexadecimal del código utf-8 del carácter ó.
 - wikipedia utf-8:
 - desglose de códigos según 1byte,2byte,3 byte, 4 bytes.
 - cómo se mapea el unicode code point del utf-8 a hexadecimal
 - **man utf-8**
- **showkey -a** : espera a pulsar una letra y visualizará el código de la letra pulsada en la codificación de entrada del sistema operativo.
 - El código de los caracteres del ASCII standard (7bits) coincide con el UTF8 pero no así para el resto de caracteres ASCII extendido.

- útil para descubrir el código de cada carácter en ascii standard y el de caracteres ñ, á, é, í, ó, ú si en el código en que el sistema esté configurado (UTF8)
- útil para descubrir el código de control de combinaciones Ctrl-C, CR, Ctrl-CR, Ctrl-D

```

\  92 0134 0x5c  -> tecla ESC: escape
^] 10 0012 0x0a  -> teclas Ctrl-CR: Salto de línea
^M 13 0015 0x0d  -> tecla CR: Retorno de Carro
^C 3 0003 0x03  -> teclas Ctrl-c
^D 4 0004 0x04  -> teclas Ctrl-d
ñ 195 0303 0xc3  -> MSB: More Significant Byte.
    177 0261 0xb1  -> LSB: Less Significant Byte

```

- UPNA → 0x55-0x50-0x4e-0x41-0x00 donde 0x00 es el carácter NUL de fin de cadena.

- Documentos HTML

- ñ → ñ → utiliza el código "unicode point"

- Sistema operativo : variables del entorno

- `env | grep LC_`

- [Unicode chart](http://www.unicode.org/charts/) [http://www.unicode.org/charts/]

- Colocando el puntero sobre la categoría se visualiza el rango hexadecimal del charset

- Symbols Punctuation:

- Punctuation: ASCII Punctuation: [U0000.pdf](http://www.unicode.org/charts/PDF/U0000.pdf) [http://www.unicode.org/charts/PDF/U0000.pdf]
- Find chart by hex code: 278a
- Pictographs: Dingbats: ☐ → [U2700.pdf](http://www.unicode.org/charts/PDF/U2700.pdf) [http://www.unicode.org/charts/PDF/U2700.pdf]
- Mathematical symbols: Mathematical Operators:

- [wikipedia](https://en.wikipedia.org/wiki/Mathematical_operators_and_symbols_in_Unicode) [https://en.wikipedia.org/wiki/Mathematical_operators_and_symbols_in_Unicode]
- [U2200](http://www.unicode.org/charts/PDF/U2200.pdf) [http://www.unicode.org/charts/PDF/U2200.pdf]
- hexadecimal x2228 → ☐ ó en decimal 8744 ☐
- ☐
- ☐
- ☐

- otros

- ☐
- ñ
- ñ
- ←
- →

- https://wiki.mozilla.org/Help:Special_characters#Unicode

- [info detallada sobre un carácter unicode](http://www.fileformat.info/info/unicode/char/305/index.htm) [http://www.fileformat.info/info/unicode/char/305/index.htm]: Pej U+0305
- [Microsoft Office](https://support.office.com/en-us/article/Insert-ASCII-or-Unicode-Latin-based-symbols-and-characters-d13f58d3-7bcb-44a7-a4d5-972ee12e50e0) [https://support.office.com/en-us/article/Insert-ASCII-or-Unicode-Latin-based-symbols-and-characters-d13f58d3-7bcb-44a7-a4d5-972ee12e50e0]
- [Overline o suprarayado](https://en.wikipedia.org/wiki/Overline) [https://en.wikipedia.org/wiki/Overline]
 - LibreOffice has direct support for several styles of overline in its **Format / Character / Font Effects** dialog: **suprarayado**

3.8. ISO-8859-1

- Alternativa a UTF-8 para el alfabeto latino
- https://es.wikipedia.org/wiki/ISO/IEC_8859-1
 - Sólo utiliza 1 byte , por lo tanto es equivalente al ascii extended.
 - La norma ISO/IEC 8859-15 consistió en una revisión de la ISO 8859-1, incorporando el símbolo del Euro
- [man iso_8859-1](#)
- La "ñ" tiene el código 0xF1

3.8.1. Programación en C

- Convertir un carácter numérico en su valor entero
 - Mediante una operación aritmética
- Convertir un carácter minúscula en mayúscula
 - Mediante una operación aritmética

3.8.2. Otros

- [Lenguaje C: printf](https://www.gnu.org/software/coreutils/manual/html_node/printf-invocation.html) [https://www.gnu.org/software/coreutils/manual/html_node/printf-invocation.html]
 - `locale -a` → C.UTF8
 - ñ → `env printf \u00f1 \n`: incluir las simples comillas
 - [printf invocation](http://www.gnu.org/software/coreutils/manual/html_node/printf-invocation.html#printf-invocation) [http://www.gnu.org/software/coreutils/manual/html_node/printf-invocation.html#printf-invocation]

Chapter 4. Operaciones Aritméticas y Lógicas

4.1. Temario

1. Aritmética y lógica
 - a. Operaciones aritméticas y lógicas sobre enteros en binario
 - b. Redondeo y propagación de error en números reales

4.2. Objetivo

- Operaciones aritméticas de suma y resta con números naturales y enteros representados en código binario.
- Operaciones lógicas de datos representados en código binario.
- Libro de texto
 - Parte 3^a, Capítulo 10 : Aritmética del Computador

4.3. Introducción

- La Unidad Aritmetico Lógica (ALU) es la unidad hardware básica encargada de realizar las operaciones de cálculo aritmético como la suma y resta y de realizar operaciones lógicas de tipo booleano como las operaciones NOT, OR, AND, etc

4.4. Aritmetica Binaria

4.4.1. Suma en módulo 2 (binaria) en binario puro (Nº NATURALES)

- Los números naturales no tienen la marca de un signo ya que son todos positivos: 0,1,2,3....
- Suma de datos en código binario puro
 - Concepto de operación **modular**
 - Ejemplo: módulo 100.000 → Interpretación modular gráfica mediante la circunferencia. Qué ocurre en el cuenta-kilómetros parcial del coche cuando llegamos a 99.999.
 - Suma binaria en módulo 2. El **acarreo** (llevada) se produce al llegar o pasar el valor 2.
 - $1+1=$ uno más uno = dos $\geq 2 \rightarrow$ al valor dos le resto el módulo 2 ($2-2=0$) y me llevo una. El valor 2 en binario se representa como 1 0, donde el cero es la representación en la misma posición que el digito sumando y el 1 la llevada a la siguiente posición.
 - $1+1+1=$ uno más uno más uno = tres $\geq 2 \rightarrow$ al valor tres le resto el módulo 2 ($3-2=1$) y me llevo una. El valor 3 en binario se representa como 1 1, donde el uno de la derecha es la representación en la misma posición que el digito sumando y el 1 de la izda es la llevada a la siguiente posición.

- Ejercicio: calcular la suma de $10011011 + 00011011 = 10110110$

Llevadas -->	1 1 1 1
	1 0 0 1 1 0 1 1 <--sumando
	+ 0 0 0 1 1 0 1 1 <--sumando
Valor suma	1 3 2 1 3 2

Resultado -->	1 0 1 1 0 1 1 0 <--suma

Overflow ó Desbordamiento

- Se dice que la suma o resta se ha desbordado cuando:
 - El valor del resultado a representar está fuera del rango debido a la limitación del número de dígitos.
 - El resultado de la operación aritmética tiene un tamaño superior al permitido por la palabra de memoria o registro donde se almacena.
 - La solución sería aumentar el número de dígitos que representan al dato, pero no siempre se puede.
 - Lógicamente si se da un desbordamiento el resultado que proporciona la ALU no es correcto. La ALU dispone de un flag o banderín de desbordamiento OF (overflow flag) que almacena un bit. Si el bit OF=1 significa que la última operación realizada por la ALU ha producido overflow. El programador puede saber si ha habido error de overflow leyendo el banderín OF.
- Ejemplos:
 - La unidad ALU dispone de dos registros de entrada de "1 byte", AL y BL, cada uno donde almacena dos datos : 10011011 y 10011011. Dispone también de un registro de salida de 1 byte, CL. Calcular el resultado de la suma en formato binario puro: $CL \leftarrow AL+BL$ y el valor del banderín OF.

Llevadas -->	1 1 1 1
	1 0 0 1 1 0 1 1 <--AL
	+ 1 0 0 1 1 0 1 1 <--BL
Valor suma	2 1 3 2 1 3 2

Resultado -->	1 0 0 1 1 0 1 1 0 <--suma
CL :	0 0 1 1 0 1 1 0
OF :	1

- Error de overflow ya que la ALU ha calculado el resultado de la suma: 00110110

- El resultado correcto 100110110 está fuera del rango de un registro de 8 bits. El rango permitido serían los números comprendidos entre 00000000 y 11111111, es decir, valores comprendidos entre 0 y 255. El dato 100110110 cuyo valor es 310. La solución sería diseñar una nueva CPU con registros cuyo tamaño de palabra sea mayor que 1 byte, pej 2 bytes. Entonces si a la entrada de la ALU tenemos AX=0000000010011011 BX=0000000010011011 , el resultado de la operación CX \leftarrow AX+BX sería 0000000100110110 y OF=0 → no hay error de overflow.

4.4.2. Resta en módulo 2 (binaria) en binario puro

- Para poder restar dos números naturales (sin signo) es necesario que el valor del minuendo sea superior al del sustraendo.
 - 0-0 = 0
 - 1-1 = 0
 - 1-0 = 0
 - ¿Qué ocurre si a 0 le tengo que restar 1? Al valor 0 NO se le puede restar el valor 1. Cuando un dígito del minuendo en la posición "p" es menor que el dígito en la misma posición "p" del sustraendo, la solución es sumarle al minuendo de la posición p el módulo (2 en binario) y al mismo tiempo también sumarle el mismo valor al sustraendo pero a través de la posición "p+1", con lo cual si sumamos el mismo valor tanto al minuendo como al sustraendo el resultado de la resta no se ve afectado.
 - posición "p": minuendo 0 - sustraendo1 → En el minuendo $0 + \text{módulo}-1 = 0 + 2 - 1 = 1$. El valor 2 en la posición "p" equivale al valor 1 en la posición "p+1". En la posición "p+1" sumaremos 1 al sustraendo.
 - posición "p": minuendo 0 - sustraendo 1 - llevada 1 → En el minuendo $0 + \text{módulo}-1-1 = 0 + 2 - 1 - 1 = 0$ y llevada 1 que sumaremos a la posición siguiente del sustraendo.
 - posición "p": 1-1-1 → en el sustraendo $1 + \text{módulo}-1-1 = 1 + 2 - 1 - 1 = 1$ y llevada 1 que sumaremos a la posición siguiente del sustraendo.
 - $10110110 - 10011011 = 00011011$

Sumar crédito al minuendo

2 2 2 2

$$\begin{array}{r} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ - 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} \quad \begin{matrix} \text{---minuendo} \\ \text{---sustraendo} \end{matrix}$$

Sumar llevada al sustraendo

1 1 1 1

Resta

0 0 0 1 1 0 1 1

4.4.3. Suma/Resta en módulo 2 (binaria) en complemento a 2

- Repasar el formato complemento a 2 para números enteros con signo

suma

- Realizar la suma de en complemento a 2 de números enteros de 1 byte 00100101 y 0111
- Los dos datos empiezan por cero, luego son positivos según el formato complemento a 2
 - extiendo los sumandos para tener todos el mismo tamaño. 0111 extendiendo el bit de signo 0 es 00000111

Llevadas -->	1 1 1
	0 0 1 0 0 1 0 1 <--AL
	+ 0 0 0 0 0 1 1 1 <--BL
Valor suma	1 3 2 2

Resultado -->	0 0 1 0 1 1 0 0 <--suma

resta

- La resta X-Y equivale a la suma X+(-Y). La resta -X-Y equivale a la suma $(-X)(-Y)$. Por lo que la ALU las restas la realiza mediante la operación suma y cambiando de signo a los operandos.
- Ejemplo: realizar la resta 27-101 en complemento a 2 utilizando registros de 1 byte

primero codifico tanto el minuendo +27 como el sustraendo +101
+27 -> 00011011
+101 -> 01100101
-101 es el complemento a 2 de +101 -> 10011011
La operación equivale a la suma $(-101)+27 \rightarrow 10011011+00011011$
Llevadas --> 1 1 1 1
1 0 0 1 1 0 1 1 <--AL
+ 0 0 0 1 1 0 1 1 <--BL
Valor suma 1 3 2 1 3 2

Resultado --> 1 0 1 1 0 1 1 0 <--suma

- ¿ Cuál es el valor del resultado?

el resultado tiene el bit de la posición más significativa a 1 por lo que su valor es negativo en complemento a 2. Si es negativo no puedo calcular su valor mediante sumas ponderadas ya que no es una representación posicional. Tengo que cambiar lo de signo para hacerlo positivo y así poder calcular su valor por suma ponderada.

Complemento a 2 del resultado 10110110 -> 01001010 cuyo valor es +74 , por lo que el valor de 10110010 es -74.

- repetir la operación cambiando de computadora y utilizando registros de 2 bytes. Basarse en el apartado anterior.

Extiendo el bit de signo del número negativo 10011011 hasta completar los 16 bits

AX <- 111111110011011 (-101)

Extiendo el bit de signo del número positivo 00011011 hasta completar los 16 bits

BX <- 000000000011011 (+27)

Extiendo el bit de signo del resultado negativo 10110110 hasta completar los 16 bits

CX <- 111111110110110 (-74)

Overflow en Complemento a 2 (C2)

- El desbordamiento u overflow ocurre en las operaciones aritméticas suma y resta cuando el resultado de la operación es de un tamaño fuera del rango de posibles representaciones, por lo que el valor resultante no es válido y provoca errores.
 - Ejemplo de suma utilizando registros de 2 bytes : $10000000+10000000 = 00000000 \Rightarrow$ Overflow
 - Error ya que $-128-128$ no es cero.
 - Si los dos sumandos son negativos el resultado no puede ser positivo

Para que el resultado fuese correcto deberíamos utilizar registros de un tamaño superior al byte, por ejemplo 9 bits. En este caso realizamos nuevamente la operación extendiendo los datos 1 bit más:

$110000000+110000000 = 100000000 \rightarrow$ no hay overflow \rightarrow la suma de dos números negativos ha dado negativo

si realizamos la operación en decimal $\rightarrow (-128)+(-128) = (-256)$

- Si los dos sumandos son positivos el resultado no puede ser negativo
- Intelx86 activa el error de overflow cuando en el resultado de una operación aritmética con signo el acarreo del bit MSB afecta al valor del resultado.



Observar que al realizar operaciones aritméticas de suma y resta, el código del resultado es idéntico en números sin signo y en complemento a 2. El código es idéntico pero su valor asociado no lo es.

4.4.4. Suma en Módulo 16 (Hexadecimal)

- Suma en módulo 16:

- el acarreo se produce al llegar o pasar el valor del módulo: 16.
- $0xF + 0x1 = 0x10$
 - $F+1=quince\ más\ uno = diecis\'is \geq 16 \rightarrow$ al resultado diecis\'is le resto 16 ($16-16=0$) y me llevo una.
- $0x3AF + 0xA = 0x3B9$
 - $F+A=quince\ más\ 10 = 25 \geq 16 \rightarrow$ al resultado veinticinco le resto 16 ($25-16=9$) y me llevo una
- $0x3A1F + 0xF4E1 = 0x12F00$
 - $F+1=quince\ más\ 1 = 16 \geq 16 \rightarrow$ al resultado diecis\'is le resto 16 ($16-16=0$) y me llevo una.

4.4.5. Resta en M\'odulo 16 (Hexadecimal)

- Todo lo visto anteriormente para n\'umeros binarios se puede realizar en cualquier otra base, por ejemplo en n\'umeros codificados en hexadecimal.
- Resta en m\'odulo 16:
 - el acarreo se produce cuando una posici\'on p del minuendo es inferior a la misma posici\'on p del sustraendo, en cuyo caso, es necesario sumar el m\'odulo 16 al minuendo y la llevada a la posici\'on siguiente $p+1$ del sustraendo:
 - $0x4308 - 0x1ABC = 0x$

	0x 4 3 0 8 <-- Minuendo
	- 0x 1 A B C <-- Sustraendo
LLevadas -->	1 1 1

	0x 2 8 4 C

- $8-C \rightarrow 8+\text{m\'odulo_}16-12=8+16-12=12=0xC$ y llevada 1 a la posici\'on siguiente
- $0-B-\text{Llevada} \rightarrow 0+\text{m\'odulo_}16-11-1=0+16-11-1=4=0x4$ y llevada 1 a la posici\'on siguiente
- $3-A-\text{Llevada} \rightarrow 3+\text{m\'odulo_}16-10-1=3+16-10=8=0x8$ y llevada 1 a la posici\'on siguiente
- $4-1-\text{Llevada} \rightarrow 4-1-1=2$

Suma en base hexadecimal en formato complemento a 2

- $0xEC + 0xAB = 0x97$
 - En binario el bit MSB es 1 significa que el valor es negativo
 - Los dos sumandos y el resultado son negativos
 - La suma de dos n\'umeros negativos da overflow si el resultado es positivo, por lo que no hay overflow
 - C2 de $0xEC \rightarrow 0xEC$ negado es $0x13$ y sumando 1 $\rightarrow 0x15$
 - C2 de $0xAB \rightarrow 0x54+1 \rightarrow 0x55$
 - C2 de $0x97 \rightarrow 0x68+1 \rightarrow 0x69$

Suma en base 8 (Octal)

- Suma en módulo 8. El acarreo se produce al llegar o pasar el valor del dígito 8.
 - $08+01 = 010$
 - $0377+06 = 0305$

4.4.6. Tipos de variables en C

- Enteros
 - char
 - short
 - int
 - long
- Reales
 - float
 - double
- Operador sizeof()
- Conversión de tipos
 - casting

4.5. Operaciones Logicas

4.5.1. Operadores BITWISE

- Bitwise: operaciones bit a bit
 - not, and, or, xor

Lenguaje C

- https://www.salesforce.com/us/developer/docs/apexcode/Content/langCon_apex_expressions_operators_understanding.htm
- [Algebra Boole](http://en.wikipedia.org/wiki/Boolean_algebra) [http://en.wikipedia.org/wiki/Boolean_algebra]
- [algebra symbols](http://en.wikipedia.org/wiki/List_of_logic_symbols) [http://en.wikipedia.org/wiki/List_of_logic_symbols]
 - Bitwise operator: and &, or |, xor ^, not ~
 - Shift operator: left <<, right signed >>, right unsigned >>>

Operador	Algebra	C
NOT	\neg $\bar{}$	\sim
OR	$\bar{\cup}$	$ $
AND	$\bar{\wedge}$	$\&$

Operador	Algebra	C
XOR	$\overline{x} \cdot \overline{y}$	\wedge
NOR	$\overline{x} + \overline{y}$	
NAND	$\overline{x} \cdot \overline{y}$	
Left SHIFT		$x << m$
Right SHIFT signed		$x >> m$
Right SHIFT unsigned		$x >>> m$

Tablas de la Verdad

x	y	$z=x \oplus y$	$z=x \cdot \overline{y}$	$z=\overline{x} \cdot y$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

Expresión Lógica

- $z = \neg x \cdot y + x \cdot \neg y$
 - Si desarrollamos la tabla de la verdad comprobamos su equivalencia con el operador XOR

4.6. Multiplicación

- Multiplicación 0xFF x 0x6
 - Realizarla en Binario
 - Observar que al multiplicar por una potencia de 2 hay un desplazamiento del multiplicando hacia la dcha
 - multiplicar = sumar y desplazar

4.7. Programación

4.7.1. funciones matemáticas

- <http://bisqwit.iki.fi/story/howto/bitmath/>
 - El código fuente está escrito en lenguaje C
- Librería libm.so del standard de C

4.7.2. Aplicación

- Desarrollar un programa que multiplique números enteros con signo.

4.8. Hardware

4.8.1. Circuitos Digitales

Básicos:Puerta lógicas

- [Puertas lógicas](http://en.wikipedia.org/wiki/Logic_gate) [http://en.wikipedia.org/wiki/Logic_gate]
 - not, and, or, xor

Complejos

- [half adder, full adder](http://en.wikipedia.org/wiki/Adder_%28electronics%29#Full_adder) [http://en.wikipedia.org/wiki/Adder_%28electronics%29#Full_adder]
- [multiplicador](http://en.wikipedia.org/wiki/Binary_multiplier) [http://en.wikipedia.org/wiki/Binary_multiplier]
 - circuito combinacional formado por puertas lógicas
 - acumulador y registro desplazador

4.8.2. Unidad Aritmetico Lógica (ALU)

- Arithmetic logic unit (ALU)
- Circuito Digital
- Conexión CPU-DRAM
 - Transferencia de Instrucciones y Datos
 - La ALU es interna a la CPU y procesa datos numéricos enteros almacenados en los registros de propósito general.

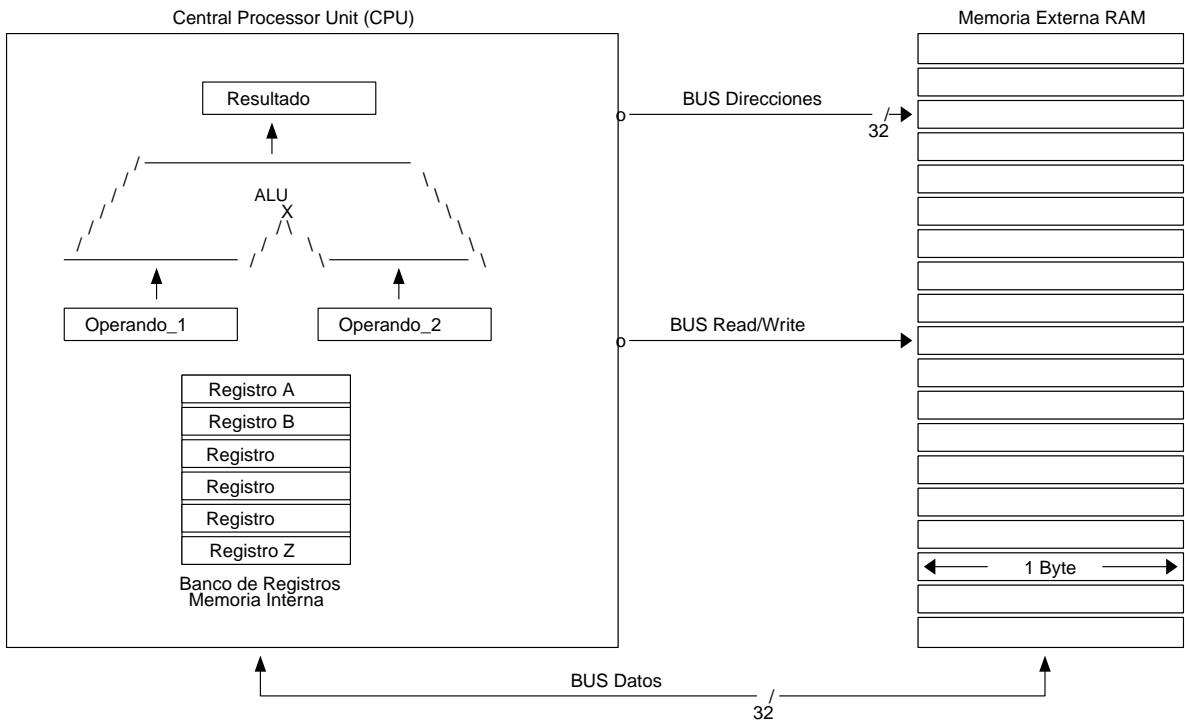


Figure 30. Arquitectura Intel x86 de 32 bits

4.8.3. Registro de flags EFLAG

- El registro de flags EFLAGS es un registro de memoria interno a la CPU Intel x86
- Cada bit del registro de 32 bits es un banderín o flag que se activa en función del resultado de la operación realizada por la última instrucción máquina ejecutada.

Table 5. 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 CF:
 - 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 OF:
 - 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 SI hay Overflow? -> Esa suma NO es correcta, ya que para realizar la operación con 9 bits los operandos tienen que ser de 9 bits y por lo tanto hay que extender el bit de signo. La suma con 9 bits sería:

111111111
 + 000000001

 000000000 -> Hay acarreo y NO hay Overflow. CF=1 y OF=0.

Nunca va haber overflow si sumamos datos de signo contrario

A=11110000
 B=00010100
 A-B=A+(-B)

A : 11110000
 -B : +11101100

A+B: 11011100 -> Hay acarreo pero NO overflow. La suman de dos datos negativos da como resultado un número también negativo. CF=1 y OF=0

A=10000000
 B=10000000
 A+B

Para hacer la suma con 9 dígitos en lugar de 8 bits, extiendo los dos operandos hasta completar los 9 dígitos

A : 110000000
 B : +110000000

A+B: 100000000

Observamos que no hay overflow en el caso de que utilizasemos 9 dígitos. Pero si la ALU está operando con registros de 8 bits SÍ HAY overflow. Los dos sumandos son negativos (bit de signo posición 7^a) y el bit de signo del resultado (bit posición 7^a) es positivo luego el resultado es erróneo. En la posición 7 también hay acarreo -> CF=OF=1

- Se ve nuevamente en el próximo capítulo [Programación en Lenguaje Ensamblador \(x86\)](#)

4.8.4. Float Point Unit-FPU

- Unidad de procesamiento de datos en coma flotante
- Antiguamente era una unidad no integrada en la CPU denominada coprocesador matemático
- Utiliza registros específicos denominados SSE distintos de los Registros de Propósito General utilizados por la ALU para realizar operaciones con números enteros.

Chapter 5. Representación de las Instrucciones

5.1. Temario

1. Representación de instrucciones
 - a. Lenguaje máquina, lenguaje ensamblador y lenguajes de alto nivel
 - b. Formato de instrucción
 - c. Tipos de instrucción y modos de direccionamiento

5.1.1. Bibliografía

- Tema referenciado en el libro de texto W. Stalling
 - Capítulo 12: Conjuntos de Instrucciones : Características y Funciones (Datos, Operandos y Operaciones)
 - Capítulo 13: Conjuntos de Instrucciones: Formatos de instrucciones y Modos de Direccionamiento (Lenguaje Ensamblador)
 - Apéndice B: Lenguaje Ensamblador y Toolchain

5.2. Objetivos

- Analizar la arquitectura del repertorio de las instrucciones máquina (Formato de instrucciones, formato de datos, operaciones y direccionamiento de operandos) de arquitecturas ISA en general.

5.2.1. Requisitos

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

5.3. Introducción

5.3.1. Lenguaje máquina y el formato binario

Los lenguajes de alto nivel como Java, Python, C, etc ... se desarrollaron para facilitar la tarea de programar algoritmos, estructuras de datos, etc...utilizando un lenguaje sencillo de manejar por los programadores. En cambio, los datos y las instrucciones que manejan las CPU de las computadoras están en otro lenguaje, el lenguaje MAQUINA, que depende del tipo de procesador (intel,AMD,RISC-

V,etc...) de la computadora. El lenguaje máquina de un procesador intel de nuestra computadora difiere del lenguaje MAQUINA del procesador arm de un smartphone.

Al igual que los datos, las instrucciones también es necesario codificarlas en un formato BINARIO. Los programas en lenguaje máquina formados por datos e instrucciones binarias están preparados para ser cargados en la memoria principal RAM y ser procesados por la CPU.

5.3.2. El lenguaje máquina y el lenguaje ensamblador

Representación de las instrucciones

- En este tema se trata de la representación e interpretación de las instrucciones.
- Las instrucciones se pueden representar en dos lenguajes
 - Lenguaje máquina en formato binario : 010101010111111000011111
 - El lenguaje binario implica un *formato de la instrucción*.
 - Lenguaje símbolico o lenguaje ensamblador en formato texto : *fin: ADD 0x33,resultado*
 - El lenguaje ensamblador implica una *sintaxis*
- La representación de las instrucciones en lenguaje binario permite su almacenamiento en la memoria principal así como facilitar el ciclo de instrucción mediante su decodificación y ejecución por parte de la CPU.
- La representación de las instrucciones en lenguaje simbólico tiene como objetivo facilitar la tarea del programador en la interpretación de las instrucciones y en el desarrollo de programas en lenguaje ensamblador.

5.3.3. Lenguajes de Alto Nivel

- Aunque en los años 1960 se programaba en el lenguaje máquina y lenguaje ensamblador, esto resultaba muy poco productivo ya que refequería mucho esfuerzo y tiempo.
- La solución fue abstraer funciones de la computadora mediante lenguajes de alto nivel:
 - <https://www.tiobe.com/tiobe-index/>
- El empleo de lenguajes de alto nivel efectivos para los programadores requiere de un traductor al lenguaje máquina de la computadora.



Figure 31. Proceso de Compilación

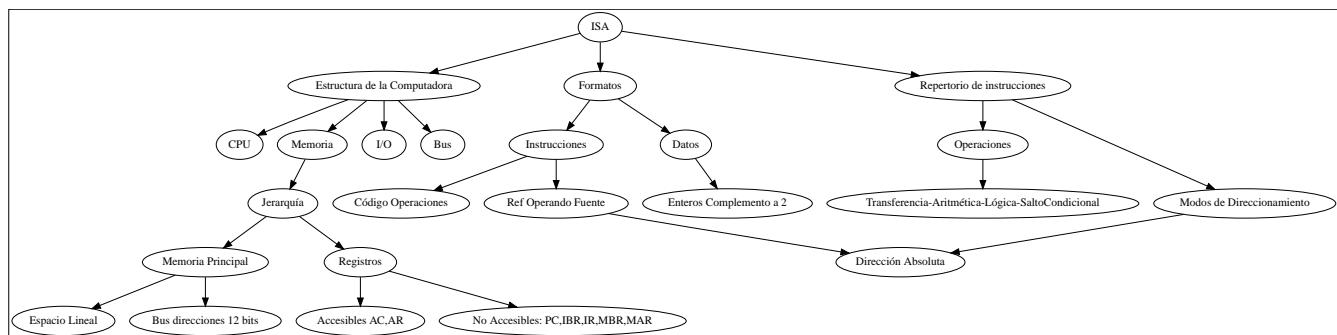
5.4. Arquitectura

5.4.1. Contexto

- El significado del término arquitectura en una computadora depende del contexto.
 - Arquitectura de un computadora: es la organización en grandes bloques como CPU,Memoria,controladores de periféricos,buses, etc
 - Arquitectura de un procesador: Es el repertorio de instrucciones y datos capaz de interpretar y ejecutar la cpu. Puede haber dos procesadores fabricados con distinta estructura interna pero que procesen las mismas instrucciones y datos ,es decir, que procesen el mismo el lenguaje máquina y por lo tanto tienen la misma arquitectura. En este contexto utilizaremos el termino ISA (Instruction Set Architecture). La ISA de un procesador AMD64 y un procesador Intel x86-64 es la misma, operan con el mismo lenguaje máquina. Un programa binario en AMD64 es compatible con Intel x86-64.
 - Microarquitectura de un procesador: es la organización interna de un procesador.

5.4.2. Instruction Set Architecture (ISA)

- La arquitectura del repertorio de instrucciones (ISA: instruction set architecture) de una computadora comprende definir principalmente:
 - Estructura de la Computadora: CPU-Memoria-Bus-I/O
 - **La representación y formato de los datos :**
 - **La representación y formato de las instrucciones**
 - Repertorio de instrucciones: Las operaciones y modos de direccionamiento que ha de interpretar y ejecutar la computadora.
- La arquitectura ISA define la potencialidad de la CPU de la computadora.
- El diseño de la arquitectura ISA va a afectar al rendimiento de la computadora.
 - Programa binario resultado de la compilación del programa fuente.
 - Ocupación de Memoria
 - Implementación (dificultad) y rendimiento de la CPU.



Ejemplos: Intel x86, Motorola 68000, MIPS, ARM

- Ver [Apéndice Lenguajes Ensamblador](#)

5.5. Procesadores Intel con arquitectura x86

5.5.1. Nomenclatura

General

- Los procesadores intel reciben nombres 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](https://computerhoy.com/noticias/hardware/que-significan-numeros-letras-procesadores-intel-56812) [<https://computerhoy.com/noticias/hardware/que-significan-numeros-letras-procesadores-intel-56812>]: El primer dígito del código indica la generación (en el 2017 salió la 8^a 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://www.intel.com/content/www/us/en/products/processors/core/x-series.html) [<https://www.intel.com/content/www/us/en/products/processors/core/x-series.html>]

- <https://ark.intel.com/products/series/123588/Intel-Core-X-series-Processors>

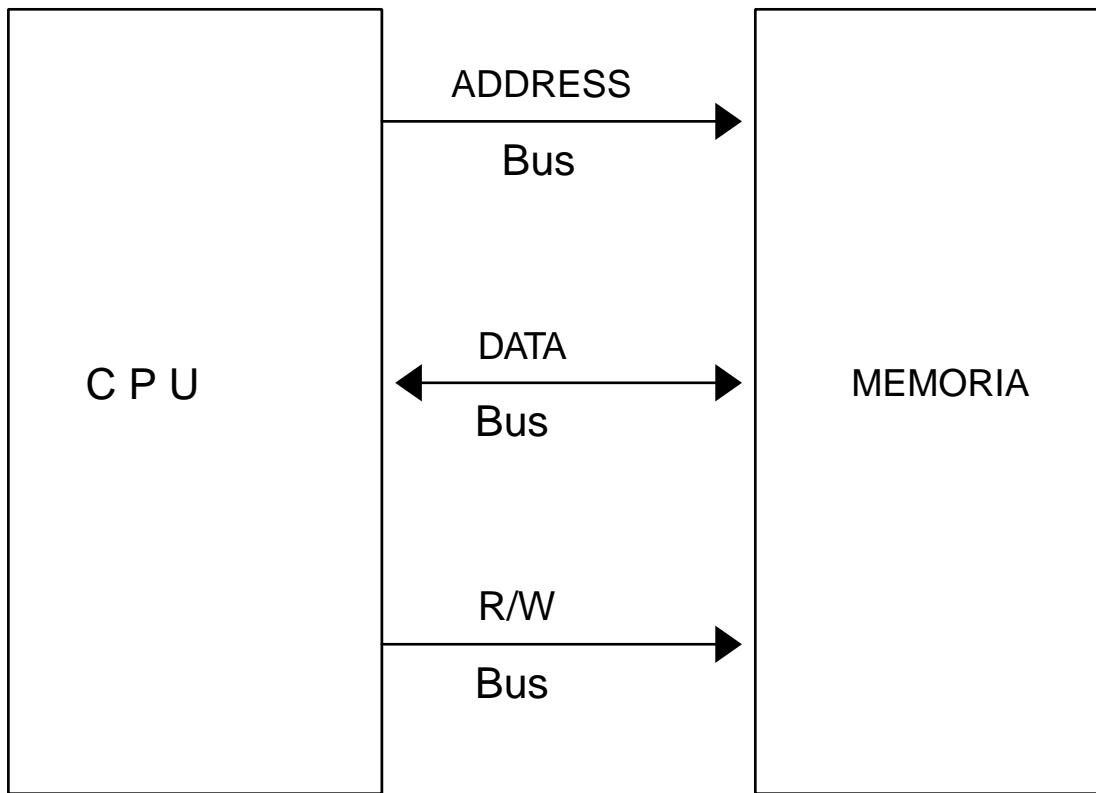
- https://es.wikipedia.org/wiki/Anexo:Procesadores_AMD

- COMPETENCIA INTEL-AMD año 2018 en computadoras de sobremesa.

- [AMD Ryzen 2nd Generation - INTEL Core i7-8086K](https://www.techradar.com/news/intel-coffee-lake-release-date-news-and-rumors) [<https://www.techradar.com/news/intel-coffee-lake-release-date-news-and-rumors>]

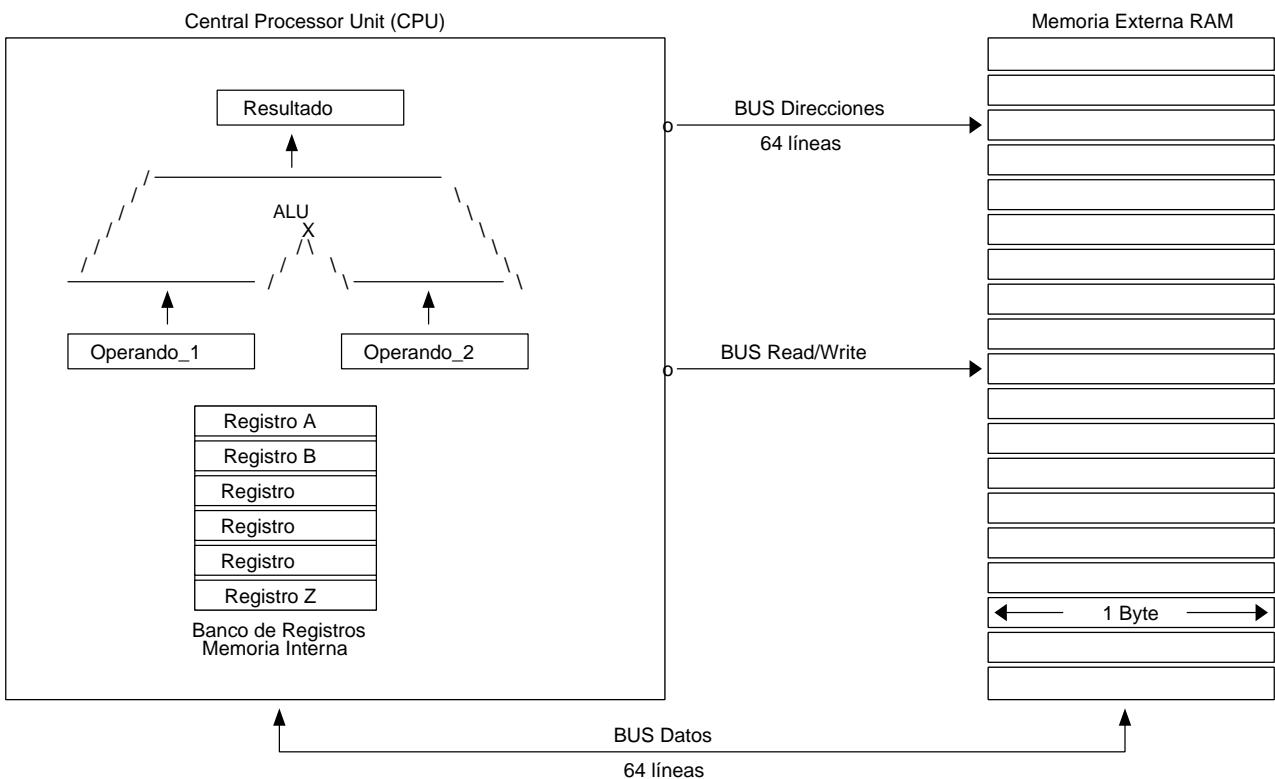
5.6. Estructura de la Computadora

5.6.1. Arquitectura



5.6.2. CPU

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



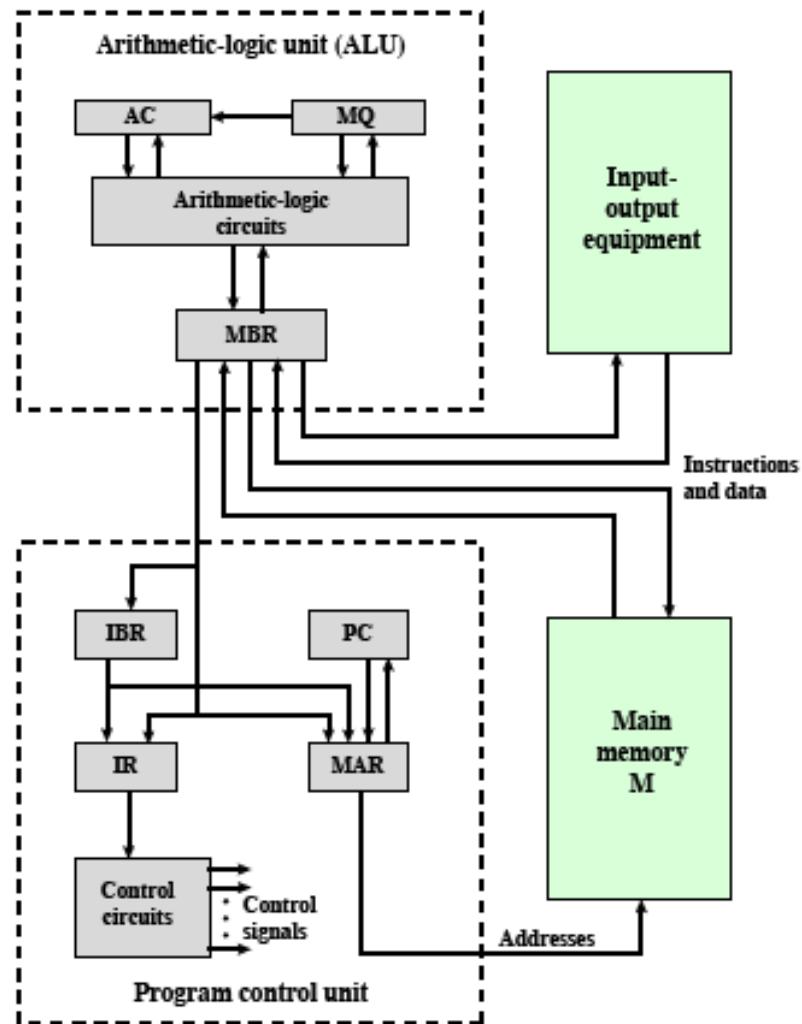


Figure 2.3 Expanded Structure of IAS Computer

Figure 32. IAS Structure

5.6.3. Memoria

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

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 hexádecimal
- Direccionamiento: bytes : notación hexadecimal

DIRECCIONES	CONTENIDO
0x00000000	1 0 1 0 1 0 1 0
0x00000001	1 0 1 0 1 0 1 0
0x00000002	1 0 1 0 1 0 1 0
	1 1 1 1 1 1 1 1
0x00000009	
0x0000000a	
0x0000000f	

Registros internos a la CPU

Registros NO visibles al programador

- Registros 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 → 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.

Registros visibles al programador

- El programador utiliza dichos registros para almacenar datos (escribir y leer).
- Hay operaciones (suma, resta, etc...) donde los operandos pueden estar en los registros.
- El acceso de la CPU a un registro es mucho más RAPIDO que el acceso a una posición de la memoria principal RAM.

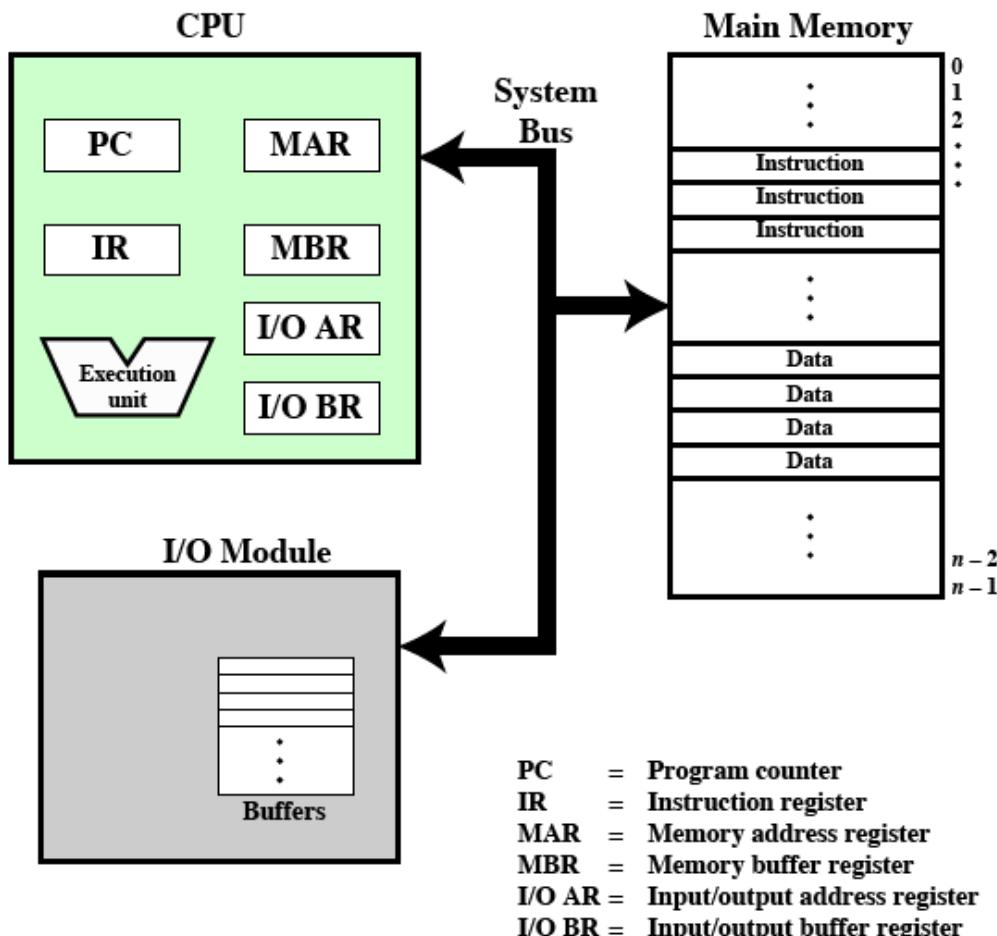


Figure 33. IAS_Architecture

5.7. Elementos de una Instrucción Máquina

- Código de Operaciones:
 - La instrucción debe de especificar que operación debe de realizar la CPU. Operaciones como las aritméticas de suma y resta, operaciones lógicas como not y and, operaciones de transferencia de datos entre posiciones de la memoria principal, operaciones de entrada y salida como la transferencia de datos del disco duro a la memoria principal, etc
- Source Operand Reference:
 - Una operación puede requerir el procesamiento de uno o más datos. Por ejemplo la operación lógica not requiere de un operando.
- Target Operand Reference:

- Una operación de suma requiere de dos operandos, uno es el operando fuente y otro el operando destino.
- Result Reference:
 - Una operación de suma requiere salvar el resultado de la operación.
- Next Instruction Reference:
 - Una vez finalizada la ejecución de la instrucción es necesario indicar a la CPU donde esta almacenada la próxima instrucción a ejecutar a través del Contador de Programa PC.

5.7.1. Direcciones implícitas

- Direcciones que no aparecen explícitamente en la instrucción. Ejemplos:
 - Próxima instrucción en el Contador de Programa
 - Resultado en el Acumulador
 - etc

5.7.2. Tipos de Arquitecturas de Operando: Ejemplos

- 3 Tipos
 - Arquitectura orientada a Acumulador: Un operando está implicitamente en el Acumulador
 - Arquitectura orientada a Stack ([Apéndice Pila](#)):
 - Los operandos se introducen o extraen de la pila interna de la CPU
 - Concepto de pila: push/pop → empujar/extraer → el primero en entrar es el último en salir → First Input Last Output
 - SP: Registro Stack Pointer : registro que apunta al Top de la pila (parte alta de la pila)
 - Arquitectura orientada a Registros:
 - Dos tipos: Reg/Mem y Load/Store, como es el caso de la arquitectura amd64 y arm respectivamente.
 - Reg/Mem : para que la instrucción se ejecute uno de los dos operandos debe de estar en un registro
 - Load/Store: Los dos operandos deben de estar en dos registros para que dicha instrucción se ejecute
- Ejemplo: código para realizar la operación $C=A+B$ en 4 arquitecturas de operando diferentes.

Stack	Acumulator	Register/Memory	Load/Store
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

- Los nombres de las variables, A, B,C son referencias a la Memoria Principal.

- Descripción RTL

- Stack: $M[SP] \leftarrow M[A], SP \leftarrow SP-1; M[SP] \leftarrow M[B], SP \leftarrow SP-1; M[SP+1] \leftarrow M[SP]+M[SP+1], SP \leftarrow SP+1;$
 - **Add** → NO hay referencia ni al operando fuente ni al operando destino.
 - Los operandos han de cargarse previamente en la pila
- Acumulator: $AC \leftarrow M[A]; AC \leftarrow AC + M[B]; C \leftarrow M[AC]$
 - **Add B** → NO hay referencia al operando DESTINO
 - El Operando destino a de cargarse previamente en el acumulador.
- Reg/Mem: $R1 \leftarrow M[A]; R3 \leftarrow R1 + M[B]; M[C] \leftarrow R3$
 - **Add R3,R1,B** → NO se puede referencia a más de un operando en MEMORIA
 - Si un operando está almacenado en la memoria, el resto a de cargarse previamente en los registros.
- Load/Store: $R1 \leftarrow M[A]; R2 \leftarrow M[B]; R3 \leftarrow R1 + R2; M[C] \leftarrow R3.$
 - **Add R3,R1,R2** → Solamente se hacen referencias a REGISTROS, ninguna referencia a memoria
 - Los operandos fuente y destino han de cargarse previamente en los registros



La arquitectura x86 está orientada a Reg/Mem, por lo que no se puede referenciar en la misma instrucción a un operando fuente en MEMORIA y el operando destino también en MEMORIA, es decir, ambos operandos referenciados a MEMORIA.

- Ejemplo de código para realizar la opeación **(A-B)/(DxE+C)** según 4 arquitecturas ISA diferentes: arquitectura con 3 operandos referenciados, con 2 operandos referenciados, con 1 operando referenciado y ningnún operando referenciado

<u>Instruction</u>	<u>Comment</u>
SUB Y, A, B	$Y \leftarrow A - B$
MPY T, D, E	$T \leftarrow D \times E$
ADD T, T, C	$T \leftarrow T + C$
DIV Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions

<u>Instruction</u>	<u>Comment</u>
MOVE Y, A	$Y \leftarrow A$
SUB Y, B	$Y \leftarrow Y - B$
MOVE T, D	$T \leftarrow D$
MPY T, E	$T \leftarrow T \times E$
ADD T, C	$T \leftarrow T + C$
DIV Y, T	$Y \leftarrow Y \div T$

<u>Instruction</u>	<u>Comment</u>
LOAD D	$AC \leftarrow D$
MPY E	$AC \leftarrow AC \times E$
ADD C	$AC \leftarrow AC + C$
STOR Y	$Y \leftarrow AC$
LOAD A	$AC \leftarrow A$
SUB B	$AC \leftarrow AC - B$
DIV Y	$AC \leftarrow AC \div Y$
STOR Y	$Y \leftarrow AC$

- (b) Four-address instructions

- 4º Caso: Arquitectura de Operando tipo Stack:

- $M[SP] \leftarrow M[C]; M[SP] \leftarrow M[E]; M[SP] \leftarrow M[D]; MUL; ADD; M[SP] \leftarrow M[B]; M[SP] \leftarrow M[A]; SUB; DIV$
- push C; push E; push D; mul; add; push B; push A; sub; div;

5.8. Representación de las instrucciones en lenguaje ensamblador (ASM) para computadoras en general

5.8.1. Lenguaje Máquina Binario

Ejemplos

- `objdump -d módulo_binario`
- Ver [Apéndice Lenguajes Ensamblador](#)

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

5.9. Operandos: Modos de Direcciónamiento

5.9.1. Localización

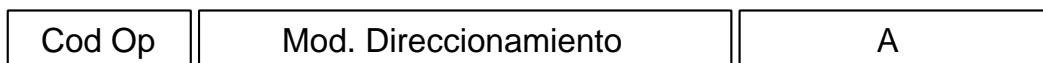
- Posibles ubicaciones de los operandos.
 - En la propia instrucción
 - Memoria interna: registros CPU
 - Memoria Principal: memoria DRAM
 - Memoria i/o: registros en controladores de entrada/salidas denominados puertos.

5.9.2. Direcciones referenciadas durante el ciclo de instrucción

- Durante el ciclo de instrucción se pueden referenciar:
 - Una dirección para referenciar a la instrucción
 - Una dirección para el operando primero
 - Una dirección para el operando segundo
 - Una dirección para el resultado
 - Una dirección que refiera a la siguiente instrucción
- Tipos de instrucciones según el número de direcciones referenciadas durante su ejecución.
 - Instrucciones sin operando, con un operando, con múltiples operandos.
 - Depende de la arquitectura: Acumulador (Ej: máquina IAS), Registro-Memoria(Ej: máquina x86), Load/Store (Ej:ARM), Stack (Ej: máquina JVM), Memoria-Memoria
 - referencias implícitas al operando

5.9.3. Direccionamiento para un lenguaje general

Formato de instrucción: Campos



- **Ejemplo particular** de una estructura del formato de instrucción en tres campos en una arquitectura ISA.
 - Código de Operación: mover, cargar, sumar, restar, etc
 - Código A: campo de operando : hace referencia a la localización del operando
 - Código Mod. Direc: representa el modo de interpretar el campo A
- EA: Efective Address : Dirección efectiva donde está localizado el operando
- Op: Operando .Es el dato contenido en la dirección efectiva EA.
- Los datos *operando* Op pueden estar almacenados en:
 1. Memoria externa RAM
 - a. Una dirección de memoria conteniendo un dato.
 - b. Una dirección de memoria conteniendo una instrucción. El dato es uno de los campos de la propia instrucción. Direccionamiento Inmediato.
 2. Memoria interna GPR
 - a. Registros rax,eax,...

Tipos de direccionamiento

- La *dirección* de referencia efectiva E.A. de la ubicación del operando se obtiene según los distintos modos de direccionamiento.
- El modo de direccionamiento está codificado en el campo M.D.
- Inmediato:
 - El operando se obtiene del campo de la propia instrucción.
 - EA= no existe
 - Op=A
- Directo:
 - El operando está en la memoria externa. El campo de operando contiene la dirección efectiva
 - EA=A
 - Op=M[EA]
- Registro:
 - El operando está en la memoria interna. El campo de operando contiene la referencia del Registro.
 - EA=A
 - Op=R
- Indirecto:
 - La dirección efectiva esta almacenada en una posición de memoria externa o interna.
 - EA=M[A] o R
 - Op=M[M[A]] o M[R]
- Desplazamiento:
 - La dirección efectiva del operando se obtiene mediante una operación aritmética entre una dirección base y un desplazamiento relativo a la dirección base. La dirección base se toma como referencia y el desplazamiento es relativo a la dirección base.
 - a. Relativo al contador de programa PC:
 - La dirección base es implícitamente el contador de programa y el desplazamiento está en el campo de operando.
 - EA=PC+A
 - Op=M[EA]
 - b. Relativo a Base:
 - El desplazamiento está en el campo de operando y la dirección base está en el registro.
 - EA=R+A
 - Op=M[EA]

c. Indexado:

- El desplazamiento está en el registro y la dirección base está en el campo de operando.
 - $EA = A + R$
 - $Op = M[EA]$
- Para hacer referencia a los operandos fuente o destino la arquitectura de la instrucción es muy *flexible* ya que se dispone de distintos modos de direccionar dichos operandos.

5.10. Operaciones

5.10.1. Códigos de Operación

- La codificación del conjunto de operaciones depende de cada arquitectura ISA.

5.10.2. Tipos de Operaciones

- Categorías:
 - Data Processing: Arithmetic and logic instructions
 - Data Load/Store: Movement of data into or out of register and/or memory locations
 - Data Movement: I/O instructions
 - Control: Test and Branch instructions

Table 12.3 Common Instruction Set Operations

Type	Operation Name	Description
Data transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand
Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT (complement)	Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
Transfer of control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued

- El repertorio puede ser: reducido/extenso, complejo/sencillo.

Chapter 6. Programación en Lenguaje Ensamblador (x86): Construcciones básicas de los lenguajes de alto nivel.

6.1. 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 función o subrutina

6.2. Introducción

6.2.1. 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.

6.2.2. Requisitos

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

6.3. Estructura de la Computadora

6.3.1. 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

6.3.2. Memoria

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

6.3.3. 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 hexádecimal
- Direccionamiento: bytes : notación hexadecimal

6.3.4. Registros internos a la CPU

introducción

- Registros 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 → 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.

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	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

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

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](http://en.wikipedia.org/wiki/FLAGS_register_(computing)) [http://en.wikipedia.org/wiki/FLAGS_register_(computing)]
- Unicamente nos fijamos en los flags OSZAPC.

Table 6. 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 8^a 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 7^a)

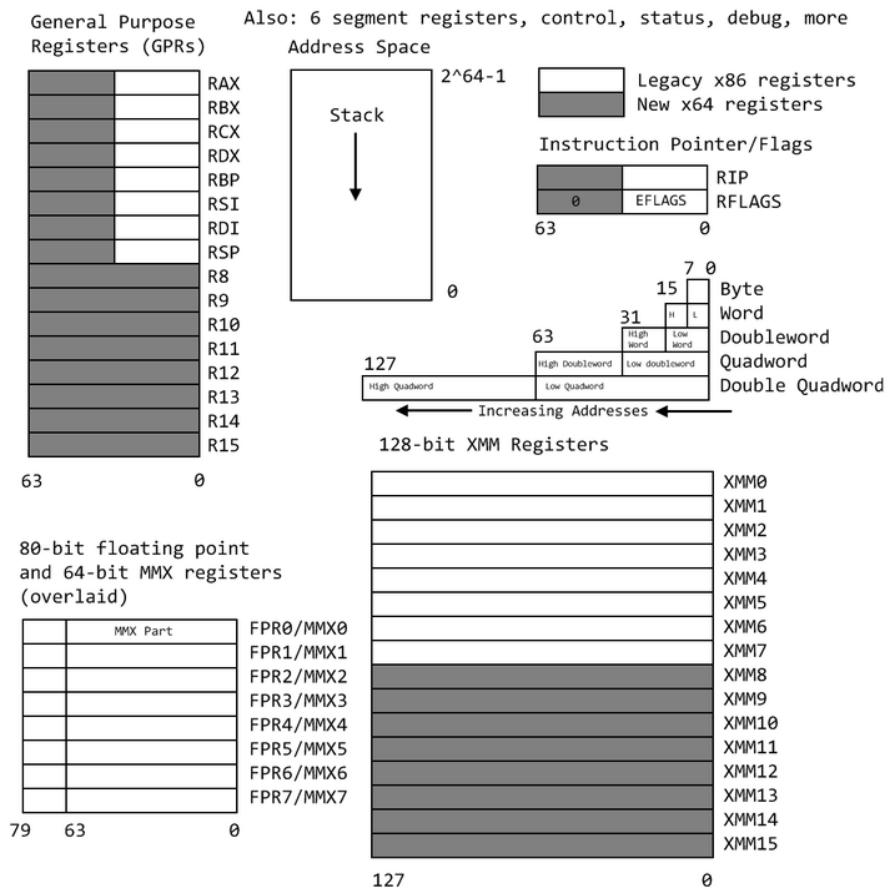
A : 11110000
 -B : +11101100

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 (8^a) y la anterior (7^a) 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 (7^a, bit de signo) de valor 0. Los dos sumandos son negativos (bit de signo posición 7^a) y el bit de signo del resultado (bit posición 7^a) es positivo luego el resultado es erróneo.

Otros Registros



- 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](#)

6.4. Lenguaje Intel versus Lenguaje AT&T

6.4.1. 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.

6.4.2. 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

Table 7. 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+



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](#) y [Dr. Paul Carter](#)

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

Table 8. 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 sufijo :
- 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.



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

6.4.3. 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.



Figure 34. Proceso de Compilación

6.4.4. Código Máquina

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**.

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**: $RSP \leftarrow RSP - 16$
 - 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 operación $PC \leftarrow PC + \text{tamaño de la instrucción en bytes}$.
 - ¿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*

6.4.5. Assembler "as"

Directivas

- Directivas del traductor ensamblador "as" utilizado por el sistema GNU/linux para el lenguaje ensamblador AT&T.
 - Al assembler de GNU también se le conoce como "gas".
 - [binutils](https://www.gnu.org/software/binutils/) [https://www.gnu.org/software/binutils/] → [as assembler](https://sourceware.org/binutils/docs-2.26/as/index.html) [https://sourceware.org/binutils/docs-2.26/as/index.html]: manual oficial
 - [gas ref card](http://www.coranac.com/files/gba/re-ejected-gasref.pdf) [http://www.coranac.com/files/gba/re-ejected-gasref.pdf]
 - .word reserva 2 bytes en amd64.
 - [oracle](https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html) [https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html]
 - [opciones para x86 y x86-64](https://sourceware.org/binutils/docs-2.26/as/i386_002dDependent.html#) [https://sourceware.org/binutils/docs-2.26/as/i386_002dDependent.html#]

Manual

- [binutils as](https://sourceware.org/binutils/docs/as/) [https://sourceware.org/binutils/docs/as/]: manual oficial
- [Linux Assembly howto](http://asm.sourceforge.net/howto/Assembly-HOWTO.html) [http://asm.sourceforge.net/howto/Assembly-HOWTO.html]: referencias a diferentes assemblers
- [MIT](http://web.mit.edu/gnu/doc/html/as_1.html) [http://web.mit.edu/gnu/doc/html/as_1.html]

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

6.5.1. Tipos de Datos

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 alfanuméricicos: código ASCII

Directivas de la Sección de Datos



Recomendable leerse los seis primeros apartados por lo menos y sobre todo lo referente a las [directivas dependientes de la arquitectura x86](https://sourceware.org/binutils/docs-2.26/as/i386_002dDependent.html#i386_002dDependent) [https://sourceware.org/binutils/docs-2.26/as/i386_002dDependent.html#i386_002dDependent:]

Table 9. 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
.common 100	reserva 100 bytes sin inicializar y puede ser referenciado globalmente
.lcomm bucle, 100	reserva 100bytes referenciados con el símbolo local bucle. Sin inicializar.
.space 100	reserva 100 bytes inicializados a cero

Directivas	descripción
.space 100, 3	reserva 100 bytes inicializados a 3
.string "Hola"	añade el byte 0 al final de la cadena
.asciz "Hola"	añade el byte 0 al final de la cadena
.ascii "Hola"	no añade le caracter NULL de final de cadena
.byte 3,7,-10,0b1010,0xFF,0777	tamaño 1Byte y formatos decimal,decimal,decimal,binario,hexadecimal,octal
.2byte 3,7,-10,0b1010,0xFF,0777	tamaño 2Bytes
.word 3,7,-10,0b1010,0xFF,0777	tamaño 2Bytes
.short 3,7,-10,0b1010,0xFF,0777	tamaño 2B
.4byte 3,7,-10,0b1010,0xFF,0777	tamaño 4B
.long 3,7,-10,0b1010,0xFF,0777	tamaño 4B
.int 3,7,-10,0b1010,0xFF,0777	tamaño 4B
.8byte 3,7,-10,0b1010,0xFF,0777	tamaño 8B
.quad 3,7,-10,0b1010,0xFF,0777	tamaño 8B
.octa 3,7,-10,0b1010,0xFF,0777	formato octal
.double 3.14159, 2 E-6	precisión doble
.float 2E-6, 3.14159	precisión simple
.single 2E-6	precisión simple
.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

6.5.2. Tamaño del operando x86

- Tamaño del operando: sufijos de los MNEMÓNICOS.

q (quad) 8bytes
l (long) 4bytes
w (word) 2bytes

- Ejemplos:

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

6.5.3. Alineamiento de Bytes: Big-LittleEndian

- Los bytes de un dato de varios bytes se pueden almacenar en memoria en sentido MSB-_LSB ó MSB_-LSB
- Alineamiento *LittleEndian*: 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)

DIRECCIONES	CONTENIDO
0x00000	
0x00001	
0x00002	
0x40000	00
0x40001	AF
0x40002	BF
0x40003	CF
0xfffff	

- 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 LittleEndian.

- 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.

6.6. Operandos: Modos de Direccionamiento

6.6.1. 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".

6.6.2. 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:



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 instrucción. 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:	<p>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.</p>
	<p>sintaxis: una etiqueta definida por el programador ejemplo: <code>je somePlace</code> . Salto a la dirección marcada por la etiqueta <code>somePlace</code> si el resultado de la operación anterior activa el flag ZF=1 del registro RFLAG.</p>
INDEXADO:	<p>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_index, MAS el offset. $EA = Offset + R_Base + R_índice * Scale$</p>
	<p>sintaxis: lista de valores separados por coma y entre paréntesis (base_register, index_register, scale) y precedido por un offset. ejemplo: <code>movl \$0x6789cdef, -16(%edx, %eax, 4)</code> . La dirección efectiva del destino es EDX + EAX*4 - 16.</p>
INDIRECTO:	<p>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.</p>
	<p>sintaxis: (base_register) ejemplo: <code>movl \$0x6789cdef, (%edx)</code> . La dirección efectiva del destino es EDX. EDX es un puntero.</p>
RELATIVO: registro base más un offset:	<p>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.</p>
	<p>sintaxis: registro entre paréntesis y el offset inmediatamente antes del paréntesis. ejemplo: <code>movl \$0xaabbccdd, -12(%eax)</code> . La dirección efectiva del operando destino es EAX-12</p>

Ejemplos

Table 10. 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.

6.7. Repertorio de Instrucciones: Operaciones

6.7.1. Manuales de referencia

Lenguaje Intel

- Manuales oficiales
 - [Intel](http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html) [http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html]: Vol 2
 - [AMD](http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/) [http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/]: apartado Manuals : vol 3
 - [binutils](https://sourceware.org/binutils/docs-2.25/) [https://sourceware.org/binutils/docs-2.25/]:
 - [as: traductor assembler](https://sourceware.org/binutils/docs-2.25/as/index.html) [https://sourceware.org/binutils/docs-2.25/as/index.html]: opciones de la línea de comandos, directivas, tipos de datos, etc
 - [ld: linker](https://sourceware.org/binutils/docs-2.25/ld/index.html) [https://sourceware.org/binutils/docs-2.25/ld/index.html]
 - [objdump, readelf,](https://sourceware.org/binutils/docs-2.25/binutils/index.html) [https://sourceware.org/binutils/docs-2.25/binutils/index.html]
- Manuales no oficiales:
 - [manual Intel quick](http://www.felixcloutier.com/x86/) [http://www.felixcloutier.com/x86/]: **recomendado**
 - [intel descriptivo i386](http://www.cs.nyu.edu/~mwalfish/classes/ut/s13-cs439/ref/i386/toc.htm) [http://www.cs.nyu.edu/~mwalfish/classes/ut/s13-cs439/ref/i386/toc.htm]
 - [Repertorio ISA y Formato de Instrucción](http://www.cs.nyu.edu/~mwalfish/classes/ut/s13-cs439/ref/i386/c17.htm) [http://www.cs.nyu.edu/~mwalfish/classes/ut/s13-cs439/ref/i386/c17.htm]
 - [AT&T Solaris Manual amd64-i386](http://docs.oracle.com/cd/E19253-01/817-5477/817-5477.pdf) [http://docs.oracle.com/cd/E19253-01/817-5477/817-5477.pdf]: lenguaje y traductor assembler.
 - [Saltos Condicionales](http://www.unixwiz.net/techtips/x86-jumps.html) [http://www.unixwiz.net/techtips/x86-jumps.html]

Lenguaje AT&T

- [Oracle Solaris ASM](https://docs.oracle.com/cd/E53394_01/html/E54851/eqbsu.html) [https://docs.oracle.com/cd/E53394_01/html/E54851/eqbsu.html]
 - En este documento a la sintaxis AT&T la denomina "Oracle Solaris".

6.7.2. TRANSFERENCIA

Nombr e	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	

Nombr e	Comentario	Código	Operación	O D I T S Z A P C
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: ± =Afectado por esta instrucción, ? =Indefinido luego de esta instrucción

6.7.3. ARITMÉTICOS

Nombr e	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 !)	± ± ± ± ±

Nombr e	Comentario	Código	Operación	O D I T S Z A P C
DEC	Decrementar	DEC Op	Op:=Op-1 (El Carry no resulta afectado !)	± ± ± ± ±
CMP	Comparar	CMP Fuente,Destino	Destino-Fuente	± ± ± ± ± ±
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 ±

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

6.7.4. LÓGICOS

Nombr e	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 ± ± ? ± ±

6.7.5. MISCELÁNEOS

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

6.7.6. SALTOS (generales)

- [wiki x86 assembly](https://en.wikibooks.org/wiki/X86_Assembly/Control_Flow#Jump_if_Lesser) [https://en.wikibooks.org/wiki/X86_Assembly/Control_Flow#Jump_if_Lesser]

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	

6.7.7. 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)

6.7.8. 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=positivo.
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

6.7.9. 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*

6.7.10. Códigos de Operación

- **MOV** [<http://www.cs.nyu.edu/~mwalfish/classes/ut/s13-cs439/ref/i386/MOV.htm>]

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



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

Programas Ejemplo

6.8. Mnemónicos Básicos (Explicados)

6.8.1. 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 RD: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

- **idiv**
 - $EAX \leftarrow \text{Cociente}\{[EDX:EAX]/M[\text{Op_fuente}]\}$, $EDX \leftarrow \text{Resto}\{[EDX:EAX]/M[\text{Op_fuente}]\}$
- 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

6.8.2. Procesamiento Condicional

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

6.8.3. Saltos

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)
```

6.8.4. Desplazamiento y rotación

- **sar,sal** : Shift Arithmetic Right, Shift Arithmetic Left.
 - desplazamiento aritmético: El dígito 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 dígito entrante es el bit de signo el operando en EDX.
- **shr,shl**
- desplazamiento lógico: entran ceros
 - Ejemplos de multiplicación y división
- ROL,ROR
 - el bit que sale fuera se copia en CF
 - Aplicación: conversión endianess

6.8.5. 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
```

6.9. Formato de Instrucción: ISA Intel x86-64

- Apéndice [Formato Instrucción](#)

6.10. Subrutinas

6.10.1. 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.

6.10.2. Lenguaje C: Sentencia Función

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.

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 short.
 - Tipo del valor de retorno: el tipo del valor de retorno es int.

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

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()*

6.10.3. 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()

6.10.4. Pila/Frame

- Ver concepto de pila en el [Apéndice](#).
- 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.

6.10.5. 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),%ebx    #capturo el 1º argumento
    mov 16(%ebp),%ecx   #capturo el 2º argumento
    XXX XXX
    XXX XXX
# Epílogo
    mov resultado,%eax #inicializo el valor de retorno
        pop xxx          #recuperar registros que se salvaron en el
    prólogo
        pop xxx
            mov %ebp,%esp
            pop %ebp
    ret

```

6.10.6. Registros a Preservar

Refs

- [ABI x86-32](#)
- [Convenio de Llamada MicroSoft](#)

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.

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

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.

6.10.7. 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
```

6.10.8. 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`: $ESP \leftarrow ESP - 4$ y $M[ESP] \leftarrow PC$
 - b. Salta a la etiqueta *sumMtoN*: $PC \leftarrow sumMtoN$
- básicamente la instrucción `call` es una salto con retorno a la dirección donde fue interrumpida la rutina llamante.

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

6.10.9. 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 $\leftarrow M[ESP]$** : 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 $\leftarrow ESP + 4$**
- 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.

6.10.10. Estado de la pila

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 correspondiente 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 EIP,EBP,ESP:
 - RIP:
 - EBP:
 - ESP:

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 correspondiente a main.
 - El último dato apilado en el frame main es la *dirección de retorno* a main desde *sumMtoN*
- Contenido de los registros EIP,EBP,ESP:
 - RIP:
 - EBP:

- ESP:

Creación del nuevo frame *sumMtoN*

- Estado de la pila después de ejecutar:

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

- Contenido de los registros EIP,EBP,ESP:

- EIP:
- EBP:
- ESP:

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 correspondiente 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 EIP,EBP,ESP:
 - EIP:
 - EBP:
 - ESP:

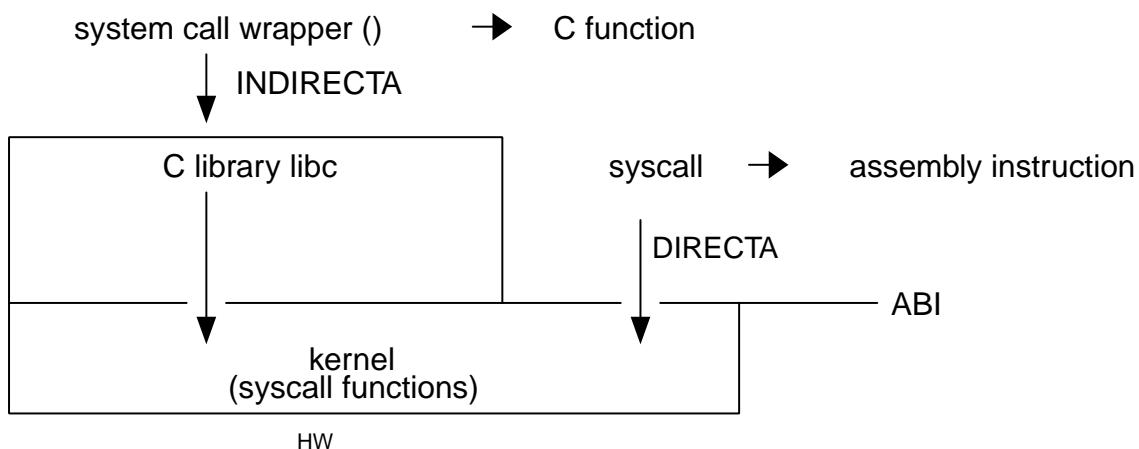
Posterior al salto de retorno

- 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 correspondiente a *main*.
- Contenido de los registros EIP,EBP,ESP:
 - EIP:
 - EBP:
 - ESP:

6.11. Llamadas al Sistema Operativo

6.11.1. 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

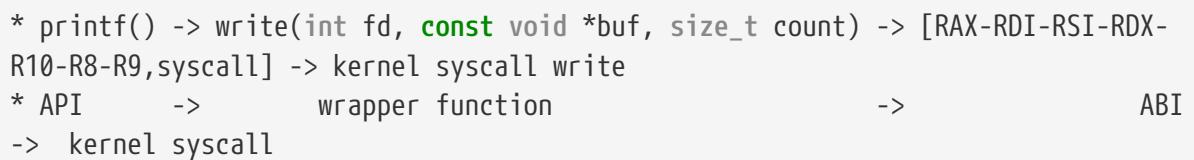
- Los 6 primeros argumentos se pasan a través de los registros: [EBX-ECX-EDX-ESI-EDI-EBP] previamente a la instrucción de la llamada [int 0x80](#)

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



- 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`



6.11.2. Ejemplos

- Ver en el [Apéndice](#)

6.12. Bibliografías

- Listado [Programación Ensamblador](#).
 - Apuntes completos [WikiBook](#) de programación en lenguaje AT&T con diversidad de aspectos.

VIII Apéndices

Chapter 7. Programas ensamblador IASSim

7.1. Ejemplo 1: sum1toN.ias

7.1.1. Lenguaje ensamblador iassim

- **2^a Versión:** módulo fuente *sum1toN_v2.ias*:

- La 2^a versión implementa un bucle cuyo cuerpo realiza una suma parcial donde uno de los sumandos varía en cada iteración.

```
;::::::::::::::::::; CABECERA
; ; 2a version : sum1toN_v2.ias
;::::::::::::::::::; SECCION DE INSTRUCCIONES
;Arquitectura orientada a Acumulador (AC)
;Registros accesibles : AC
; algoritmo: Bucle que realiza la operación suma = n + suma si n>=0
; inicio bucle
bucle: S(x)->Ac+ n      ; AC <- M[n] .Cargar sumando
        Cc->S(x)    sumar   ; si AC >= 0, PC <- sumar .Si el sumando < 0 fin del
bucle
        halt          ; stop
        .empty        ; un 20-bit 0, para que el nº de instrucciones sea par.
        ; realizar la suma
sumar: S(x)->Ah+ suma ; AC <- AC + M[suma]
        At->S(x)    suma ; M[suma] <- AC

;::::::::::::::::::;SECCION DE DATOS
; Declaracion de etiquetas, reserva de memoria externa, inicializacion.
; Variables ordinarias
n:  .data  5 ; variable sumando
uno: .data  1 ; cte
suma: .data  0 ; sumas parciales y resultado final
```

- Se ha desarrollado la sección de datos.
 - Se ha realizado un BUCLE con la operación SUMA en el cuerpo del bucle y con PARADA al salirse del bucle.
 - Se ha COMENTADO el código
- **Versión Demo** *tutorial.ias*: la versión demo que se incluye en el archivo de descarga del simulador.

```
loop:  S(x)->Ac+ n      ;load n into AC
        Cc->S(x)    pos  ;if AC >= 0, jump to pos
        halt          ;otherwise done
        .empty        ;a 20-bit 0
pos:   S(x)->Ah+ sum   ;add n to the sum
```

```

At->S(x)    sum ;put total back at sum
S(x)->Ac+  n ;load n into AC
S(x)->Ah-  one ;decrement n
At->S(x)    n ;store decremented n
Cu->S(x)    loop ;go back and do it again
n:     .data 5 ;will loop 6 times total
one:   .data 1 ;constant for decrementing n
sum:   .data 0 ;where the running/final total is kept

```

- Ejemplo con la Versión Demo *tutorial.ias*:

- cambiar el nombre del módulo fuente: *sum1toN.ias*
- reeditar el programa con etiquetas y comentarios en castellano.
- comentar el código con la información de los módulos descritos en las fases previas del desarrollo del programa

```

;;;;;;;;;; CABECERA
; Modulo fuente sum1toN.ias
; Calcula la suma de una secuencia de numeros enteros: suma = 1+2+...+n
; dato de entrada : N y dato de salida : suma
; Algoritmo : bucle de N iteraciones
;           Los sumandos se van generando en sentido descendente de n a -1
;           Si el sumando es negativo -> -1 , no se realiza la suma y finaliza
el bucle
; Estructuras de datos : variables n y suma . Constante uno.
; Lenguaje ensamblador: IASSim
; ISA: Arquitectura de la maquina IAS de Von Neumann

;;;;;;;;;; SECCION DE INSTRUCCIONES
;Arquitectura orientada a Acumulador (AC)
;Registros accesibles : AC
; algoritmo: Bucle que genera los sumandos n, n-1, .... -1
;           y realiza la operación suma = n + suma si n>=0

; inicio bucle : suma y generacion de sumandos
bucle: S(x)->Ac+ n ;cargar sumando
      Cc->S(x)  sumar ;si el sumando < 0 fin del bucle
; fin del bucle
      halt          ; stop
      .empty        ;a 20-bit 0 para que el nº de instrucciones sea par.
; realizar la suma
sumar: S(x)->Ah+ sum ;
       At->S(x)  sum ;
; actualizar sumando
       S(x)->Ac+ n ;
       S(x)->Ah- uno ;
       At->S(x)  n ;
; siguiente iteracion
Cu->S(x)  bucle ;

```

```
;;;;;;;;;;;;;;SECCION DE DATOS
; Declaracion de etiquetas, reserva de memoria externa, inicializacion.
; Variables ordinarias
n:    .data 5 ; sumando e inicializacion
sum:   .data 0 ; suma parcial y final
; constantes
uno:   .data 1 ;
```

7.1.2. Ejemplo 2: Producto/Cociente

Enunciado

- Desarrollar el programa que realice la operación $N(N + 1)/2$ equivalente a obtener el resultado de la suma del Tutorial1 $\sum_{i=1}^N i = N(N + 1)/2$.
 - Pseudocódigo del algoritmo
 - Organigrama del algoritmo
 - Programa en lenguaje RTL → Comentar apropiadamente el programa (cabecera con metainformación, secciones estructurales, bloques funcionales).
 - Programa en lenguaje Ensamblador
 - Ejecutar el programa paso a paso analizando el valor de los registros al ejecutar la multiplicación y la división.
- A TENER EN CUENTA en la descripción RTL:
 - El producto de dos números de M dígitos da como resultado un número de 2M dígitos, es decir, el doble que los multiplicandos. Esto dificulta las operaciones aritméticas posteriores a la multiplicación en la expresión matemática. Por ello dejaremos la operación multiplicación para el final dando prioridad a la suma y a la división
 - $N(N + 1)/2 = ((N + 1)/2) * N$
 - La división puede tener resto 1 ó 0 dependiendo de si el dividendo es par o impar
 - Si N es impar → $(N+1)/2$ tiene el resto 0 → $((N + 1)/2) * N$ donde $N+1$ es par
 - Si N es par → $(N+1)/2$ tiene el resto 1 → $(N+1) = \text{Cociente}*2+\text{Resto}$ → $N(N + 1)/2 = N*C+N/2$ donde N es par
 - La división por una potencia de 2 como 2^1 se realiza mediante una operación lógica: desplazar 1 bit a la izda el dividendo. El número de bits a desplazar es el valor del exponente.
 - descripción RTL **AC ← AC<<1**

Pseudocódigo

- Descripción del algoritmo mediante expresiones modo texto en lenguaje NATURAL
- VARIABLES:

- variable suma : almacena los resultados parciales y final
- variable N : almacena el dato de entrada
- Estructura del CODIGO imperativo:
 - La construcción de instrucciones básica es una asignación
 - $suma = N(N + 1) / 2$

Organigramas: Alto Nivel y RTL

- Descripción gráfica del algoritmo:
 - Alto Nivel: lenguaje natural imperativo
 - RTL: lenguaje de bajo nivel que tiene en cuenta la ISA de la computadora

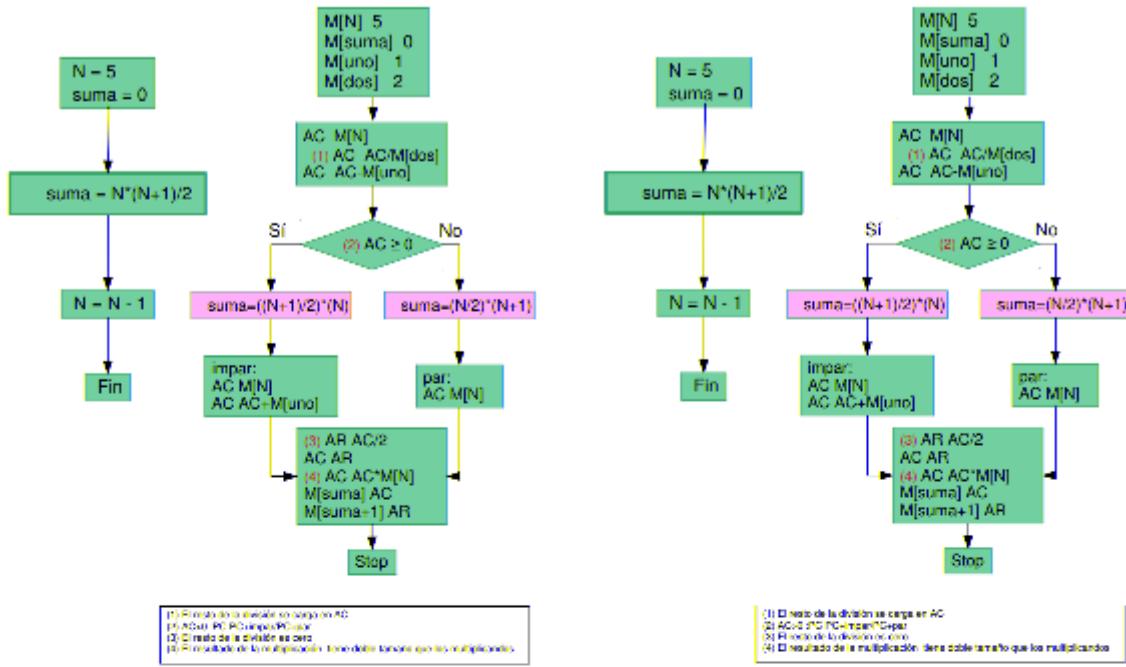


Figure 35. Diagrama de Flujo: Pseudocódigo y RTL

Lenguaje Ensamblador IAS

- sum1toN_mul.ias

; Suma de los primeros N numeros enteros. Y=N(N+1)/2

```

; CPU IAS
; lenguaje ensamblador: simaulador IASSim
; Ejercicio 2.1 del libro de William Stalling, Estructura de Computadores

; SECCION DE INSTRUCCIONES
; ¿Es N par? -> Resto de N/2
S(x)->Ac+ n    ;01 n    ;AC      <- M[n]
.

.

;

; Caso 1º: N par
.

.

;

; Caso 2º: N impar
.

.

;

; Multiplicación N(N+1)/2
.

.

;

; SECCION DE DATOS
; Declaracion e inicializacion de variables
y:    .data 0 ;resultado

; Declaracion de las Constantes
n:    .data 5 ;parametro N
uno:  .data 1
dos:  .data 2

```

simulación

- simulación con el emulador IAStim

7.1.3. Ejemplo 3: Vectores

Enunciado

- Realizar la suma $C = A + B$ de dos vectores A y B de 10 elementos cada uno inicializados ambos con los valores del 1 al 10.

 Para acceder a cada elemento de un vector es necesario ir incrementando la dirección absoluta de memoria del operando en la instrucción que accede a los elementos del vector, por lo tanto, es necesario modificar el campo de operando de la instrucción. Hay una instrucción de transferencia de los 12 bits del campo de operando a los 12 bits de menor peso del registro AC , es decir, $AC(28:39) \leftarrow M[\text{operando}](8:19)$. Y otra instrucción que realiza la transferencia inversa $M[\text{operando}](8:19) \leftarrow AC(28:39)$. De esta manera se pueden realizar operaciones de aritméticas y lógicas sobre los 12 bits del campo de operando de una instrucción.

- Pseudocódigo del algoritmo

- Organigrama del algoritmo
- Programa en lenguaje RTL → Comentar apropiadamente el programa (cabecera con metainformación, secciones estructurales, bloques funcionales).
- Programa en lenguaje Ensamblador: Se aconseja no realizar el programa directamente en su totalidad sino por fases, comenzando por una versión sencilla e ir avanzando hasta completar el programa en la versión final. Por ejemplo:
 - 1^a versión : Inicializar el vector $A[i]=i$
 - 2^a versión : Inicializar los vectores $A[i]=i$, $B[i]=i$, $C[i]=i$
 - 3^a versión : $C[i]=A[i]+B[i]$
 - Posibles variables : len: longitud del vector, A0: dirección del primer elemento del Vector A, i: índice del vector, etc.
- Ejecutar el programa paso a paso depurando las distintas versiones del programa.

Pseudocódigo

- Descripción del algoritmo mediante expresiones modo texto en lenguaje NATURAL
- VARIABLES:
 - variables vector A,B,C : Declararlas e inicializarlas $A[i]=i$, $B[i]=i$, $C[i]=0$
 - variable len : almacena el tamaño de los vectores
 - variable A0 : almacena la dirección del primer elemento de vector A
 - variable i : índice al elemento de posición i de cualquier vector
- Estructura del CODIGO imperativo:
 - La construcción de instrucciones básica es un bucle
 - El bucle cuenta las iteraciones en sentido descendente
 - Se inicializa el índice "i"= $len-1$ y
 - En cada iteración se asigna $A[i]=i$, $B[i]=i$, $C[i]=A[i]+B[i]$
 - En cada iteración se actualiza el índice $i=i-1$
 - Se sale del bucle cuando $i=-1$

Organigramas (1^a versión): Alto Nivel y RTL

- Descripción gráfica del algoritmo:
 - Alto Nivel: lenguaje natural imperativo
 - RTL: lenguaje de bajo nivel que tiene en cuenta la ISA de la computadora

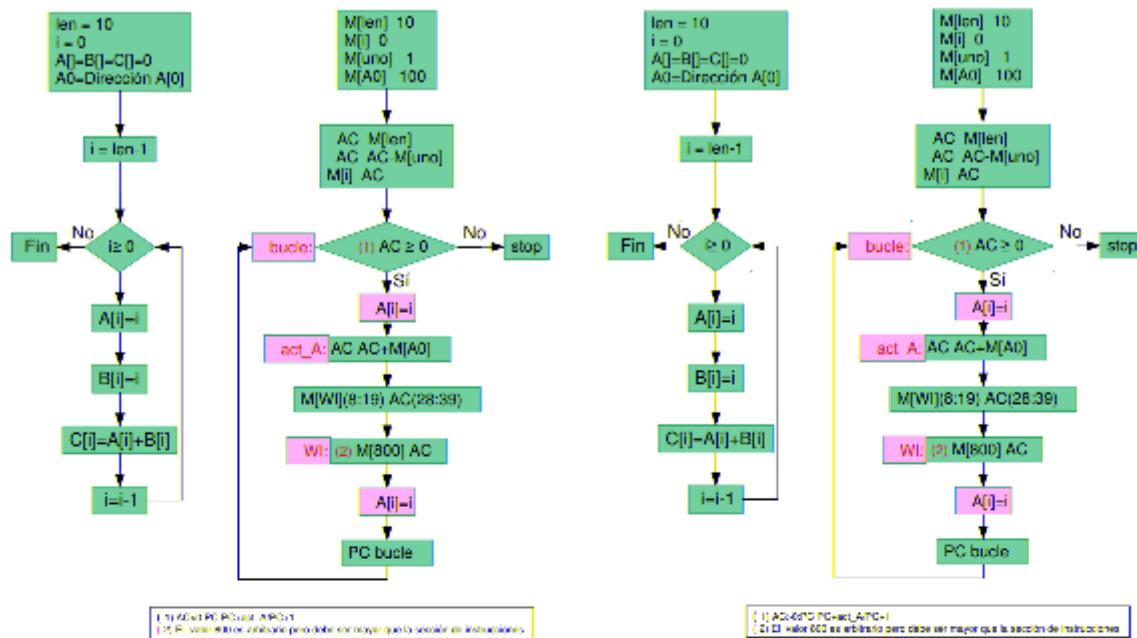


Figure 36. Diagrama de Flujo: Pseudocódigo y RTL

Organigrama (2^a versión): RTL

- Una posibilidad: inicializar los 3 vectores A[], B[] y C[]

Organigrama (3^a versión): RTL

- Versión definitiva: El vector C[] = A[]+B[]

Lenguaje Ensamblador IAS (1^a versión)

- vector_iniciar_A.ias

```
; vector_iniciar_A.ias
; Inicializar el vector A
; A es un vector de tamano "len" que esta almacenados en secuencia. La direccion
del primer elemento de A se guardo en la variable A0
; inicializamos el vector A[i]=i
; El acceso a los elementos del array se realiza escribiendo en el campo de
direcciones de la instruccion de lectura/escritura.
; Unicamente puden tener etiquetas las instrucciones de las izquierda por lo que
habrá que utilizar las instrucciones Cu'->S(x) etiqueta [Salto a la instrucción
derecha en la posición etiqueta] y Cu'->S(x) etiqueta [Salto a la instrucción
izquierda en la posición etiqueta] para ALINEAR todas las etiquetas en
instrucciones izquierda.
; Es necesario saber si las instrucciones de las direcciones bucle,suma y C estan a
la izda o derecha de la palabra de memoria.
; El numero de instrucciones ha de ser par. Utilizar .empty en caso impar.
; sin acentos en los comentarios
; Help online : manual de referencia -> tipos de datos
; View -> Preferences -> Capacidad de Memoria Selectron
```

;;;;;;;;;;SECCION DE INSTRUCCIONES

```
;;;;;; Inicializo indice i = len - 1
dcha1: Cu'->S(x) dcha1 ; salta a la dcha de dcha1
      S(x)->Ac+ len ;
      S(x)->Ah- uno ;
      At->S(x) i ;

;;;;;; inicio while : condicion elemento > 0
bucle: Cc->S(x) actu_A ;si AC >= 0, salto a Actu_A
      Cu->S(x) fin

;;;;;; Actualizo vector A[i]=i
; actualizo el puntero a A[i]
actu_A: S(x)->Ac+ cero ;
      S(x)->Ah+ A0 ;inicializo puntero con A[0]
      S(x)->Ah+ i ;inicializo puntero con A[0]+i
      Ap->S(x) wa ;Actualizo campo de direcciones de la instruccion IZDA
localizada en "wa" M[wa](8:19) <- AC(28:39)
; actualizo A[i]=i
      S(x)->Ac+ i ;
      Cu->S(x) wa ; salta a la izda de wa
wa: At->S(x) 100 ;M[100]<-AC. La direccion 100 cambia en tiempo de
```

ejecucion.

```
;;;;;;;; Siguiente iteracion
S(x)->Ac+ i      ;
S(x)->Ah- uno      ;
At->S(x) i      ;
Cc->S(x) bucle
.empty
fin:   halt
.empty

;;;;;;;;;;SECCION DE DATOS

;;;;;; variables ordinarias
len:       .data 10    ; longitud vectores A[], B[] y C[]
A0:       .data 30    ; direccion A[0]
i:        .data 0     ; indice del array

;;;;;; constantes
uno:       .data 1     ;
cero:      .data 0
```

Simulación (1^a versión)

- Realizar la ejecución del código binario con el emulador IASSim

Lenguaje Ensamblador IAS (2^a versión)

- Desarrollar el código fuente del programa vector_iniciar_A_B_C.ias:

Simulación (2^a versión)

- Realizar la ejecución del código binario con el emulador IASSim

Lenguaje Ensamblador IAS (3^a versión)

- Desarrollar el código fuente del programa vectorA+B.ias:

Simulación (3^a versión)

- Realizar la ejecución del código binario con el emulador IASSim

Chapter 8. Simulador IASSim

8.1. Máquina Virtual Java JVM

- Instalar el Kit de Desarrollo Java ([Java Development Kit-JDK](http://openjdk.java.net/) [<http://openjdk.java.net/>]) en el sistema ubuntu
 - [openjdk-11-jdk](https://packages.ubuntu.com/bionic/openjdk-11-jdk) [<https://packages.ubuntu.com/bionic/openjdk-11-jdk>] en la distribución linux/GNU ubuntu 18.0 bionic.
 - Comprobar que se tiene acceso al paquete: `apt-cache search openjdk-11-jdk`
 - Instalar el paquete: `sudo apt-get install openjdk-11-jdk`
 - Comprobar que está instalado el paquete: `dpkg -l openjdk-11-jdk`
 - Comprobar la versión de java instalada: `java --version`
- datos de la instalación en Ubuntu 17

```
Date: September 15, 2017.  
Emulator version: IASSim2.0.4  
Emulator command: java -cp IASSim2.0.4.jar:jhall.jar:IASSimHelp2.0.jar iassim.Main  
-m IAS.cpu  
Operating System: GNU/linux  
    Distributor ID: Ubuntu  
    Description:    Ubuntu 17.04  
    Release:       17.04  
    Codename:      zesty  
Java version: openjdk version "1.8.0_131"  
    OpenJDK Runtime Environment (build 1.8.0_131-8u131-b11-  
2ubuntu1.17.04.3-b11)  
    OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode)
```

IASSim se ejecuta en la máquina virtual de Java JVM, por lo tanto es un requisito tener instalada la máquina virtual JVM. Virtualizar una máquina consiste en instalar una capa SW por encima de cualquier Sistema Operativo (Linux, MacOS, Windows) de tal forma que cualquier aplicación (Por ejemplo IASSim) que se instale sobre la capa de virtualización no depende del Sistema Operativo y así se consigue independizar la aplicación (Por ejemplo IASSim) de los diferentes Sistemas Operativos.

8.2. Simulador IAS

- IASSim : Herramienta de simulación de la computadora IAS de Von Neumann útil para la simulación de la ejecución paso a paso de las instrucciones de un programa en código máquina. Permite visualizar el contenido de la memoria principal Selectron y de los registros de la CPU al finalizar cada ciclo de instrucción.
- [IASSim Web](http://www.cs.colby.edu/djskrien/IASSim/) [<http://www.cs.colby.edu/djskrien/IASSim/>] : Al hacer click nos conectamos al repositorio

del simulador IASSim.

- Descargar el Simulador IASSim2.0.4 : archivo zip
- Descomprimir el archivo IASSim2.0.4.zip.
- Abrir el Simulador mediante el comando:
 1. `./IASSim2.0.4$ java -cp IASSim2.0.4.jar:jhall.jar:IASSimHelp2.0.jar iassim.Main -m IAS.cpu`
 - En Windows se puede hacer doble click sobre el archivo por lotes con extensión `.bat`.

8.3. Simulación/Depuración

- Los objetivos de la simulación son dos:
 - a. Interpretar la ejecución de cada instrucción observando como varía la memoria y los registros
 - b. Depurar posibles errores en el desarrollo del programa.
- <https://www.linuxvoice.com/john-von-neumann/>
- Es necesario conocer la codificación hexadecimal de los números enteros y su conversión a código binario.
- Al programa **Demo tutorial.ias** que viene con el propio emulador le llamaremos *sum1toN.ias*
 1. El archivo zip descargado ha debido de ser descomprimido: observar los archivos extraídos, uno de ellos son las instrucciones de apertura del emulador.
 2. Abrir el emulador:
 - En linux mediante el comando en línea: `java -cp IASSim2.0.4.jar:jhall.jar:IASSimHelp2.0.jar iassim.Main -m IAS.cpu`
 - En windows: Doble click en el archivo por lotes con la extensión `*.bat`
 3. Ayuda: `Help → General IASSim Help → Assembly Language → Sintaxis y Regular Instructions` : manual del lenguaje ensamblador
 4. Borrar el contenido de la memoria tanto interna como externa. `Execute → Clear all`
 5. Desactivar el modo depuración : `Execute → Debug Mode NO seleccionado`
 6. Cargar el programa *sum1toN.ias* en lenguaje ensamblador : `File → Open → sum1toN.ias`
 - Lenguaje ensamblador: creado por los autores de la aplicación *IASSim*.
 7. Ventana RAM Selectrons: direcciones y contenido en código hexadecimal,decimal,binario... Anchura memoria : 20 ó 40 bits.
 8. Seleccionar la ventana con el código fuente en lenguaje ensamblador.
 9. Ensamblar y Cargar el módulo ejecutable en memoria : `Execute → Assemble & Load`
 10. Analizar el mapa de memoria : sección de instrucciones y sección de datos
 11. Activar el modo depuración : `Execute → Debug Mode`
 12. Ejecución de cada instrucción paso a paso : `Step by Step`

- Contenido de la Memoria

- La primera instrucción está almacenada en los 20 bits de la izda de la posición de memoria y la segunda instrucción en la dcha.

Address	Data	Comments
0	01 005	loop: S(x)->Ac+ n ;load n into AC
0	0F 002	Cc->S(x) pos ;if AC >= 0, jump to pos
1	00 000	halt ;otherwise done
1	00 000	.empty ;a 20-bit 0
2	05 007	pos: S(x)->Ah+ sum ;add n to the sum
2	11 007	At->S(x) sum ;put total back at sum
3	01 005	S(x)->Ac+ n ;load n into AC
3	06 006	S(x)->Ah- one ;decrement n
4	11 005	At->S(x) n ;store decremented n
4	0D 000	Cu->S(x) loop ;go back and do it again
5	00 000	n: .data 5 ;will loop 6 times total
5	00 005	
6	00 000	one: .data 1 ;constant for decrementing n
6	00 001	
7	00 000	sum: .data 0 ;where the running/final total is kept
7	00 000	
8	00 000	
8	00 000	

Figure 37. IAS Código Maquina

- Contenido de los Registros:

Registers		
Base: Hexadecimal		
name	width	value
Accumulator (AC)	40	00 0000 0000
Arithmetic Register (AR)	40	00 0000 0000
Control Counter (CC)	12	000
Control Register (CR)	20	0 0000
Function Table Register (FR)	8	00
Memory Address Register (MAR)	12	000
Selectron Register (SR)	40	00 0000 0000

Figure 38. IAS Registros

- Ejercicio:

- Antes de la ejecución de cada instrucción interpretarla: interpretar la instrucción en lenguaje máquina.
- prever el nuevo contenido de la sección de datos de la memoria
- prever el nuevo contenido de los registros de la CPU.
- prever la próxima instrucción a ejecutar
- Deducir el organigrama del programa.

Chapter 9. Lenguajes Ensamblador

9.1. Intel x86 / AMD 64

9.1.1. Hola Mundo

- Cada Arquitectura de Computador posee su propio lenguaje ensamblador.
- Módulo Fuente hola_mundo.s en lenguaje ensamblador.
 - x86-64

```
### -----
###      hola_x86-64_att.s
###
###      Programa simple de iniciación para el desarrollo de programas en
###      Ensamblador x86-64 AT&T.
###
###      Ficheros complementarios: macros_x86-64_gas.h
###
###
###      Compilación:
###          assemble using: as hola_intel_gas.s -o hola_intel_gas.o
###          link using:     ld hola_intel_gas.o -o hola_intel_gas
###          Driver gcc:    gcc -nostartfiles hola_intel_gas.s -o
hola_intel_gas
###
###      revised on: FEBRERO 2015 -- for Linux x86_64 environment
###
### -----
.att_syntax

## Incluir el fichero con las Macros
.include "macros_x86-64_gas.h"

## Declaración de símbolos externos
.global _start      # visible entry-point


## Reserva de Memoria para datos variables
.section .data

msg0:  .ascii "Hola Mundo\n"
len0:   .quad  . - msg0      #tamaño en bytes de la cadena msg0


## Sección para el Código de las Instrucciones en Lenguaje Ensamblador
.section .text


_start:
```

```

## Prompt del programa: imprimir mensaje

## Llamada al kernel para que acceda a la pantalla e imprima.
mov    $SYS_WRITE, %rax      # service ID-number
mov    $STDOUT_ID, %rdi      # device ID-number
mov    $msg0, %rsi       # message address
mov    len0, %rdx      # message length
syscall

## terminate this program
mov    $SYS_EXIT, %eax      # service ID-number
mov    $0, %rdi        # setup exit-code
syscall                 # enter the kernel

.end                      # no more to assemble

```

Macros en el fichero macros_x86-64_gas.h

```

## Llamadas al Sistema
.equ   STDIN_ID,  0      # input-device (keyboard)
.equ   STDOUT_ID, 1      # output-device (screen)
.equ   SYS_READ,  0      # ID-number for 'read'
.equ   SYS_WRITE, 1      # ID-number for 'write'
.equ   SYS_OPEN,  2      # ID-number for 'open'
.equ   SYS_CLOSE, 3      # ID-number for 'close'
.equ   SYS_EXIT, 60      # ID-number for 'exit'

```

9.1.2. Programación ensamblador

- [Felix Cloutier](http://www.felixcloutier.com/x86/) [<http://www.felixcloutier.com/x86/>]
- [kluge](http://kluge.in-chemnitz.de/docs/notes/assembly.php) [<http://kluge.in-chemnitz.de/docs/notes/assembly.php>]
- [IA64](http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html) [<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>]
- [AMD64](http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/) [<http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/>]
 - [AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions](http://developer.amd.com/wordpress/media/2008/10/24594_APM_v3.pdf) [http://developer.amd.com/wordpress/media/2008/10/24594_APM_v3.pdf]

9.1.3. Números Reales

- [kluge](http://kluge.in-chemnitz.de/docs/notes/assembly.php) [<http://kluge.in-chemnitz.de/docs/notes/assembly.php>]
 - interesantes los ejemplos de operaciones con números reales

9.1.4. Discusión por qué ASM AT&T

- <http://es.tldp.org/Presentaciones/200002hispalinux/conf-28/28.ps.gz>

9.1.5. Miscellaneous

Tipos de Datos

- Tipos de Datos:
 - Dirección de memoria o referencia a memoria: etiqueta longitud
 - `mov $longitud,%edx` → `mov 0x8049fff,%edx` → en lenguaje de alto nivel es la inicialización de un puntero
 - número entero con signo : formato complemento a 2.
 - `mov $0x4,%eax`
 - El operando 0x4 está localizado en la propia instrucción, en el campo de operando. El dato 0x4 se almacena en "little endian" → Campo de operando: double word: 32 bits 0x00000004 → En memoria ascendente : dirección 8048191: 04 00 00 00
 - carácter: codificación ASCII
 - `08049ff4 <mensaje>`: 48 6f 6c 61 20 → H o l a SP

Ciclo de Instrucción

- Intervención de la CPU en la instrucción **4001a4: 48 83 ec 10 → subq \$16,%rsp**
 - La CPU durante el ciclo de instrucción (fase captura- fase decodificación-fase ejecución) realiza una secuencia de tareas.
 - La secuencia de tareas a realizar la CPU durante el ciclo de instrucción lo describimos en lenguaje RTL.
 - MBR ← M[0x4001a4]
 - IR ← MBR
 - AC ← RSP
 - AC ← AC-16 ; (ALU resta)
 - RSP ← AC
 - PC ← PC+1
 - MAR ← PC

hola_mundo.s

- Compilar el programa en lenguaje ensamblador *hola_mundo.s* y volcar el módulo objeto binario.
 - Módulo fuente: *hola_mundo.s*.
 - Código Máquina- Código Ensamblador

- Sección Datos

```
08049ff4 <mensaje>: 48 6f 6c 61 20 4d 75 63 64 6f 0a      H o l a SP m u n  
d o /n  
08049fff <longitud>: 0b 00
```

- En un lenguaje de alto nivel sería la declaración e inicialización de variables.
- Etiqueta: referencia a memoria
- Cada carácter ocupa un byte (codificación ASCII). No interpretar el string como un todo (no little endian) a diferencia de los números enteros y reales.
- El dato referenciado por la etiqueta longitud está en formato *little endian* → 00 0b

- Sección Instrucciones

```
08048190 <_start>:  
08048190: b8 04 00 00 00      mov    $0x4,%eax  
08048195: bb 01 00 00 00      mov    $0x1,%ebx  
080481a0: b9 f4 9f 04 08      mov    $0x8049ff4,%ecx  
080481f0: 8b 15 ff 9f 04 08      mov    0x8049fff,%edx  
080481a5: cd 80              int    $0x80  
080481a7: b8 01 00 00 00      mov    $0x1,%eax  
080481ac: bb 00 00 00 00      mov    $0x0,%ebx  
080481b1: cd 80              int    $0x80
```

9.2. Motorola 68000

9.2.1. Hola Mundo

```
;CISC Sharp X68000 (Human68K): Motorola 68000  
pea (string)      ; push string address onto stack  
dc.w $FF09        ; call DOS "print" by triggering an exception  
addq.l #4,a7      ; restore the stack pointer  
  
dc.w $FF00        ; call DOS "exit"  
  
string:  
dc.b "Hello, world!",13,10,0
```

9.2.2. ISA

- Referencias

- [Instruction Set Basic](http://lux.dmcsl.pl/pn/assembler_68000/asm.html) [http://lux.dmcsl.pl/pn/assembler_68000/asm.html]
- [Wikibook](https://en.wikibooks.org/wiki/68000_Assembly#Indirect_addressing_with_postincrement) [https://en.wikibooks.org/wiki/68000_Assembly#Indirect_addressing_with_postincrement]
- [Manual de Referencia](https://www.nxp.com/files-static/archives/doc/ref_manual/M68000PRM.pdf) [https://www.nxp.com/files-static/archives/doc/ref_manual/M68000PRM.pdf]

- **Motorola 68K ó M68000** [http://www.freescale.com/files/archives/doc/ref_manual/M68000PRM.pdf]
- m68k hasta 1991
- ppc (powerpc) desde 1991 con Apple e IBM → iMac (1996-2006)
- arquitectura general

2 versiones: Procesador de 16 bits ó 32 bits
 Aprox . 90 instrucciones máquina
 12 modos de direccionamiento
 9 formatos de instrucción distintos y con tamaños de una a cinco palabras
 Ancho del bus de datos: 16 bits ó 32 bits
 Tamaño mínimo direccionable : 1 byte
 Ancho del bus de direcciones: 24 bits (2^{24} bytes = 16 Mbytes de memoria direccionables)

- Registros:
 - 8 Registros de Datos de propósito general (16/32): D0-D7
 - 7 Registros de Instrucciones de propósito general (16/32) :A0-A6
- modos de direccionamiento
 - #: inmediato
 - Di : registro directo
 - (Ai): indirecto de registro
 - +(Ai): indirecto de registro con postincremento con la escala del tamaño del operando (1,2,4 bytes)
 - (Ai)+: indirecto de registro con postincremento con la escala del tamaño del operando
 - -(Ai): indirecto de registro con predecremento con la escala del tamaño del operando
 - (Ai)-: indirecto de registro con preincremento con la escala del tamaño del operando
 - D(Ai): indirecto de registro con desplazamiento D
 - D(Ai,Ri,X) : registro Ai indirecto indexado Ri con desplazamiento D
 - D(PC) : relativo al PC con desplazamiento D
 - D(PC,Ri,X) : relativo al PC indexado Ri con desplazamiento D
- Datos
 - Enteros en Complemento a 2 .
 - Sufijos Operación: B byte (1 byte), W word (2 bytes) , L long (4 Byte)
 - Prejijos datos: \$ hexadecimal
- Memoria
 - Big Endian : LSB en la dirección más alta y MSB en la dirección más baja

9.3. MIPS

9.3.1. ISA

- Procesador con una arquitectura de 32 bits
- Microprocessor without Interlocked Pipeline Stages (MIPS) Architecture
- Versiones de la arquitectura MIPS:
 - MIPS I (R2000 cpu), II (R6000), III (R4000), IV (R8000, R5000, R10000), and V (nunca implementada);
 - MIPS32/64 :MIPS32 is based on MIPS II with some additional features from MIPS III, MIPS IV, and MIPS V; MIPS64 is based on MIPS V

70 instrucciones máquina
Instrucciones clasificadas en cuatro grupos
 Movimiento de datos
 Aritmética entera, lógicas y desplazamiento
 Control de flujo
 Aritmética en punto flotante
4 modos de direccionamiento
 Inmediato
 Directo de registros
 Indirecto con desplazamiento
 Indirecto con desplazamiento relativo al PC
Banco de 64 registros (32 bits cada uno)
 32 de propósito general (R0-R31)
 32 para instrucciones en punto flotante (F0-F31). Pueden usarse como:
 32 registros para operaciones en simple precisión (32 bits)
 16 registros para operaciones en doble precisión (64 bit)
3 formatos de instrucción distintos con longitud única de 32 bits:
 Op Code: 6 bits
 R :three registers, a shift amount field, and a function field;
 I :two registers and a 16-bit immediate value
 J :26-bit jump target
Arquitectura registro-registro
 Sólo las instrucciones de LOAD y STORE hacen referencia a memoria
 El resto de instrucciones operan sobre registros
 Instrucciones con tres operandos: 2 op.fuente y 1 op.Destino

Notación ensamblador: op x, y, z x<-(y)op(z)
Datos:
 Enteros Complemento a 2 : byte (1B), media palabra (2B), palabra (4B)
 Nº Reales: IEEE-754 simple y doble precisión

- [MIPS architecture](https://en.wikipedia.org/wiki/MIPS_architecture) [https://en.wikipedia.org/wiki/MIPS_architecture]
- [Versiones de la ISA MIPS](https://en.wikipedia.org/wiki/List_of_MIPS_architecture_processors) [https://en.wikipedia.org/wiki/List_of_MIPS_architecture_processors]
- [procesadores con arquitectura MIPS](https://en.wikipedia.org/wiki/MIPS_architecture_processors) [https://en.wikipedia.org/wiki/MIPS_architecture_processors]:

R2000, etc

- [quick tutorial](http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm) [http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm]
- [Emulador MIPS Online](https://rivoire.cs.sonomia.edu/cs351/wemips/) [https://rivoire.cs.sonomia.edu/cs351/wemips/]

9.4. ARM

9.4.1. Hola Mundo

```
/*
```

Programa en lenguaje ensamblador AT&T para el procesador ARM

Programa fuente: hello_world.s

Assembler: arm-linux-gnueabi-as -o hello_world.o hello_world.s
Linker: arm-linux-gnueabi-ld -o hello_world hello_world.o

```
*/
```

```
.data
```

```
msg:
```

```
.ascii "Hello, ARM World!\n"
```

```
len = . - msg
```

```
.text
```

```
.globl _start
```

```
_start:
```

```
/* write syscall */
```

```
mov %r0, $1
```

```
ldr %r1, =msg
```

```
ldr %r2, =len
```

```
mov %r7, $4
```

```
swi $0
```

```
/* exit syscall */
```

```
mov %r0, $0
```

```
mov %r7, $1
```

```
swi $0
```

9.4.2. ISA

- [ARM](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc_subset_architecture.reference/index.html) [http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc_subset_architecture.reference/index.html]: Advanced RISC Machine

◦ [Developer](#)

[Guides](#)

[http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc_subset_architecture.reference/index.html]

Chapter 10. Lenguajes de programación para sum1toN

10.1. Otros Lenguajes para sum1toN

- Refs
 - <http://wiki.c2.com/?ArraySumInManyProgrammingLanguages>
 - https://www.rosettacode.org/wiki/Sum_and_product_of_an_array#With_explicit_conversion
- Desarrollar el algoritmo sum1toN en : Lisp, Python, Java, C, Pascal, ...
- elisp

```
(setq array [1 2 3 4 5])
(apply '+ (append array nil))
(apply '* (append array nil))
```

- Phyton

```
>>> sum(range(5,0,-1))
```

- Java

```
/* Programa Fuente: sum1toN.java

compilación: javac sum1toN.java -> genera el BYTECODE sum1toN.class
ejecución -> java -cp . sum1toN ; necesita el bytecode *.class y ejecutará el
main de class

*/
public class sum1toN {
// método main encapsulado en la clase class, static para que main no pueda cambiar
los atributos, publico para ser accesible.
    public static void main(String[] args) {
        System.out.println("Suma de Números enteros");
        int x=5, suma=0;

        while (x >= 0 ) {
            System.out.print( x );
            System.out.print(",");
            suma=suma+x;
            x--;
        }
        System.out.print("\n");
        System.out.print("suma="+suma);
```

```

        System.out.print("\n");
    }
}

```

- C

```

/*
Programa:      sum1toN.c
Descripción:   realiza la suma de la serie 1,2,3,...N
                Es el programa en lenguaje C equivalente a sum1toN.ias de la
máquina IAS de von Neumann
Lenguaje:      C99
Descripción:   Suma de los primeros 5 números naturales
Entrada:       Definida en una variable
Salida:        Sin salida
Compilación:   gcc -m32 -g -o sum1toN sum1toN.c -> -g: módulo binario depurable
                -> -m: módulo binario
arquitectura x86-32 bits
S.O:           GNU/linux 4.10 ubuntu 17.04 x86-64
Librería:      /usr/lib/x86_64-linux-gnu/libc.so
CPU:           Intel(R) Core(TM) i5-6300U CPU @ 3.0GHz
Compilador:    gcc version 6.3
Ensamblador:   GNU assembler version 2.28
Linker/Loader:  GNU ld (GNU Binutils for Ubuntu) 2.28
Asignatura:    Estructura de Computadores
Fecha:         20/09/2017
Autor:         Cándido Aramburu
*/

```

```

#include <stdio.h> // cabecera de la librería de la función printf()

// función de entrada al programa
void main (void)
{
    // Declaración de variables locales
    char suma=0;
    char n=0b101;
    // bucle
    while(n>0){
        suma+=n;
        n--;
    }
    printf("\n La suma es = %d \n",suma);
}

```

- Lenguaje ensamblador INTEL y assembler nasm

```

;; Programa: sum1toN.asm
;; Descripción: realiza la suma de la serie 1,2,3,...N

```

```

;;; Lenguaje INTEL
;;; Assembler NASM

;;; nasm -hf -> ayuda de la opción f
;;; Ensamblaje nasm -g -f elf sum1toN.asm -o sum1toN.o
;;; linker -> ld -m elf_i386 -o sum1toN sum1toN.o

BITS 32 ; cpu MODE
;; Declaración de variables
section .data

n: dd 5 ; 4 bytes

global _start

;; Comienzo del código
section .text
_start:
    mov ecx,0 ; ECX implementa la variable suma
    mov edx,[n] ; EDX implementa es un alias de la variable n
bucle:
    add ecx,edx
    sub edx,1
    jnz bucle

    mov ebx, ecx ; el argumento de salida al S.O. a través de EBX según
convenio

    ; salida
    mov eax,1 ; código de la llamada al sistema operativo: subrutina exit
    int 0x80 ; llamada al sistema operativo

```

- Lenguaje ensamblador ATT para la arquitectura x86-32

```

### Programa: sum.s
### Descripción: realiza la suma de la serie 1,2,3,...N
### gcc -m32 -g -nostartfiles -o sum1toN sum1toN.s
### Ensamblaje as --32 --gstabs fuente.s -o objeto.o
### linker -> ld -melf_i386 -I/lib/i386-linux-gnu/ld-linux.so.2 -o ejecutable
objeto.o -lc

## Declaración de variables
.section .data

n: .int 5

.global _start

## Comienzo del código
.section .text

```

```

_start:
    mov $0,%ecx # ECX implementa la variable suma
    mov n,%edx
bucle:
    add %edx,%ecx
    sub $1,%edx
    jnz bucle

    mov %ecx, %ebx # el argumento de salida al S.O. a través de EBX según
convenio

## salida
    mov $1, %eax # código de la llamada al sistema operativo: subrutina exit
    int $0x80 # llamada al sistema operativo

.end

```

- Lenguaje ensamblador AT&T para la arquitectura x86-64

```

##### Programa: sum1toN.s
### Descripción: realiza la suma de la serie 1,2,3,...N. La entrada se define
en el propio programa y la salida se pasa al S.O.
### Lenguaje: Lenguaje ensamblador de GNU para la arquitectura AMD64
### gcc -no-pie -g -nostartfiles -o sum1toN sum1toN.s
### Ensamblaje as --gstabs sum1toN.s -o sum1toN.o
### linker -> ld -o sum1toN sum1toN.o
## Declaración de variables
## SECCION DE DATOS
.section .data

n:     .quad 5

.global _start

## Comienzo del código
## SECCION DE INSTRUCCIONES

.section .text
_start:
    movq $0,%rdi # RDI implementa la variable suma
    movq n,%rdx
bucle:
    add %rdx,%rdi
    sub $1,%rdx
    jnz bucle

## el argumento de salida al S.O. a través de RDI según convenio ABI AMD64
## salida
    mov $60, %rax # código de la llamada al sistema operativo: subrutina exit

```

syscall # llamada al sistema operativo para que ejecute la subrutina según el valor de RAX

.end

- ARM

```
/*
Programa: sum1toN.s
Descripción: realiza la suma de la serie 1,2,3,...N
Es el programa en lenguaje ARM equivalente a sum1toN.ias de la máquina IAS
de von Neumann
    gcc -g -nostartfiles -o sum1toN sum1toN.s
    Ensamblaje as --gstabs sum1toN.s -o sum1toN.o
    linker -> ld -o sum1toN sum1toN.o
*/
@ Declaración de variables
.section .data
n: .int 5

.global _start

@ Comienzo del código
.section .text
_start:
    mov r0,#0      @ R0 implementa la variable suma
    ldr r2,=n      @ R1 implementa la variable n indirectamente
    ldr r1,[r2]
/* Direccionamiento directo:
   mov r1,n da error porque mov no admite direccionamiento a memoria directo.
   mov admite direccionamiento inmediato si el literal de 32 bits no
tiene repetición de ceros a izda y dcha
   para convertirlo en un literal de 8 bits seguido de
desplazamientos
   ldr r1,n Error: reubicación_interna (tipo OFFSET_IMM) no compuesta
   Da error al intentar codificar un literal (dirección n) de 32
bits.
*/
bucle:
    add r0,r1
    subs r1,#1
    bne bucle

@r0 es el argumento de salida al S.O. a través de EBX según convenio

/* exit syscall */
mov r7, #1
swi #0
```

.end

Chapter 11. RTL Register Transfer Language

11.1. Lenguaje RTL

11.1.1. Introducción

- Lenguaje de descripción de INSTRUCCIONES: Register Transfer Language (RTL)
- El lenguaje RTL tiene como objetivo poder expresar las instrucciones que ejecuta la CPU como sumar(ADD),restar(SUB),mover(MOV), etc. La descripción se realiza a nivel de transferencia de datos entre *registros* internos de la CPU o entre registros internos y la memoria externa.
- El lenguaje RTL , mediante símbolos interpretables por el programador, permite describir su comportamiento a nivel hardware y así definir el diseño de la arquitectura de una máquina.
- Los *registros* son el elemento fundamental de memoria en la ruta de los datos e instrucciones entre las distintas unidades básicas de la computadora. Un registro es un circuito digital que almacena, memoriza, un dato.
- La ruta de los datos está formada por los buses y los elementos (*registros*, multiplexores, switches, contadores, etc) que se conectan a través de los buses
 - Ejemplo: ruta de un dato desde una posición de la memoria principal hacia los registros de operando de la ALU.
 - Concepto de buffer: etapa intermedia de memoria en la ruta de los datos.
- La ejecución de las instrucciones que ejecuta la CPU implica la transferencia de datos a través de los registros de la ruta de datos.



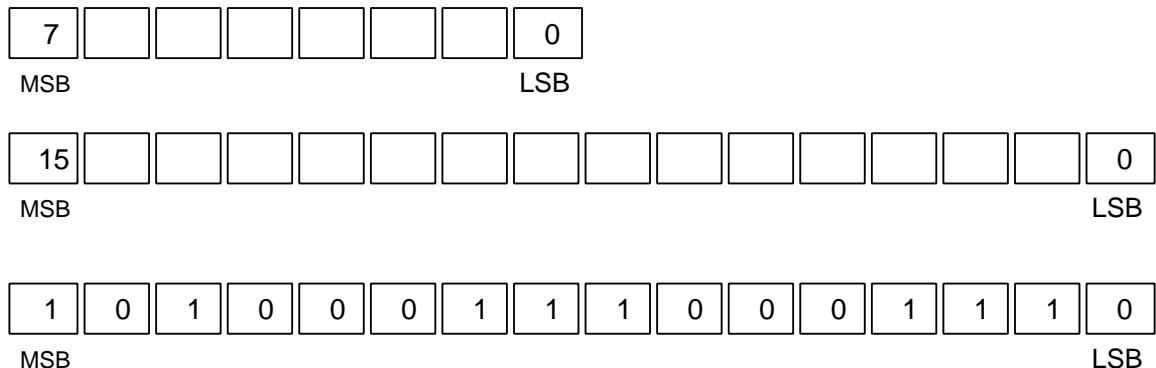
No confundir el RTL (Register Transfer Language) con el RTL (Register Transfer Level). El Register Transfer Level es utilizado por los lenguajes de descripción de HARDWARE (Hardware Description Language HDL)

11.1.2. Registros

Arquitectura

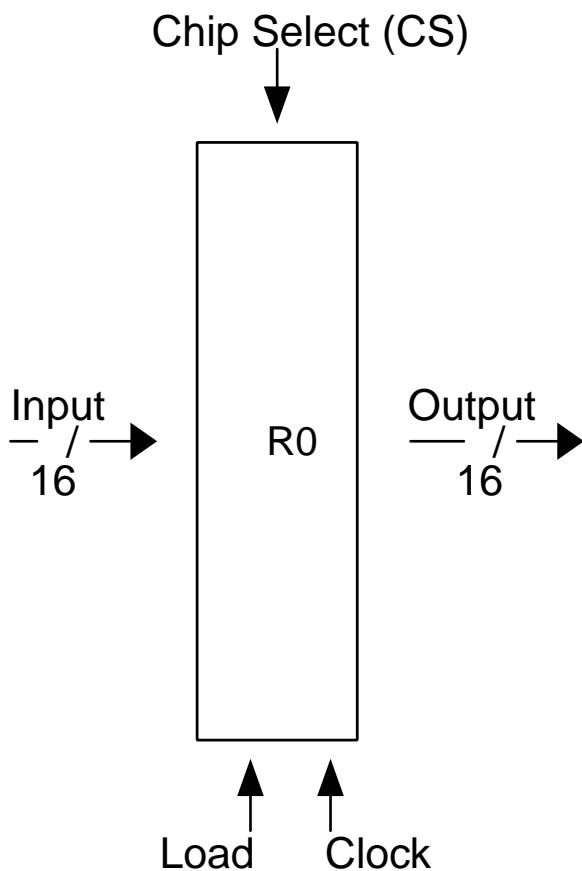
- La arquitectura de un registro comprende su funcionalidad y la estructura su implementación
- Los registros:
 - *almacenan* una palabra formada por una secuencia de bits.
 - son una array de celdas en una dimensión, donde cada celda almacena un bit.
- Su tamaño normalmente es un múltiplo de 8 bytes y recibe un nombre.
 - 8 bits: 1 Byte
 - 16 bits: Word. Por razones históricas.(recordad que el tamaño de una palabra en otro contexto depende de la máquina de que se trate)
 - 32 bits: double word
 - 64 bits: quad word

- Las celdas se enumeran empezando por cero.
 - LSB: Least Significant Bit es el bit de menor peso
 - MSB: Most Significant Bit es el bit de mayor peso

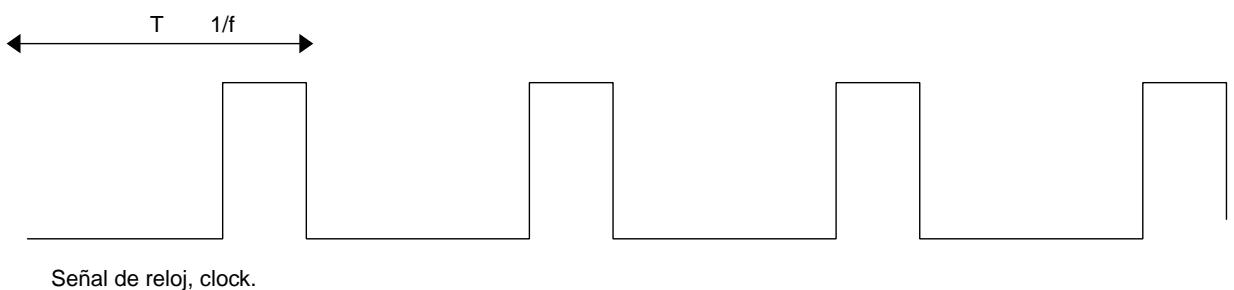


Estructura

- La estructura del registro es la implementación de su funcionalidad
- Cada bit del dato a registrar se almacena en una celda con capacidad de memoria. Las celdas de un registro se implementan con un circuito digital denominado flip-flop. Cada flip-flop almacena un bit.



- El registro está conectado al mundo exterior mediante buses: bus de entrada y bus de salida
- CS: Chip Select : conecta la salida interna del registro R0 al bus de salida → Operación de lectura del registro
- Load: Si la señal está activa se ordena la carga del valor del bus de entrada en el registro R0, se registra el dato de entrada. Operación de escritura en el registro.
- Clock: señal digital binaria periódica.
- La carga es síncrona con la señal de reloj clock CLK. El sincronismo se produce en los flancos positivos _|| o negativos



11.1.3. Símbolos

- Los nombres de los registros se expresan mediante mayúsculas

- PC: Program Counter
- IR: Instruction Register
- R2: Registro 2
- Secciones de un registro
 - PC(L) : Byte de menor peso del registro contador de programa
 - PC(H) : Byte de mayor peso del registro contador de programa
 - PC(7:0): Secuencia de bits de la posición cero hasta la posición séptima del registro contador de programa.

11.1.4. Sentencias RTL

Operaciones y Sentencias RTL

- En lenguaje RTL entendemos por sentencia una expresión que implica realizar operaciones con los registros.
- Operaciones RTL:
 - transferencias entre registros, suma del contenido de dos registros, invertir el contenido de un registro, etc

Microoperación

- MICROoperaciones: operaciones realizadas por el MICROprocesador internamente, al ejecutar una Instrucción Máquina.
 - Ejemplos: escribir en un registro, orden de lectura a la M.Principal, leer de un registro, Decodificar una instrucción, incrementar un contador, sumar (microordenes al circuito sumador), desplazamiento de los bits de un registro, lógica AND, etc...
- La operación de escribir en un registro o leer en un registro para la CPU es una microoperación.

Transferencia entre registros

- Operador transferencia \leftarrow
- Sentencia transferencia: $R2 \leftarrow R1$
 - A R1 se le llama registro fuente y a R2 registro destino
 - Copiamos el contenido del registro R1 en el registro R2

Sentencia Condicional

- If ($K1=1$) then $R2 \leftarrow R1$
 - $K1:R2 \leftarrow R1$
 - La transferencia o copia se realiza únicamente si K1 es verdad es decir K1 vale el valor lógico 1.

Sentencia Concurrente

- Operador coma
- $K3:R2 \leftarrow R1, R3 \leftarrow R1$
 - Si $K3$ es verdad el contenido de $R1$ se copia en $R2$ y $R3$

Referencia a la Memoria Principal

- Se utilizan los corchetes y el símbolo M .
- $M[0x80000]$: contenido de la posición de memoria $0x8000$
- $AC \leftarrow M[0x80000]$: copiar el contenido de memoria de la posición $0x8000$ al registro AC
- $AC \leftarrow M[AC]$: copiar el contenido de la posición de memoria a la que **apunta** el registro AC en el registro AC
- $M[0x8000] \leftarrow AC$: copiar el contenido del registro AC en la posición de Memoria $0x8000$
 - $M[0x8000] \leftarrow R[AC]$: copiar el contenido del registro AC en la posición de Memoria $0x8000$

Left-Right Value

- Este concepto se utiliza en el lenguaje C al definir la sentencia asignación $=$
- $M[0x1000] \leftarrow M[0x2000]$
 - El contenido de la posición $0x2000$ se copia en la posición $0x1000$
 - Lo que hay a la derecha del operador \leftarrow se evalua y se obtiene un VALOR
 - Lo que hay a la izda del operador \leftarrow es una DIRECCION o REFERENCIA a Memoria (Principal o Registro)

11.1.5. Ejemplos RTL con expresiones aritmético-lógicas

- $AC \leftarrow R1 \vee R2$
 - Operación lógica OR
- $(K1+K2):R1 \leftarrow R2+R3, R4 \leftarrow R5^R6$
 - El símbolo $+$ tiene dos significados: booleano o aritmético.
 - En $k1+k2$ tiene significado booleano: or. Aquí no tiene sentido la suma aritmética de señales lógicas. Tiene sentido evaluar si las señales están activas o no.
 - En $R2+R3$ tiene significado aritmético.
- Para indicar prioridad en una expresión utilizaremos los paréntesis.

Chapter 12. Formato de Instrucción: ISA Intel x86-64

12.1. Formato de Instrucción: ISA Intel x86-64

12.1.1. Ejemplo subq \$16,%rsp

- Ejemplo:
 - instrucción máquina intel x86.
 - **4001a4: 48 83 ec 10 → subq \$16,%rsp**
 - ¿Cómo interpretar la instrucción máquina **4883EC10**? Es necesario consultar el **Manual de Referencia de la Arquitectura ISA de la máquina x86** y tener conocimientos de los modos de direccionamiento.
 - [manual oficial de intel x86 ó x86-64](http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html) [<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>]: cuidado con la sintaxis intel.
 - consultar el volumen 2B (capítulo 4, pag 394) para la instrucción **SUB**. Hay que tener en cuenta el tamaño de los operandos y los modos de direccionamiento.
 - El sufijo q de la operación **SUBQ** indica operando de 64 bits. El operando fuente **\$16** es referenciado con direccionamiento inmediato y se puede codificar con 8 bits y el operando destino **%RSP** es un registro de 64 bits. Por lo tanto la descripción intel en el manual será **SUB r64, imm8** que se corresponde con el código de operación **REX.W + 83 /5 ib**.
 - La descripción del código de operación que hace intel no es sencilla y es necesario consultar la Interpretación de la Instrucción en el **volumen 3 3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES** y el Formato de Instrucción en el **volumen 2A (capítulo 2 Formato de Instrucción)**
 - Figura 2.1 Intel 64 and IA-32 Architectures Instruction Format
 - El formato de instrucción tiene los campos: **REXprefix-CodOp-ModRB** que en nuestro caso valen **48-83-EC**
 - interpretación del campo REXprefix: **REX.W**: Manual → El prefijo REX se utiliza para operandos de 64 bits bien inmediatos y/o registros GlobalPurposeRegister(rax,rbx, etc), 2.2.1.2 More on REX Prefix Fields
 - El primer byte es **48** → **01001000** donde el bit de la posición 3 está activado por lo que según la tabla "Table 2-4. REX Prefix Fields [BITS: 0100WRXB]" quiere decir que el **operando es de 64bits**
 - **/5** : the ModR/M byte of the instruction uses only the r/m (register or memory) operand. **En este caso register**. Ver el subcampo R/M más abajo.
 - **ib** : A 1-byte (ib) immediate operand.
 - Campo Primary Opcode: El segundo byte vale **83** → Operación de resta **SUB**
 - Campo ModRB: El tercer byte vale **EC** → **1110-1100** hace referencia al registro RSP.

- 2.1.3 ModR/M and SIB Bytes: Many instructions that refer to an operand in memory (memoria principal o registro interno CPU) have an addressing-form specifier byte (called the ModR/M ...). Este campo se divide en subcampos: **Mod-Reg/Opcode-R/M**
- Subcampo Mod: **11** : The mod field combines with the r/m field to form 32 possible values: eight registers (rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp) and 24 addressing modes
- Subcampo Reg/Opcode: En este caso no es Secondary Opcode sino que es Reg: rrr= **101**
- Subcampo R/M: En este caso R: bbb= **100** The r/m field can specify a register as an operand or it can be combined with the mod field to encode an addressing mode. **En este caso es el registro operando con código 4.** En la tabla 3.1 el código del **quad word register** con el **Reg Field** de valor 4 es el registro **RSP**
- Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used
→ Mod=11 → Rrrr =0101 → Bbbb=0100
- El cuarto byte vale en hexadecimal **10** que se corresponde con el valor inmediato 16 en decimal y debe ser expandido a 64bits.

12.1.2. Otros x86-32

- http://www.c-jump.com/CIS77/CPU/x86/lecture.html#X77_0010_real_encoding
 - ADD cl,al → 02C8
 - ADD EAX, [ESI + disp8] → 0346XX
- jump instructions [http://en.wikibooks.org/wiki/X86_Assembly/Control_Flow#Jump_Instructions]
- Hola mundo x86

```

08048190 <_start>:
08048190: b8 04 00 00 00      mov    $0x4,%eax
08048195: bb 01 00 00 00      mov    $0x1,%ebx
0804819a: b9 f4 9f 04 08      mov    $0x8049ff4,%ecx
0804819f: 8b 15 ff 9f 04 08    mov    0x8049fff,%edx
080481a5: cd 80              int    $0x80
080481a7: b8 01 00 00 00      mov    $0x1,%eax
080481ac: bb 00 00 00 00      mov    $0x0,%ebx
080481b1: cd 80              int    $0x80

```

- **b8 04 00 00 00** → mov \$0x4,%eax
 - Manual: *B8 + rd MOV reg32,imm32 2 Move immediate dword to register:*
 - Campo de Código de Operación: **B8**
 - Campo de operando: double word: 32 bits: **04 00 00 00** → little endian → es el dato 0x00000004

Chapter 13. FPU x87

13.1. FPU x87

13.1.1. Resumen

- Arquitectura x87:
 - 1980
 - es un repertorio de instrucciones que realiza operaciones matemáticas complejas con números reales como calcular la tangente,etc .
- x87 coprocessor o x87 FPU(Float Point Unit):
 - es un procesador independiente de la CPU x86 para ejecutar instrucciones de la arquitectura x87.
- The x87 registers:
 - Son registros internos a la FPU. 8-level deep non-strict *stack structure* ranging from ST(0) to ST(7). No son directamente accesibles, sino que se acceden con push, pop o desplazamiento relativo al top de la pila.
- FPU : es un componente de la unidad central de procesamiento especializado en el cálculo de operaciones en coma flotante de la misma manera que la ALU lo es con números enteros almacenados en los registros RPG.
- Formato de datos:
 - single precision, double precision and 80-bit double-extended precision binary floating-point arithmetic as per the IEEE 754
 - ó múltiples enteros en el mismo registro de 8,16 o 32 bits.
- FP: Float Point : Registros de la pila de la FPU, nueva denominación de los registros ST.
- MMX: Conjunto de instrucciones SIMD (Single Instruction Multiple Data) diseñado por Intel e introducido en 1997 en sus microprocesadores Pentium MMX.
 - MMX reutiliza los ocho registros FPR existentes de la FPU por lo que no se puede utilizar simultáneamente con instrucciones mms e instrucciones fpu. Los registros MMX de 64 bits son directamente accesibles a diferencia de los FPR con arquitectura de pila.
 - Still, x87 instructions are the default for GCC when generating IA32 floating-point code.
- SSE: Streaming SIMD Extensions (SSE) es un conjunto de instrucciones SIMD extension del subconjunto MMX para la arquitectura x86 , no la x87, designed by Intel for digital signal processing and graphics processing applications.
 - Comenzó con el Pentium III en 1999.
 - Añade 16 nuevos registros de 128 bits XMM0-XMM15
 - XMM: SSE floating point instructions operate on a new independent register set (the XMM registers), and it adds a few integer instructions that work on MMX registers.
 - SSE2 in the Pentium 4 (2000).

- AVX: extensiones vectoriales avanzadas
 - Añade 16 registros de 256 bits: YMM0-YMM15
 - Las instrucciones que antes operaban con XMM de 128 bits ahora operan con los 128 bits de menor peso de los YMM.

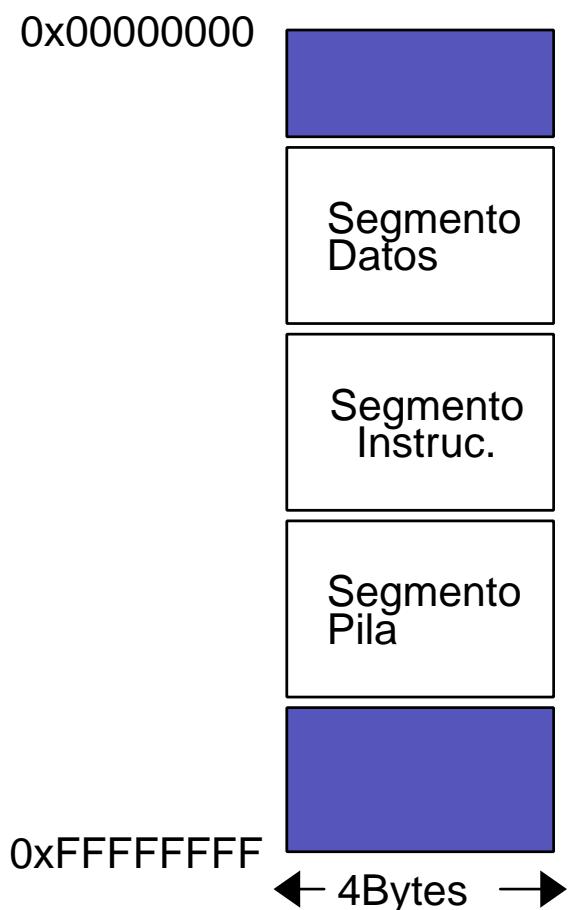
13.1.2. Refs

- [Programming With the x87 Floating-Point Unit](http://home.agh.edu.pl/~amrozek/x87.pdf) [<http://home.agh.edu.pl/~amrozek/x87.pdf>]: Intel Vol. 1 8-1
- [Computer Systems: A Programmer's Perspective, 2/E \(CS:APP2e\)](http://csapp.cs.cmu.edu/2e/waside.html) Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University [<http://csapp.cs.cmu.edu/2e/waside.html>]

Chapter 14. Pila

14.1. Concepto

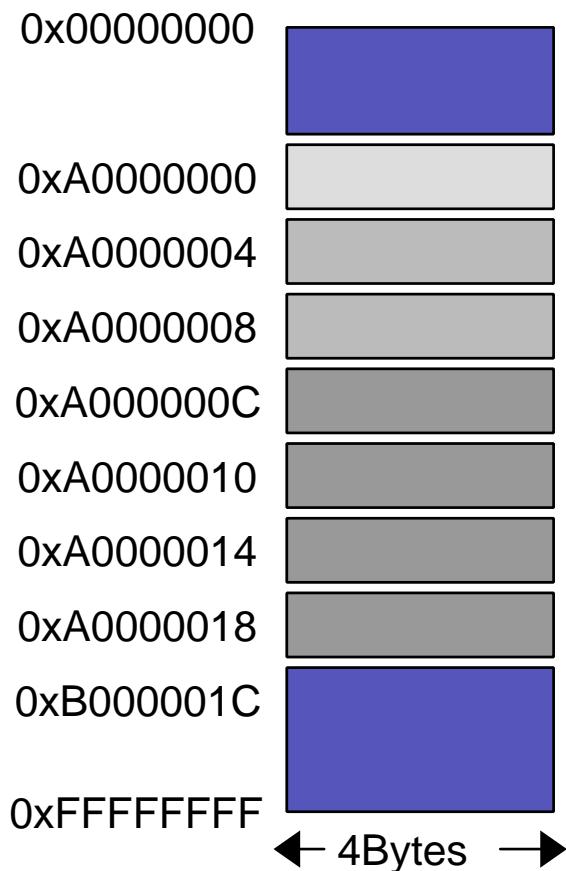
- Stack ó Pila:
 - Estructura de Datos Last Input First Output (LIFO)
- Memoria Externa
- Dirección de apilamiento: En sentido de direcciones de memoria DECRECIENTE.
- Un programa está estructurado en segmentos: Segmento datos, Segmento instrucciones, Segmento pila, ...
 - Memoria Principal Segmentada:



14.2. Anchura

- Anchura de la pila → Word Size :
 - En el caso de x86-64 : anchura de 64 bits
 - En la arquitectura i386 son 32 bits

- Alineamiento de memoria de pila → múltiplos del word size
 - En el caso de x86-64 : múltiplos de 8 bytes (64 bits) → Direcciones en hexadecimal finalizadas en 0 y en 8.
 - Si el dato a apilar es menor que la anchura de la pila será necesario extenderlo. El tipo de extensión dependerá del tipo de dato (entero con signo, etc)
- Segmento Pila de la arquitectura i386:



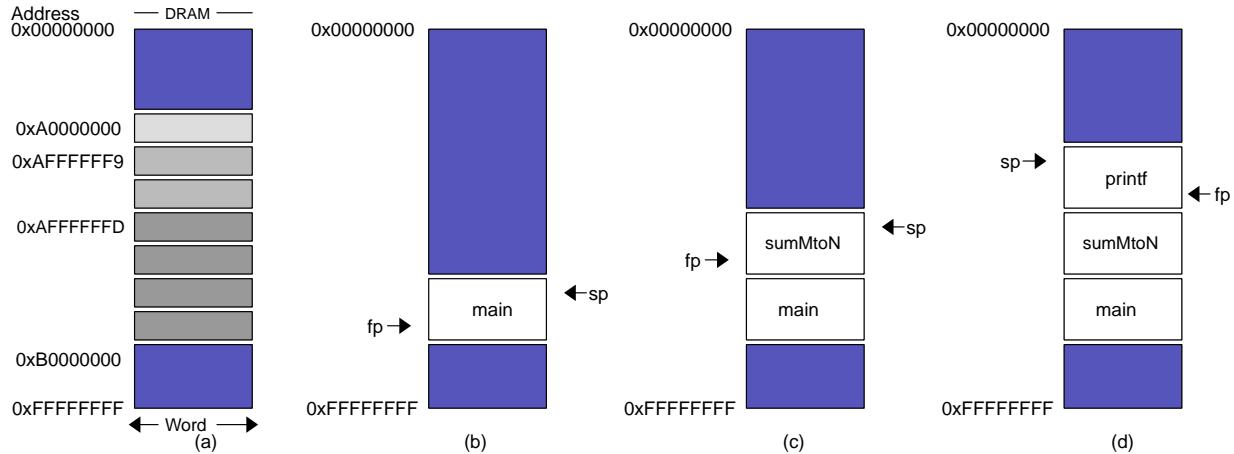
14.3. Frame: frame pointer y stack pointer

- Frame: Partición de la sección pila
 - Cada función que es llamada genera un frame
 - Los límites del **frame activo** se señalan con dos punteros:
 - límite inferior: frame pointer, señala la ubicación del *primer* elemento apilado.
 - límite superior: stack pointer, señala la ubicación del *último* elemento apilado.
- Stack Pointer (**sp**)
 - Puntero que apunta al elemento TOP del frame: límite alto de la pila donde se ubica el *último* elemento apilado.
 - En intel x86 es el registro RSP

- Frame pointer (**fp**)

- Puntero que apunta al elemento BOTTOM del frame : límite bajo de la pila donde se ubica el primer elemento apilado.
- En intel x86 es el registro RBP

- Sección de Pila (partición en Frames)



- (a) La pila no esta formada
- (b) llamada a main: se forma el frame de main. El frame crece y decrece según apilamos y extraemos
- (c) llamada de main a sumMtoN: el frame sumMtoN se forma sobre el anterior de main: nuevos punteros FP y SP.
- (d) llamada de sumMtoN a printf: el frame printf se forma sobre el anterior de sumMtoN: nuevos punteros FP y SP.



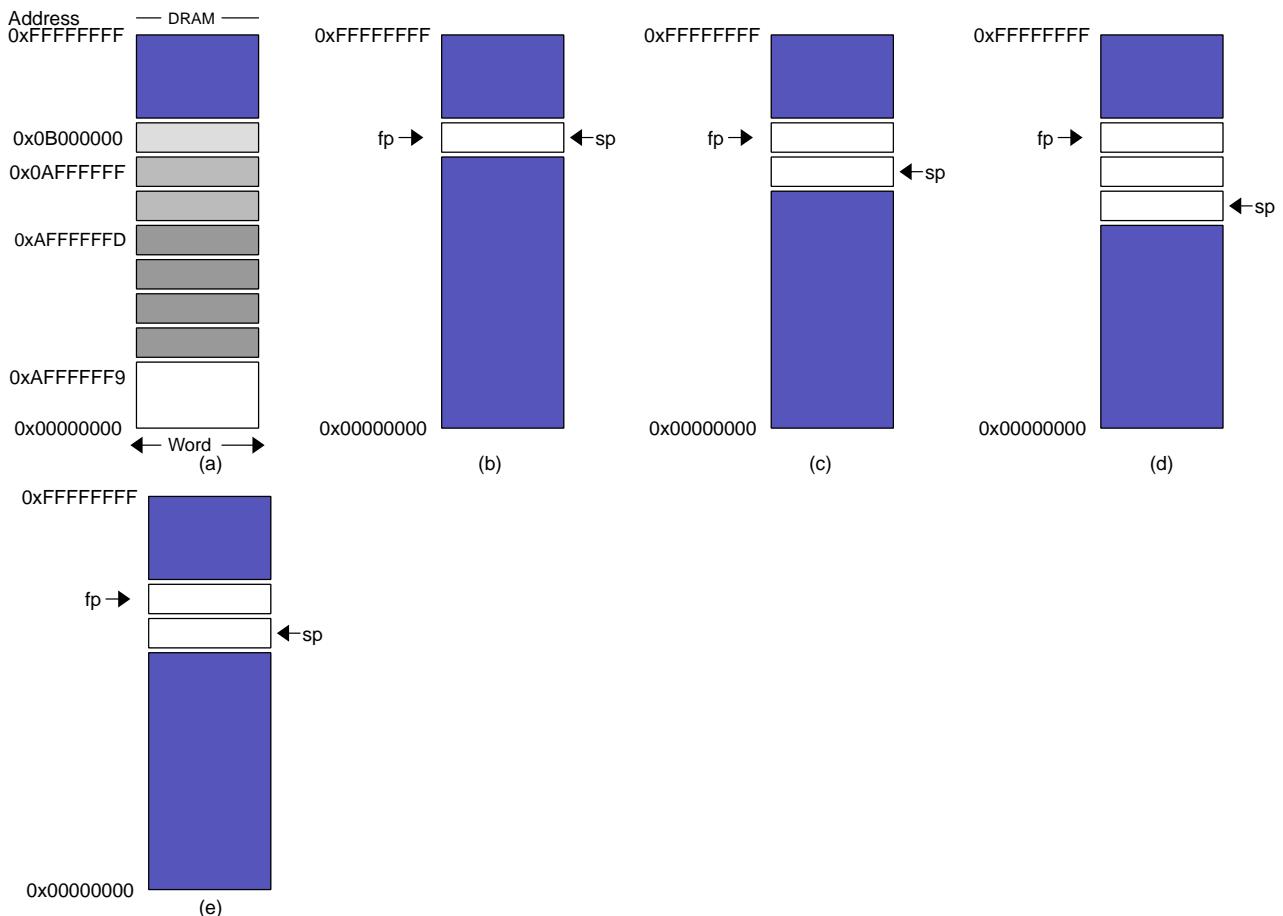
La pila es una estructura dinámica que se genera en el momento de la llamada de una función y desaparece con el retorno de la función

14.4. Instrucciones Ensamblador Push-Pop

- Instrucción Push-Pop : Apilamiento-Extracción

- Push Op_source
 - Operación: insertar dato.
 - Operando destino: la pila.
 - El stack pointer se DECREMENTA en una palabra . $SP \leftarrow SP - 1 * \text{WordSize}$ y después se inserta el operando fuente en el destino.
- Pop Op_dest
 - Operación: extraer dato.
 - Operando fuente: El último objeto apilado.
 - Primero se extrae el objeto referenciado por el stack pointer. A continuación el stack

pointer se INCREMENTA en una palabra. $SP \leftarrow SP + 1 * \text{WordSize}$



- (a) La pila no esta formada
- (b) Se forma la pila inicializando los punteros de pila: frame pointer (fp) y stack pointer (sp)
- (c) Ejecución de push
- (d) Ejecución de push
- (e) Ejecución de pop

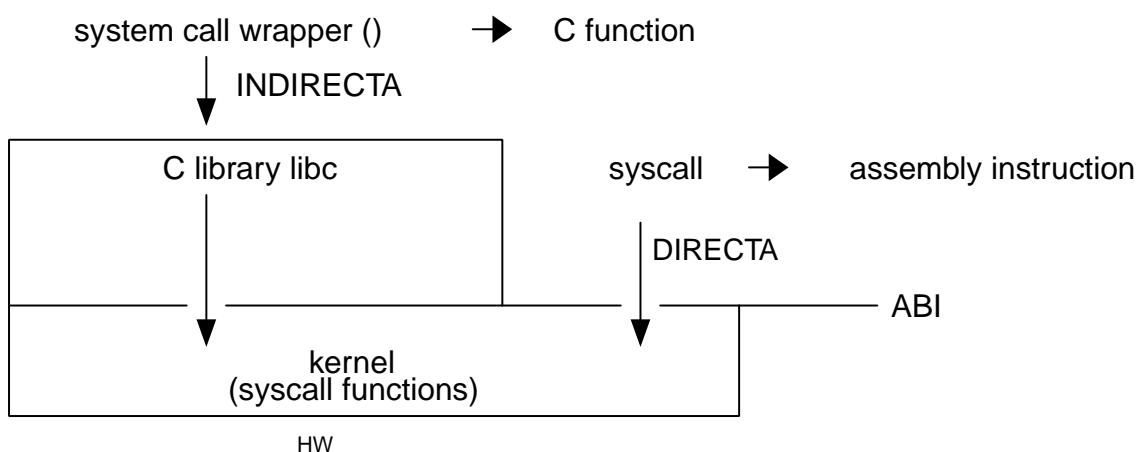
14.4.1. Anidamiento de llamadas

- TODO

Chapter 15. Llamadas al Sistema Operativo

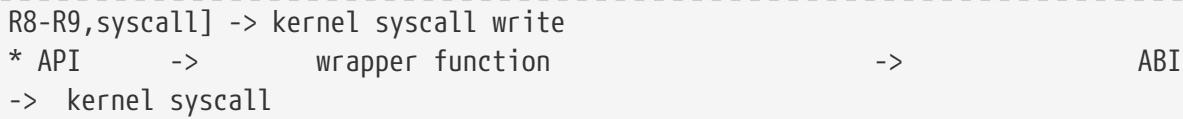
15.1. Introducción

- 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

```
* printf() -> write(int fd, const void *buf, size_t count) -> [RAX-RDI-RSI-RDX-R10-
```



15.2. Manuales de las llamadas

- Los syscall están descritos en los manuales de los wrappers de la librería libc
- listado de los syscall
 - `info syscalls` o `man syscalls`
- syscalls:
 - exit → `man 3 exit`
 - read → `man 2 read`
 - write → `man 2 write`
 - open → `man 2 open`
 - close → `man 2 close`
 - etc..
- Los argumentos de la llamada al sistema son los asociados a la función wrapper de la biblioteca libc.
 - El 1º argumento de la llamada al sistema es el argumento de la IZDA de la función en libc y el último el de la DCHA.

15.3. Llamada INDIRECTA

- **C:** El programador de aplicaciones en C utiliza las funciones interfaz de la librería *libc* de GNU para acceder **indirectamente** al kernel a través de los *contenedores (wrapper)*.
 - system calls wrapper: adaptación al lenguaje C de las llamadas implementadas en lenguaje ASM

15.4. LLamada DIRECTA

- **ASM:** El programador de aplicaciones en lenguaje ASM utiliza las *llamadas al sistema* para acceder **directamente** al kernel
 - La llamada se realiza mediante la instrucción ensamblador `syscall` en x86-64 y `int 0x80` en x86-32
 - Los argumentos de la llamada se pasan a través de los registros de propósito general GPR
 - El tipo de llamada se especifica a través de un número entero y se pasa a través **RAX**
 - Códigos "System call number" disponibles en el fichero `/usr/include/asm/unistd_32.h`

`## Macros en el fichero macros_x86-64_gas.h`

```

## Llamadas al Sistema
.equ STDIN_ID, 0      # input-device (keyboard)
.equ STDOUT_ID, 1     # output-device (screen)
.equ SYS_READ, 0       # ID-number for 'read'
.equ SYS_WRITE, 1      # ID-number for 'write'
.equ SYS_OPEN, 2        # ID-number for 'open'
.equ SYS_CLOSE, 3       # ID-number for 'close'
.equ SYS_EXIT, 60       # ID-number for 'exit'

```

15.4.1. Argumentos de la llamada directa

- El convenio de la llamada está descrito en la norma ABI
- x86-64
 - Los 6 primeros argumentos de la llamada se pasan a través de los registros siguiendo la secuencia: **RDI-RSI-RDX-R10-R8-R9**
 - El valor de retorno de la llamada se pasa a través del registro **RAX**
- x86-32
 - Los 6 primeros argumentos se pasan a través de los registros siguiendo la secuencia: **EBX-ECX-EDX-ESI-EDI-EBP**
 - El valor de retorno de la llamada se pasa a través del registro **EAX**
- manual libc: Información sobre cuáles son los argumentos de las llamadas

15.4.2. Códigos de la llamada directa

- El código de llamada es un número entero asociado a la función que va a ejecutar el kernel
- El código de llamada se pasa al kernel a través de **RAX**
- Códigos:
 - */usr/include/asm/unistd_64.h*: declaración de macros con el código de la llamada en la arquitectura x86-64
 - exit → 60, read → 0, write → 1, open → 2, close → 3, etc..
 - */usr/include/asm/unistd_32.h* : declaración de macros con el código de la llamada en la arquitectura x86-32
 - */usr/include/bits/syscall.h* : macros antiguas también válidas en la arquitectura x86-32

15.5. Ejemplos: lenguaje C

- **exit (status_value)** y **syscall(exit_code,status_value)**
 - exit(0xFF) y syscall(60,0xFF)
- **write (int fd, const void *buf, size_t count)** y **syscall(write_code,int fd, const void *buf, size_t count)**
 - write (0,buffer,80) y syscall(1,1,buffer,80)

15.6. Ejemplos: ASM INDIRECTO

- Programando en lenguaje ASM podemos llamar a los wrappers de la librería libc.
- `exit(status_value)`

```
mov $status_value,%rdi  
call exit
```

- `syscall(exit_code,status_value)`

```
mov $60,%rax  
mov $status_value,%rdi  
call syscall
```

- `write(int fd, const void *buf, size_t count)`

```
mov fd,%rdi          #fd es la referencia al fichero donde se va a escribir  
mov $buffer_address_label,%rsi #dirección de memoria de lo que se va a escribir en  
el fichero  
mov size,%rdx          #tamaño del buffer de memoria que se va a escribir  
call write            #orden de escritura al kernel a través de la librería libc
```

- `syscall(write_code,int fd, const void *buf, size_t count)`

```
mov $1,%rax  
mov $1,%rdi          # 1 es el código del fichero pantalla. En unix los dispositivos  
son ficheros.  
mov $buffer_address_label,%rsi  
mov size,%rdx  
call syscall
```

15.7. Ejemplos: ASM DIRECTO

- `exit`

```
mov $60,%rax  
mov $status_value,%rdi  
syscall
```

- `write`

```
mov $1,%rax  
mov $1,%rdi          # 1 es el código del fichero pantalla. En unix los dispositivos
```

```

son ficheros.
mov $buffer_address_label,%rsi
mov size,%rdx
syscall

```

15.8. Línea de Comandos

15.8.1. Procedimiento

- Process Initialization
 - Cuando escribimos un comando o programa en la línea de comandos del shell el sistema operativo los interpreta como una secuencia de strings. Por ejemplo `$suma 2 3` son tres argumentos en la línea de comandos:
 - La codificación de un string es la secuencia de sus caracteres en código ASCII y finalizada con el carácter NULL cuyo código es 0x00
 - el string "suma": 5 caracteres ASCII: 0x73,0x75,0x6d,0x61,0x00
 - el string "2" : 2 caracteres ASCII: 0x32,0x00
 - el string "3" : 2 caracteres ASCII: 0x33,0x00
 - Como son 3 los argumentos de la línea el parámetro argument counter **argc** valdrá 3.
 - Los tres strings de la línea de comandos, "suma"- "2"- "3", son asignados a la variable array de strings **argv**
 - argv[0] apunta al string "suma"
 - argv[1] apunta al string "2"
 - argv[2] apunta al string "3"
 - argv[argc] apunta al carácter NULL
 - argv es una array de punteros, por lo tanto, es del tipo (char **)argv
- kernel
 - El kernel declara el prototipo `extern int main (int argc , char* argv[] , char* envp[]);`
 - declaración y definición del módulo principal **main**
 - La función **main** es declarada como global por el kernel y es definida por el usuario.
 - *argc* is a non-negative argument count;
 - *argv* is an array of argument strings, with argv[argc]==0;
 - *envp* is an array of environment strings, also terminated by a null pointer.

```

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

/*
 * Introducimos en la línea de comandos el programa y un argumento

```

```

* Si el argumento tiene espacios en blanco, entrecerrarlo con comillas
simples:'Hola Mundo'
* gcc -g -o linea_comandos linea_comandos.c
* ./programa 'Hola Mundo'
*/
```

```

int main (int parc, char *parv[])
{
    if (parc==1){
        printf("Introducimos en la línea de comandos cualquier mensaje\n\n");
        exit (EXIT_FAILURE);
    }
    printf("%s\n",parv[1]);
    return EXIT_SUCCESS;
}
```

15.8.2. Stack Initialization

- Cuando comienza a ejecutarse al función *main()* o la instrucción *_start* el estado de la pila es el siguiente:
- Stack Initialization
 - El kernel pasa los argumentos **argc** y **argv** de la función global *main* a través de la PILA. La función *main* es la función llamada.

Table 11. Convenio ABI: Stack

Stack Reference	Interpretation
	arguments strings
	0
1 word cada variable	Environment pointers
8+8*argc(%rsp)	0
8*argc(%rsp)	- pointer to argcº string
-----	-----
16(%rsp)	- pointer to 2º argument string → argv[1]
8(%rsp)	- pointer to 1º argument string → string argv[0]
0(%rsp)	- argument count → argc

15.8.3. Rutina principal con Retorno

- Si la rutina principal no termina con la llamada **exit** y termina con la instrucción **ret** el convenio de llamada es el de llamada a función por lo que los parámetros *argc* y *argv* se pasan a través de los registros **RDI-RSI-RDX-RCX-R8-R9**
- Ejemplo: *imprimir_arg.s*

```
### gcc imprimir_arg.s
### ./a.out 'Hola Mundo'
###
###

.equ STDOUT,1
.equ SYSWRITE,1
.equ EXIT_SUCCESS,0xFF
.equ ARGV1,8

mensaje:
    .ascii "Introducir un mensaje como argumento del programa. Si el mensaje
tiene espacios blancos, poner el mensaje entre comillas simples ''\n"
    .equ LON,. - mensaje      #longitud del mensaje

.section .text
.global main

main:
    push %rsi          #salvo el argumento argv
## comprobar que la linea de comandos tiene dos argumentos
    cmp $2,%rdi
    je imp_arg
## si solo tengo el programa sin argumentos :imprimir en la pantalla
    mov $SYSWRITE,%rax
    mov $STDOUT,%rdi      #fd es la referencia al fichero donde se va a
escribir
    mov $mensaje, %rsi      #dirección de memoria de lo que se va a
escribir en el fichero
    mov $LON,%rdx          #tamaño del buffer de memoria que se va a
escribir
    syscall                #orden de escritura al kernel
    jmp salida

imp_arg:
    pop %rsi           #el stack pointer apunta al %rsi salvado y lo recupero
-> argv -> argv[0]
    add $ARGV1, %rsi     #rsi apunta al primer puntero, si le sumo 8 apunto al
segundo puntero
    mov (%rsi), %rdi     #mediante la indirección tengo el segundo puntero
    call puts

salida:
    ret
```

```
.end
```

15.8.4. Ejercicios: suma_linea_com.s ,maximum_linea_com.s

1. suma_linea_com.s

- Introducir los datos del programa *suma_linea_com.s* (suma de dos sumandos) a través de la línea de comandos

```
### función: sumar dos números enteros de un dígito.  
### los sumandos se pasan a través de la línea de comandos  
## Compilación en la arquitectura x86-64  
## gcc -nostartfiles -g -o sum_input sum_imput.s  
## run 5 7  
## x /x %rsp           ->3          argc:número de  
argumentos  
## x /a *(char**)(%rsp+8) ->0xfffffd0a4: 0xfffffd26e  
## x /c *(char**)(%rsp+8) ->0xfffffd26e: 47 '/'  
## x /s *(char**)(%rsp+8) ->0xfffffd26e:  
"/home/candido/tutoriales/asTutorial/algoritmos_x86-32/basicos/sum_input"  
## p /s *(char**)(%rsp+8) ->0xfffffd26e  
"/home/candido/tutoriales/asTutorial/algoritmos_x86-32/basicos/sum_input"  
## x /s *(char**)(%rsp+16) ->0xfffffd2b7: "5"  
## x /s *(char**)(%rsp+24) ->0xfffffd2b9: "7"  
  
.section .text  
.globl _start  
_start:  
  
## instrucciones aclaratorias  
  
    lea 8(%rsp),%rax      #eax contiene argv[1] la dirección de la pila que  

```

```

        movb (%rbx),%dl          # indirección para acceder al string
referenciado por argv[1]
        sub $0x30,%rcx
        sub $0x30,%rdx

        mov %rcx,%rsi
        mov %rdx,%rdi

        call suma

## salida
        mov %rax,%rdi
        mov $60,%rax      #1 is the exit() syscall
        syscall

### Función que calcula la suma entre dos valores
.type suma, @function
.section .text
suma:
## prologo
        push %rbp
        mov %rsp,%rbp
        sub $8,%rsp           #reserva de memoria

## captura de argumentos
        mov %rdi,%rax      #1º argumento
        mov %rsi,%rcx      #2º argumento
## cuerpo
        addl %ecx,%eax      #
## guardar resultado
## el resultado está en EAX
salto:
## epilog
        mov %rbp,%rsp          # frame anterior
        pop %rbp
        ret                   # recuperar dirección de retorno

```

2. maximum_linea_com.s

- Introducir los datos del programa *maximum_linea_com.s* a través de la línea de comandos

Chapter 16. Lenguaje de Programación C

16.1. Introducción

- Esto no es un tutorial de Programación en Lenguaje C, el objetivo de este capítulo es comentar aspectos puntuales de la programación en lenguaje C que son utilizados en la asignatura de Estructura de Computadores.

16.2. Casting

16.2.1. Concepto

- Sintaxis

```
(type_name) expression
```

- Conversión explícita mediante el operador unitario () .
- Los operadores unitarios tienen mayor precedencia que los binarios.

16.2.2. Ejemplo

- Ejemplo de la división de números enteros

```
int i=8,j=5;  
float x;  
x = i / j;  
x = (float) i / j;
```

- La variable ordinaria es declarada inicialmente como tipo *int*
 - La operación i/j → 8/5 daría como resultado el número entero 1
- Si realizamos el casting (**float**) sobre la variable **i** entonces la variable **i** es de tipo float y no int, por lo que su valor será el número real 8.000 y no el entero 8.
 - (float)i/j = 8.000/5 = 1.6000

16.3. Puntero

16.3.1. Referencias

Libro de texto K.N. King: Capítulo 11. Pointers. Pg241

16.3.2. Introducción

El concepto de puntero es fundamental en programación imperativa de bajo nivel ya que simplifica

el código para la programación de algoritmos que incluyen estructuras de datos sencillas o complejas.

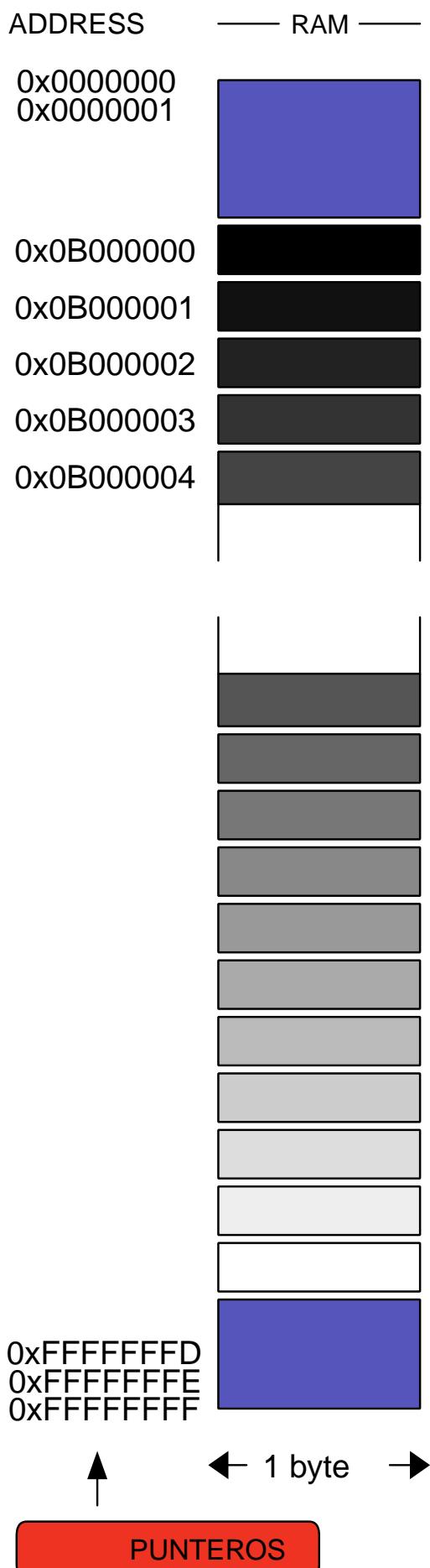


Para el aprendizaje de: conceptos relacionados con los punteros, su sintaxis, su aplicación, etc..., es necesaria la ejecución de los programas en modo PASO A PASO para poder visualizar los contenidos y referencias de los objetos en memoria. Utilizaremos el debugger GDB.

16.3.3. Concepto

Memoria

- La memoria principal RAM esta organizada en Bytes direccionables.
- El rango de direcciones depende de la arquitectura de la máquina.
- P.ej: un Bus de direcciones de 48 líneas podría direccionar $2^{48} = 2^8 \times 2^{40} = 256 \text{ TB}$
- Un *objeto* es una región de memoria (múltiples bytes) asignada a un dato entero, dato carácter, array de datos float, bloque de instrucciones, etc. En este contexto de memoria el concepto objeto difiere del concepto objeto de programación orientada a objetos.
- En la memoria RAM se implementan *objetos* que son referenciados por las direcciones de memoria donde se encuentran. La referencia es la dirección del primer byte donde se almacena el objeto de múltiples bytes.
- Mapa de memoria:



Puntero

Un puntero equivale a una dirección de memoria

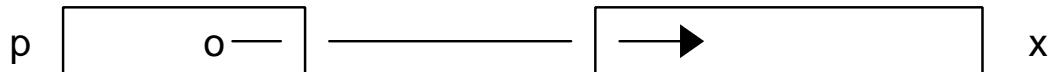
En cambio una VARIABLE PUNTERO:

- Es una variable que almacena un dato que representa una dirección de memoria.
- Las variables puntero almancenan punteros.
- Restringen sus valores a los valores de las direcciones de memoria. Nunca podrá ser un valor negativo o real, etc
- Apuntan a objetos
- Hacen referencia a objetos



El libro de K.N.King distingue entre "variable puntero" y puntero. En la literatura en general cuando se habla de punteros se está hablando de variables puntero, en cuyo caso al contenido del puntero se le llama referencia o dirección al objeto referenciado.

Representación gráfica de la "variable puntero" *p*

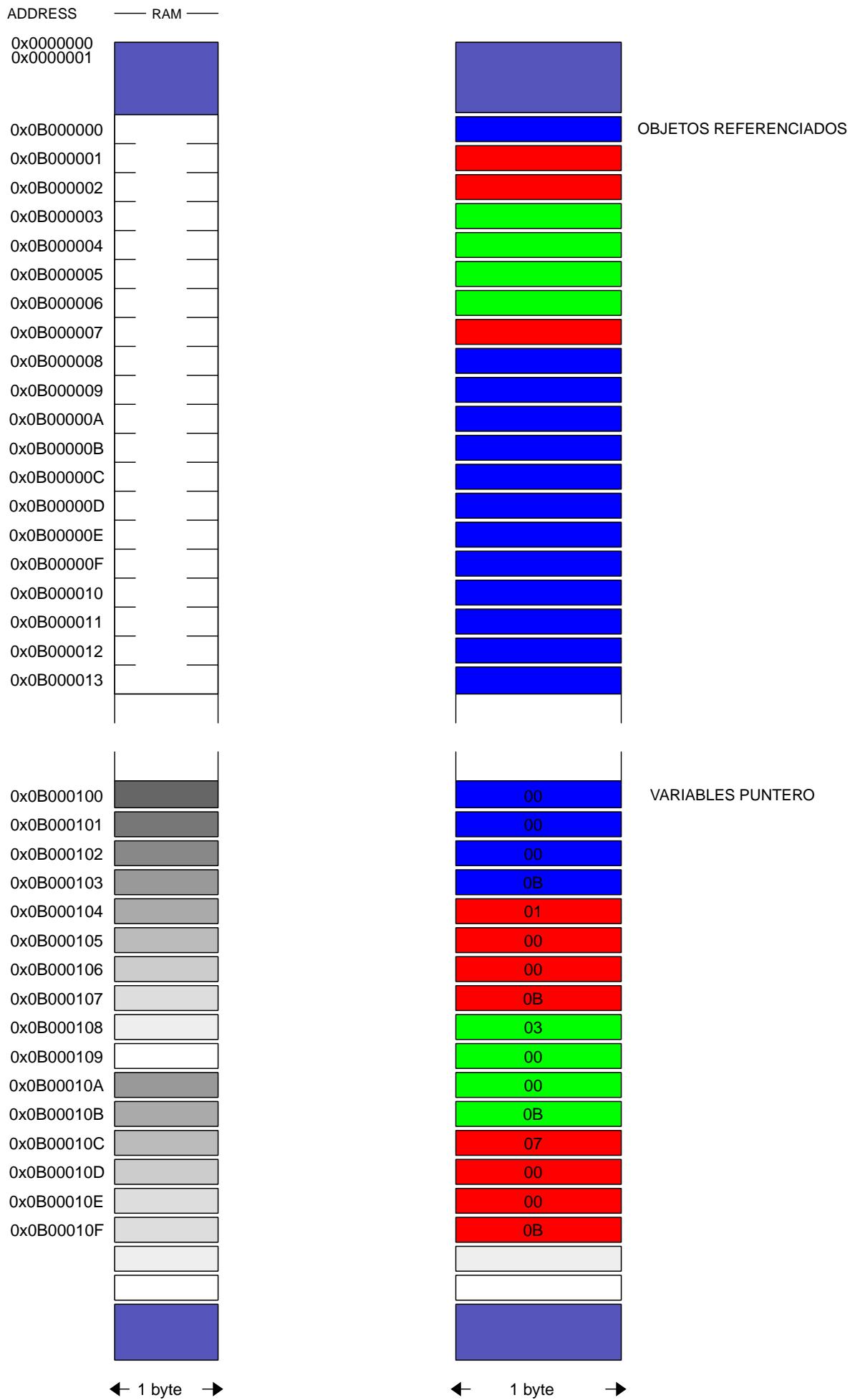


p : identificador de la variable puntero

x : identificador del objeto referenciado, por ejemplo una variable ordinaria.

flecha : *inicialización* de la variable puntero *p* apuntando al objeto *x*

Ejemplos de punteros, objetos y variables de punteros



- La variable puntero de la dirección 0x0B000100 (bytes +0,+1,+2,+3) contiene la dirección 0x0B000000 que apunta a un objeto de 1 byte.
- La variable puntero de la dirección 0x0B000104 (bytes +0,+1,+2,+3) contiene la dirección 0x0B000001 que apunta a un objeto de 2 bytes.
- La variable puntero de la dirección 0x0B000108 (bytes +0,+1,+2,+3) contiene la dirección 0x0B000003 que apunta a un objeto de 4 bytes.

LeftValue-RightValue

- Una variable ordinaria referenciada en un operador asignación (=) tiene diferente interpretación si está a la izquierda o derecha del operador asignación:
 - x=y
 - x : la variable ordinaria a la izda se interpreta como la dirección en memoria de x : leftvalue de x
 - y : la variable ordinaria a la derecha se interpreta como el contenido en memoria de y : rightvalue de y
- El contenido del objeto es el RightValue
- La referencia al objeto es el LeftValue
- El contenido de una variable puntero es el LeftValue del objeto referenciado.

16.3.4. Módulo Ilustrativo

```
/* Iniciación a los punteros.*/
```

```
#include <stdio.h>
#include <stdlib.h>
```

```
void main (void)
{
```

```
/* Concepto */
```

```
/*Operador Dirección*/
```

```
int i, k, *p, *q;
float x, y, *r, *s;
char c, d, *u, *v;
i = 10;
k = 100;
x = 3E-10f;
y = 3.1416;
c = 'A';
d = '@';
```

```

p = &i;
q = &k;
r = &x;
s = &y;
u = &c;
v = &d;

printf("Introducir un carácter \n");
scanf("%c", &c);
printf("El carácter leído es el %c \n", c);

/*Operador Indirección*/

printf("El carácter leído es el %c \n", *u);
printf("El valor de la variable i es %d o también %d \n", i, *p);
printf("El valor de PI es %f o también %f \n", y, *s);

/*String Variable*/
/*Array*/
char cadena[]="Hola";

/*Puntero*/
char *saludo="Hola";
char **pt_saludo;

pt_saludo = &saludo;

exit (0);
}

```

16.3.5. Declaración

Syntaxis: `type *pointer_variable`

```

int i, k, *p, *q;
float x, y, *r, *s;
char c, d, *u, *v;
i = 10;
k = 100;
x = 3E-10f;
y = 3.1416;
c = 'A';
d = '@';

```

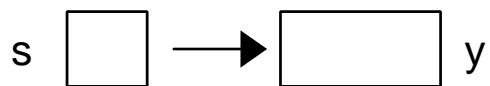
`*p, *q, etc...` son declaraciones de *variable puntero*. El asterisco NO realiza ninguna operación sobre la variable, únicamente es el prefijo para indicar el TIPO puntero.

16.3.6. Operador Dirección

Símbolo &

```
p = &i;  
q = &k;  
r = &x;  
s = &y;  
u = &c;  
v = &d;  
scanf(&c);  
scanf(u);
```

El operador & obtiene el LeftValue de la variable y se utiliza para inicializar punteros.



16.3.7. Operador Indirección o Dereferencia

Símbolo *

Prefijo de una variable puntero: accede al objeto referenciado

```
printf("El valor de la variable i es %d o %d \n", i, *p);  
printf("El valor de PI es %f o %f \n", y, *s);
```

16.3.8. Ejemplo

- Declarar objetos de distintos tipos: integer, float, char
- Declarar objetos de tipo puntero e inicializarlos con los objetos anteriores
- Representar gráficamente los punteros
 - Low Level: memoria RAM
 - High Level: diagramas con cajas que apuntan con flechas.

16.3.9. Aplicaciones de los punteros

- Array
 - Puntero Array
 - Aritmética de Punteros
- String Literal
- Puntero a Puntero
- Acceso a String
 - Nombre del array
 - Variable puntero
- Estructura de datos
 - Lista de Nombres (Array de punteros a strings)
- Funciones
 - Pase de argumentos por referencia
 - Retorno por referencia.
- Argumentos del comando en línea del shell de Linux.

Puntero Array

- Concepto
 - Un array es un puntero y una lista de elementos. El puntero apunta al primer elemento de la lista.
 - Cuando se crea un array se crean dos objetos
 - Los elementos del array cuya asignación de memoria es contigua
 - El puntero que apunta al primer elemento del array
- Ejemplo
 - Array de Números : `data_items : 3,67,34,222,45,75,54,34,44,33,22,11,66,0`
 - Declarar e inicializar
 - Lectura
 - Escritura
- Puntero CONSTANTE
 - NO SE PUEDE MODIFICAR EL VALOR DEL PUNTERO
 - Modificar el puntero
- Ejemplo:
 - Array de Caracteres: `cadena : H,o,l,a,\0`
 - Declarar e inicializar `char cadena[]={H,o,l,a,\0};`
 - Lectura

- Escritura

Aritmética de Punteros

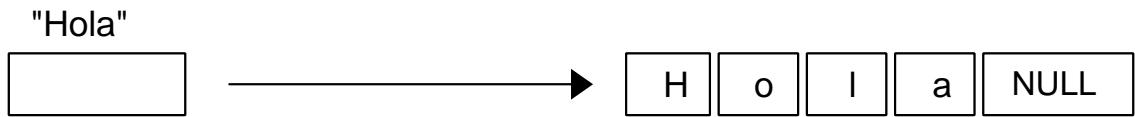
- Indexación: primer elemento MÁS la posición elemento i
 - $data_items + i$
- Modificar las expresiones de referencia a los elementos del array por expresiones aritmética de punteros

16.3.10. String Literal

- Concepto en dos fases
 - Array de nombre "Hola" cuyos elementos son de tipo carácter.

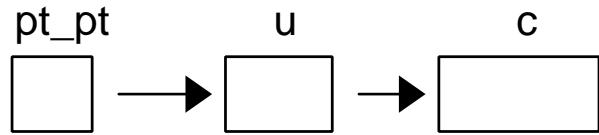


- Inicializar Array con el String *Hola*



- Ejemplo
 - Declarar un array tipo carácter e inicializarlo con un string literal "Hola"
 - `char cadena[]="Hola";`
 - String literal:
 - Cadena de caracteres
 - Dobles comillas
- Arrays
 - Acceder al array declarado : lectura y escritura
 - Acceder al array de inicialización: lectura y escritura
 - ¿ Copia de arrays mediante asignación `cadena1=cadena2` ?

16.3.11. Puntero a Puntero



- Ejemplo
 - *u* apunta al carácter *c*
 - *pt_pt* apunta a *u*

16.3.12. String Variable

Nombre del Array

- Declaro el array de caracteres `cadena` y lo inicializo con el string `Hola` : `char cadena[]="Hola";`

Variable Puntero

- Declaro la variable `saludo` y lo inicializo con el puntero `cadena`
 - `char **saludo`

16.3.13. Funciones

- Pase de argumentos por **Referencia**
 - Declarar los parámetros de la función como variables puntero.
- Retorno por **Referencia**.
 - Declarar el valor de retorno como puntero.

Chapter 17. Apéndice Prácticas

17.1. Prácticas

17.1.1. Documentación: guiones, bibliografía, apuntes

- Disponible en miaulario:
 - Apuntes *eecc_book.pdf* que incluyen los guiones, hojas de referencia, apéndices, ejercicios de autoevaluación y teoría.
 - Los módulos con el código fuente *.s* (miaulario/Recursos/prácticas/codigo_fuente.zip) [link G1](#) [https://miaulario.unavarra.es/portal/site/2018_0_240306_1/tool/d61518aa-72cb-45df-a7f8-b71f90e7907e?panel>Main] utilizados en todas las prácticas están disponibles en el servidor de miaulario de la UPNA: *Recursos/prácticas*
 - El libro de texto en que se basan los guiones de prácticas en lenguaje ensamblador : *Programming from the Ground-Up*.

17.1.2. Plataforma de Desarrollo

Herramientas

- Editores
 - [Editores](https://www.tecmint.com/best-open-source-linux-text-editors/) [<https://www.tecmint.com/best-open-source-linux-text-editors/>]: gedit, emacs, vim, sublime, kate,
 - [Herramientas integradas de edición, compilación, depuración](https://www.tecmint.com/best-linux-ide-editors-source-code-editors/) [<https://www.tecmint.com/best-linux-ide-editors-source-code-editors/>]: eclipse CDT, netbeans, code::blocks, codelite, Microsoft's Visual Studio Code Editor, jetbrains clion, jeany, ajunta , GNAT Programming Studio, emacs, kdevelop, codestudio, etc
- Denominaciones
 - **i386** : denominación de linux a la arquitectura x86-32
 - **amd64**: denominación de linux a la arquitectura x86-64
 - **IA32**: denominación de Intel para la arquitectura x86-32
 - **IA64**: denominación de Intel para la arquitectura x86-64
- Sistema Operativo GNU/linux: Distribución Ubuntu : cualquier versión posterior al año 2014: 14.04, 14.08,..,17.04, 17.08
 - **lsb_release -a**: distribución
 - **uname -o** : S.O.
 - **uname -r** : kernel
 - **uname -a** : procesador
- Librerías necesarias para que las herramientas *gcc*, *as*, *ld* sean operativas en la arquitectura **i386** de 32 bits.

- `dpkg -l gcc-multilib`:

```
Deseado=desconocido(U)/Instalar/eliminar/Purgar/retener(H)
|
Estado=No/Inst/ficheros-Conf/desempaquetado/medio-conf/medio-inst(H)/espera-
disparo(W)/pendiente-disparo
|/ Err?=(ninguno)/requiere-Reinst (Estado,Err: mayúsc.=malo)
||/ Nombre          Versión      Arquitectura Descripción
+---+-----+-----+-----+-----+
ii  gcc-multilib   4:7.3.0-3ubu amd64      GNU C compiler (multilib files)
```

- Si en las dos primeras columnas "Deseado/Estado" no pone **ii** significa que no están instaladas las librerías.
 - Compruebo que están en el repositorio accesible a través de la red internet:
 - `apt-cache show gcc-multilib`: repositorio
 - `sudo apt-get install gcc-multilib`: descarga e instalación sólo en caso de tener derechos de administrador
- Toolchain
 - `as --version & ld --version & gcc --version`: anotar las versiones

Programación online

- **ias assembler unicamp online** [<https://www.ic.unicamp.br/en/~edson/disciplinas/mc404/2017-2s/abef/IAS-Assembler/assembler.html>]
- **gdb online** [<https://www.onlinegdb.com/classroom>]

Referencias

- Recursos **GNU**:
 - Herramienta integrada de desarrollo IDE (Emacs,Eclipse,Vim, etc...) o un Editor (Geany,Kate,Gedit,Sublime, etc...)
 - `as` : ensamblador del lenguaje AT&T
 - `ld` : linker
 - `cc` : compilador de C
 - **GCC** : front-end del toolchain automático : Gnu Compiler Collection. Driver de diferentes lenguajes dependiendo de la extensión del fichero fuente.
 - `man gcc`
 - **GDB** : depurador.
 - `man gdb`

17.1.3. Documento Memoria: Contenido y Formato

Contenido

- Durante el desarrollo de la práctica :
 - a. Es necesario reeditar el código fuente de los programas desarrollados con *comentarios*.
 - b. Compilar el módulo fuente mediante *comandos en línea*
 - c. Analizar el código fuente y binario mediante el *depurador*: las operaciones a realizar con el depurador es necesario salvarlas en un fichero.
- Durante la realización de la práctica es necesario tener abierto un Editor de texto para ir realizando la memoria simultáneamente a la ejecución de la práctica.
- El Documento Memoria ha de contener:
 - Una portada con el título de la práctica y los datos personales.
 - La primera hoja con una tabla de contenidos a modo de índice, no es necesario indicar Nº de página.
 - Los módulos fuente comentados,
 - Los comandos de compilación y análisis.
 - El historial de comandos GDB y sus salidas, utilizados durante la práctica.
 - Un apartado de conclusiones con lo aprendido en la práctica.
 - Un apartado de dudas sin resolver.
 - Preguntas explícitas que aparecen a lo largo de la memoria, si las hay.
 - **OPCIONALMENTE** las preguntas y respuestas del cuestionario de Autoevaluación de Prácticas. Ver apartado Evaluación.
 - Todo tipo Informacion Personal Necesaria a modo de apuntes para utilizar en el exámen.

Formato

- La estructura interna de la memoria es libre.
- El formato de la memoria ha de ser **PDF**, y no microsoft word u otro formato diferente.
- El nombre del fichero memoria ha de ser **N-XXX-apellido1_apellido2.pdf**
 - el nombre del ficheero no contendrá sni acentos ni eñes ni espacios en blanco
 - XX significa el grupo de prácticas: P1 ó P2 ó P3
 - N significa el número de la sesión de prácticas: 1,2,3,4 ó 5.

Entrega del Documento Memoria

- Entregar el Documento Memoria a través de la aplicación **Tareas** del Servidor Miaulario. El plazo será el indicado por el profesor a través del calendario de tareas. La entrega de memorias fuera de plazo significa tener que examinarse de dicha práctica en la convocatoria ordinaria.

17.1.4. Evaluación

- Se evaluará:
 - la entrega de la memoria por el canal establecido con una penalización de 1 punto por cada día de retraso.
 - la estructura y formato de la memoria con los datos personales, índice, introducción, desarrollo, conclusiones y formato pdf con el nombre apropiado.
 - los comentarios de alto nivel (pseudocódigo) especificados en el módulo fuente tanto a nivel de bloque de instrucciones como instrucciones complicadas de interpretar o que se consideren importantes en la comprensión del código.
 - el cuestionario opcional de **Autoevaluación de Prácticas**



El profesor evaluará de forma continua la actitud y labor del estudiante en el laboratorio pudiendo liberar al alumno de la realización del examen si los conocimientos y tareas realizadas así lo demuestran.



Las preguntas y respuestas del cuestionario de **Autoevaluación de Prácticas** localizable en el capítulo V de los apuntes de la asignatura se realizan fuera del horario de prácticas a título personal. Si no se realiza la autoevaluación la puntuación máxima de la memoria será de **6 puntos** y si se realiza la puntuación máxima será de **10 puntos**.

17.1.5. Programación

Metodología

- Leer el enunciado del programa a desarrollar.
- Editar la descripción del algoritmo como Pseudocódigo:
 - Desarrollar el algoritmo definiendo las estructuras de datos y estructuras de instrucciones.
 - constantes, variables, arrays, punteros, inicializaciones, bucles, sentencias selección, funciones y parámetros, entrada y salida del programa, etc
- Dibujar el Organigrama de alto nivel
 - Para un lenguaje de alto nivel (Pascal, C ...), basado en el pseudocódigo.
- Dibujar Organigrama de bajo nivel
 - Desarrollar el algoritmo en lenguaje **RTL** basándose en la arquitectura x86. Traducir el organigrama de alto nivel a bajo nivel. Traduciendo secciones, variables, arrays, punteros, inicializaciones, bucles, sentencias selección, subrutinas y parámetros, entrada y salida del programa etc.
- Convertir el código RTL en lenguaje ensamblador **AT&T** para la arquitectura x86.
- Compilación con **gcc** o mediante la cadena de herramientas (toolchain) : **as-ld**
 - Depurar errores de síntesis.
- Ejecución: depurar errores en modo paso a paso mediante el depurador **GDB**

17.1.6. Compilación

Módulo fuente en lenguaje C

- Compilación
 - `gcc -m32 -o sum1toN sum1toN.c`
 - *m32* : 32 bits architecture machine
 - *sum1toN.c* : módulo fuente en lenguaje C
 - *-o* : output
 - *sum1toN* sin extensión: módulo objeto ejecutable aunque sería más preciso decir cargable en la memoria principal.
 - carga en memoria principal
 - la hace automáticamente el S.O. al llamar al programa ejecutable desde un terminal o escritorio.
 - `gcc -m32 -g -o sum1toN sum1toN.c`
 - *-g*: especifica que se genere la tabla de símbolos del programa fuente *sum1toN.c* para el debugger GDB y se inserte en el módulo ejecutable *sum1toN*. De esta manera se asocian el código binario, por ejemplo de una etiqueta, a su símbolo (lenguaje texto).

Fases de la compilación

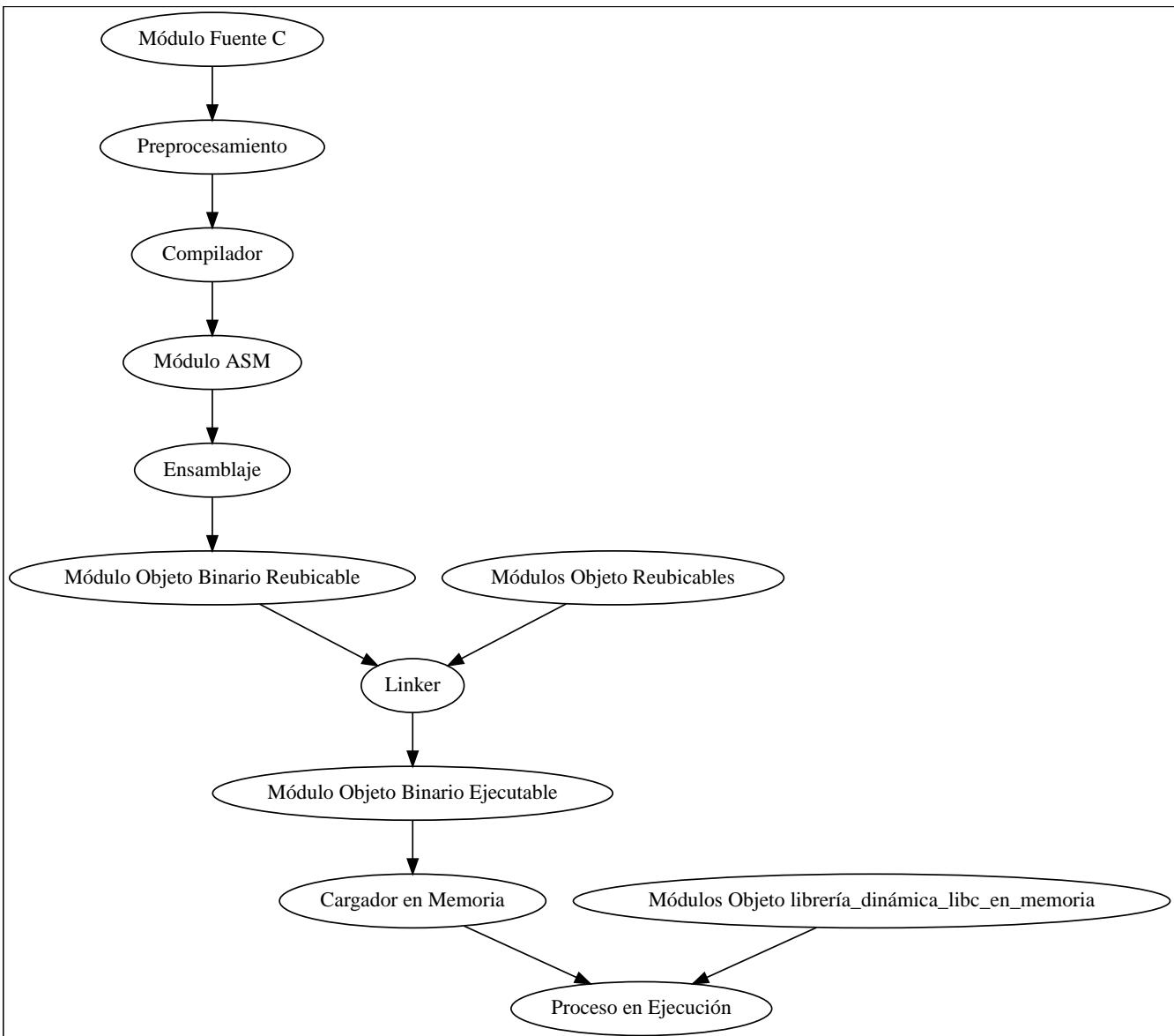


Figure 39. Proceso de Compilación

- Parar la compilación en la 1^a fase: preprocesamiento: `gcc -E sum1toN.c -o sum1toN.i`
 - *.i: Salida del preprocesador: elimina la información que no es código (comentarios,etc)
- Parar la compilación en la 2^a fase: traducir C a ensamblador: `gcc -S sum1toN.c -o sum1toN.s`
 - .s: módulo en lenguaje fuente ensamblador.s
- Parar la compilación en la 3^a fase: Generar el módulo objeto reubicable: `gcc -c sum1toN.c -o sum1toN.o`
 - *.o: módulo objeto reubicable : código binario antes de ser enlazado mediante el linker con otros módulos objeto del sistema operativo, de la librería de C *libc* u otros módulos del programador.
- Realizar las 4 fases : Generar el módulo objeto ejecutable: `gcc -c sum1toN.c -o sum1toN`
 - fichero sin extensión: módulo objeto ejecutable: módulo binario configurado para ser cargado en la memoria principal y ejecutado por la CPU.
- `gcc -m32 --save-temp -o sum1toN sum1toN.c`
 - --save-temp: gcc genera (save) los 3 ficheros parciales (temp) del proceso total de

compilación *.i.s,*o* .

- Comprobar que en total disponemos de 5 ficheros: *.c,i,s,o* y el ejecutable sin extensión.

Toolchain

- Cómo alternativa a realizar la compilación mediante un único comando con el driver **gcc** que ejecuta las distintas fases de compilación el proceso de compilación de puede realizar mediante el encadenamiento de herramientas que realizan cada una de ellas una de las distintas fases.
- Herramientas del toolchain:
 - Traducción de C a Ensamblador: no tiene una herramienta propia: `gcc -S sum1toN.c -o sum1toN.s`
 - **as**: Herramienta de Ensamblaje o ensamblador: `as --32 --gstabs -o sum1toN.o sum1toN.s`
 - `--32`: arquitectura de 32 bits
 - `--gstabs`: genera la tabla de símbolos
 - `-o`: fichero de salida : módulo binario reubicable **o*
 - **ld**: Herramienta de Enlazado ó Lincado: `ld -melf_i386 -o sum1toN sum1toN.o`
 - `-melf_i386` : arquitectura 32 bits
 - `-o`: fichero de salida : módulo binario ejecutable

módulo fuente en lenguaje ensamblador

- Comentar el programa fuente de manera abstracta funcional/operativa y no literal RTL
- Toolchain manual:
 - `as --32 --gstabs -o sum1toN.o sum1toN.s` : ensamblaje
 - `*.s` : módulo fuente en lenguaje asm
 - `*.o` : módulo objeto reubicable
 - `--stabs`: generación de la tabla de símbolos e inserción en el módulo ejecutable.
 - `--32` : módulos fuente y objeto para la ISA de 32 bits
 - `ld -melf_i386 -o sum1toN sum1toN.o`
 - `-melf_i386`: módulos objeto para la ISA de 32 bits
- Toolchain automático
 - `gcc -m32 -nostartfiles -g -o sum1toN sum1toN.s`
 - `-m32`: módulos fuente y objeto para la arquitectura i386.
 - `-nostartfiles` : especifica que el punto de entrada no es `main` sino `_start`.



Si el punto de entrada es **main** entonces es necesario informar al linker de que el punto de entrada (entry) es `main`: `gcc -e main -m32 -nostartfiles -g -o sum1toN sum1toN.s` y `ld -e main -melf_i386 -o sum1toN sum1toN.o`

- `g`: especifica que se genere la tabla de símbolos del programa fuente *sum1toN.s* para el debugger

GDB y se inserte en el módulo ejecutable *sum1toN*

17.1.7. Errores Comunes

gcc

- En Ubuntu 18.0 si se compila para amd64 (gcc -nostartfiles -g -o sum1to64 sum1to64.s) la compilación se detiene con el mensaje de error:

```
/usr/bin/x86_64-linux-gnu-ld: /tmp/ccbhD6Vr.o: relocation R_X86_64_32S against  
`.data' can not be used when making a PIE object; recompile con -fPIC  
/usr/bin/x86_64-linux-gnu-ld: falló el enlace final: Sección no representable en la  
salida  
collect2: error: ld returned 1 exit status
```

- causa: está activada por defecto al opción **-pie** y hay que desactivarla
- solución: (gcc **-no-pie** -nostartfiles -g -o sum1to64 sum1to64.s)

gdb

- El logging histórico de los comandos gdb para salvarlos en un fichero se encuentra desactivado

Chapter 18. Arquitectura amd64

18.1. Módulo fuente: sum1toN.s

```
#### Programa: sum1toN.s
### Descripción: realiza la suma de la serie 1,2,3,...N. La entrada se define en
el propio programa y la salida se pasa al S.O.
### Lenguaje: Lenguaje ensamblador de GNU para la arquitectura AMD64
### gcc -no-pie -g -nostartfiles -o sum1toN sum1toN.s
### Ensamblaje as --gstabs sum1toN.s -o sum1toN.o
    ### linker -> ld -o sum1toN sum1toN.o
## Declaración de variables
## SECCION DE DATOS
.section .data

n:    .quad 5

.global _start

## Comienzo del código
## SECCION DE INSTRUCCIONES

.section .text
_start:
    movq $0,%rdi # RDI implementa la variable suma
    movq n,%rdx
bucle:
    add %rdx,%rdi
    sub $1,%rdx
    jnz bucle

    ## el argumento de salida al S.O. a través de RDI según convenio ABI AMD64
## salida
    mov $60, %rax # código de la llamada al sistema operativo: subrutina exit
    syscall # llamada al sistema operativo para que ejecute la subrutina según el
valor de RAX

.end
```

Chapter 19. Exámenes de Cursos Anteriores

19.1. Año 2018

19.1.1. Noviembre

1ª Prueba Parcial. 2018 Noviembre 10.

Grado de Informática 2º curso. Estructura de Computadores.

Universidad Pública de Navarra.

Duración: 90 minutos.

Apellidos:

Nombre:

Puede utilizarse todo tipo de información escrita como memorias de prácticas, apuntes, hojas de referencia, etc. No puede utilizarse ningún dispositivo electrónico como calculadoras, teléfonos, ordenadores, etc ... Se han de incluir en la respuesta todo tipo de desarrollo necesario para llegar al resultado.



1. Computadora Institute Advanced Studies (IAS) de von Neumann:
 - a. (1 pto) Desarrollar un programa que realice la resta 0x00-0xFF y almacene el resultado en la variable denominada "resta". ¿Cuál será el contenido de la posición de memoria de la variable "resta"?
 - b. (1 pto) ¿Qué relación existe entre los tres componentes MAR, MBR y PC?
2. (1 pto) Cuál es el código digital del string de seis caracteres "Hola \n"
3. (1 pto) Los números 0123 y 0777 son números sin signo en base octal. Realizar la suma 0123+0777 directamente en base octal.
4. (1 pto) Los números 0xABCD y 0xEFED son números con signo en complemento a dos. Realizar la resta 0xABCD-0xEFED directamente. Calcular el valor del resultado.
5. (1 pto) Representar el número decimal 6.25 en formato IEEE-754 de doble precisión.
6. Formato de instrucciones:
 - Una computadora tiene una unidad de memoria de 256K palabras 32 bits cada una direccionable byte a byte. En una de las palabras de la memoria se almacena una instrucción. La instrucción tiene un formato de cuatro campos: un bit de indirección, un código de operación, un campo de operando para direccionar uno de los 64 registros y campo de operando que contiene direcciones de memoria.
 - a. (2 pto) ¿Cuántos bits forman el campo de código de operación? ¿Y del campo de registro? ¿Y del campo de direcciones?
 - b. (2 pto) ¿Cuántos bits forman parte del bus de direcciones y del bus de datos de la unidad de memoria?
7. (2 pto) En una subrutina indicar qué relación existe entre el puntero "frame pointer" del frame de la subrutina y la dirección de memoria donde se guarda la dirección de retorno.

8. (3 pto) Completar el código fuente del programa en lenguaje ensamblador adjunto teniendo en cuenta los comentarios que se adjuntan en el módulo fuente siguiente donde el algoritmo desarrollado realiza la conversión de un número decimal a código binario:

```

### Programa: convert_decbin.s
### Descripción: Convierte el número natural decimal 15 en binario mediante
divisiones sucesivas por 2
###           El código binario tiene un tamaño de 32 bits
### gcc -m32 -g -nostartfiles -o convert_decbin convert_decbin.s
### Ensamblaje as --32 --gstabs convert_decbin.s -o convert_decbin.o
### linker -> ld -melf_i386 -o convert_decbin convert_decbin.o

## MACROS

## DATOS

dec:    .      15    # decimal (tamaño 4 bytes) a convertir en un código binario de
32 bits
## bin almacena el código en sentido inverso, bin[0] almacena el bit de menor
peso.
bin:    .space 32  # array de 32 bytes: almacena en cada byte un bit del código
binario de 32 bits.
divisor:   .  # divisor (de tamaño 1 byte)

## INSTRUCCIONES

## inicializo ECX con el valor del divisor

## inicializo el índice del array bin

## Cargo el dividendo en EAX
# eax <-x

## extiendo el bit de signo del dividendo en EDX
# El dividendo siempre es positivo

## Divisiones sucesivas por 2 hasta que el cociente valga 0
bucle:
## idivl : [EDX:EAX] / Operando_fuente
# EAX<-Cociente{x/y} , EDX<-Resto{x/y}
# guardo el resto (de tamaño 1 byte) en el array bin
## extiendo el bit de signo en edx
# El dividendo siempre es positivo

```

```

## actualizo el índice del array

## compruebo si el cociente ha llegado a cero para salir del bucle

## Devuelvo el número de bits del código binario en EBX

## Código de la llamada al sistema operativo

## Interrumpo la rutina y llamo al S.O.

```

- Mediante comandos del depurador GDB
 - a. (2 pto) imprimir el contenido del array "bin" con dos expresiones diferentes utilizando los comandos "examinar" y/o "imprimir".
 - b. (2 pto) imprimir el contenido del primer elemento del array bin
 - c. (2 pto) imprimir el contenido del último elemento del array bin
9. (2 pto) Llamadas al sistema
- Completar el programa "convert_decbin.s" con el código necesario para imprimir en la pantalla un mensaje de bienvenida mediante la llamada directa write.

19.2. Año 2017

Prueba Parcial. 2017 Septiembre 22.
 Grado de Informática 2º curso. Estructura de Computadores.
 Universidad Pública de Navarra.
 Duración: 30 minutos.
 Apellidos:
 Nombre:

1. En el modelo de Von Neumann cuál es la función de la Unidad de Control .
2. Cuáles son las distintas fases del ciclo de instrucción de la máquina de Von Neumann.
3. Convertir el número decimal 291 en base octal.
4. Realizar la operación -18-21 en complemento a 2.
5. En qué consiste el concepto de abstracción en la organización de una computadora.
6. Desarrollar el programa en lenguaje ensamblador sum.ias, de la máquina IAS, que implemente el algoritmo $s=1+2$.

Prueba Parcial. 2017 Octubre 10.
 Grado de Informática 2º curso. Estructura de Computadores.

Universidad Pública de Navarra.

Duración: 30 minutos.

Apellidos:

Nombre:



Puede utilizarse todo tipo de información escrita como memorias de prácticas, apuntes, hojas de referencia, etc

1. Completar el módulo fuente exa_2017.s en lenguaje ensamblador AT&T x86-32.(6 ptos)

```
### Estructura de Computadores curso 2017-18. Prueba evaluatoria 2017 Octubre 10
###
### Objetivos:
###      Manejar la codificación de datos enteros con signo
###      Estructuras de datos: puntero y array
###      Modos de direccionamientos indirectos e indexados
###      Lenguaje asm x86-32
### Algoritmo: El array lista contiene cinco números enteros negativos de tamaño dos bytes,
###             desde -5 hasta -1, siendo -5 el valor de la posición cero.
###             Copiar el contenido del array lista en el buffer.
###             Al buffer se accede indirectamente a través de la variable puntero EAX
###             El argumento de salida enviado al sistema operativo ha de ser
###             el primer valor del array lista.

## MACROS
.equ   SYS_EXIT, 1 # Código de la llamada al sistema operativo
.equ   LEN,      5 # Longitud del array y del buffer
## VARIABLES: lista y buffer
.data
lista: # Array inicializado con datos representados en HEXADECIMAL
        .quad -5
        .quad -4
        .quad -3
        .quad -2
        .quad -1

        -- --
buffer: # Reserva memoria para el buffer sin inicializar.

        -- --
## INSTRUCCIONES
## Punto de entrada

        -- --
.start:
## iniciozo el argumento de salida con el valor cero
        .quad 0

        -- --
## iniciozo la variable puntero EAX
        .quad lista
```

```

-----  

-----  

## inicializo el bucle con el número de iteracciones. Utilizar las macros.  

mov    ,%esi  

bucle:  

-----  

-----  

-----  

-----  

-----  

dec %esi  

jns bucle  

## salida  

mov _ _ _ _,%eax  

int _ _ _ _  

.end

```

◦ Cuestiones:

- Comando gdb para visualizar el contenido del buffer una vez finalizada la copia (2 pto):
 - .
 - (gdb)
- Si la etiqueta lista apunta a la dirección 0x00555438 indicar el contenido de las direcciones (2 pto):
 - .
 - 0x0055543C :
 - .
 - 0x0055543D :

Prueba Ordinaria. 2018 Diciembre 7.
 Grado de Informática 2º curso. Estructura de Computadores.
 Universidad Pública de Navarra.
 Duración: 45 minutos.

1ª PARTE (10 ptos)

- Duración: 20 minutos
- Calificación:
 1. (3 ptos) Resta de números sin signo: 0x8000 - 0x7AFF → las operaciones han de realizarse en código HEXADECIMAL exclusivamente

2. (3 ptos) Resta de números con signo: 0x8000 - 0x7AFF → las operaciones han de realizarse en código HEXADECIMAL exclusivamente

3. (3 ptos) Relacionar en una sola frase los conceptos: contador de programa, ruta de datos, ciclo de instrucción, secuenciador, microordenes, unidad aritmético lógica, microarquitectura , captura de instrucción.

2^a PARTE (10 pts)

- Duración: 25 minutos
- Calificación:
 1. (6 ptos) Desarrollar el módulo fuente *cadena_longitud.s* en lenguaje ensamblador AT&T x86-32.

```
/*
```

Programa: calcular el tamaño de una cadena de caracteres inicializada en el propio programa fuente con la frase "Hola"

Algoritmo: Implementar un bucle hasta encontrar el carácter fin de string : \0

Etiquetas: La referencia al string se realizará mediante el símbolo cadena.

Comentarios: Se ha de comentar el módulo fuente por bloques de código que tengan un sentido en lenguajes de alto nivel exclusivamente, no por líneas de código que describan una instrucción máquina.

```
*/
```

```
## Definición de MACROS
.equ SUCCESS, 0
.equ SYS_EXIT, 1
.equ FIN_CAR, '\0'
```

- Cuestiones: (4 ptos)

- Dos comando gdb para visualizar el contenido del objeto almacenado en la dirección cadena
 - (gdb)
 - (gdb)
- Comando gdb para visualizar exclusivamente el carácter fin de cadena.
 - (gdb)
- Indicar los dos comandos necesarios para compilar el programa fuente anterior mediante un toolchain manual, sin utilizar el front-end gcc.

GRUPO:
APELLIDOS:
NOMBRE:

Prueba Ordinaria. 2018 Diciembre 7.
Grado de Informática 2º curso. Estructura de Computadores.
Universidad Pública de Navarra.
Duración: 50 minutos.

3ª PARTE (10 ptos)

- Duración: 50 minutos
 - Calificación:
1. (2 ptos) En una llamada a una subrutina con 6 argumentos y una variable local al finalizar el ciclo de instrucción de la instrucción **CALL subrutina** el stack pointer apunta a la dirección 0xFFFFA0C. Calcular:
- La dirección de memoria donde se guarda la dirección de retorno
 - La dirección de memoria de la variable local
 - La dirección de memoria del 1º argumento de la subrutina

1. (4 ptos) El diagrama de bloques de la microarquitectura de la cpu de una computadora con un tamaño de palabra de 16 bits se corresponde con el de la figura en la hoja adjunta. La ISA de dicha computadora dispone de un lenguaje ensamblador que se corresponde con los mnemónicos y la sintaxis AT&T x86-32 . En la memoria principal se carga el código máquina, correspondiente a la sección de instrucciones del módulo fuente, siguiente:

```
movw $0xF000,R0  
movw R0,R1  
addw R1,R0  
subw R1,R0
```

- Si el secuenciador de la unidad de control está diseñado como una máquina de 4 estados T0,T1,T2 y T3 , indicar en la tabla adjunta las microórdenes a ejecutar en cada estado del ciclo de instrucción para cada instrucción del programa.

	T0	T1	T2	T3
mov \$0xF000,R0				
movw R0,R1				
addw R1,R0				
subw R1,R0				

1. (4 ptos) Organización de una memoria jerarquizada

- En el proceso de compilación de un programa, desde la fase inicial de edición hasta la carga del programa en un proceso en la memoria principal, la cadena de herramientas "toolchain" genera distintos espacios de memoria en los diferentes módulos del proceso de traducción de código. Rellenar la tabla adjunta con las características propias de cada espacio generado.

Herramienta	Programa	Estructura del Espacio de Memoria y Direccionamiento	Tipo de direcciones	Localización del código
Edición	Módulo Fuente	Secciones y Etiquetas	Virtual, No lineal	Mem. Secundaria: Disco duro

- Cómo estructura el controlador de memoria caché dentro de la jerarquía de memoria la memoria caché y la memoria RAM dinámica.

- Físicamente, en que consiste una celda de memoria RAM dinámica.

- Cómo sincroniza las transferencias de datos a través del bus del sistema, una memoria ram dinámica Double Data Rate DDR.

1. (3 ptos) Mecanismos de operaciones E/S

- Dibujar el diagrama de bloques del HW necesario entre una tecla del teclado y las unidades básicas de la arquitectura von Neumann de una computadora para realizar la transferencia de datos mediante el mecanismo de interrupciones.

- Dibujar el diagrama de bloques del SW necesario entre una tecla del teclado y las unidades básicas de la arquitectura von Neumann de una computadora para realizar la transferencia de datos mediante el mecanismo de interrupciones.

Chapter 20. Miaulario: Videoconferencia

20.1. Introducción

- Información
 - https://miaulario.unavarra.es/access/content/group/b15fc60b-3c66-452d-a7d9-42ec8cdab3a1/CSIE_WEB/site_csie/zoom-gida/zoom-gida_es.html
 - correo electrónico csie@unavarra.es [mailto:csie@unavarra.es]

20.2. Instalación de Zoom

- El conferenciante necesita instalarse la aplicación ZOOM en su versión básica (gratuita)
- <https://zoom.us/>
- Es necesario registrarse
- En la versión Ubuntu 18.0 desde el navegador Firefox no se puede iniciar el cliente
- Descargar el paquete zoom_amd64.deb
- comando de instalación: `dpkg -i zoom_amd64.deb`
- Abrir el cliente: `./zoom`

20.3. Guía de usuario Zoom

20.3.1. Configuración

- Testear el audio

20.4. Sesión de videoconferencia

- Desde miaulario → login → asignatura → videoconferencia
- https://miaulario.unavarra.es/access/content/group/b15fc60b-3c66-452d-a7d9-42ec8cdab3a1/CSIE_WEB/site_csie/zoom-gida/zoom-gida_es.html

IX Bibliografía

Arquitectura de Computadores

- [WS_es] William Stallings. Organización y arquitectura de computadores . Edición 7, reimpressa Pearson Prentice Hall ISBN 8489660824, 9788489660823 . 2006
- [WS_en] William Stallings. Computer Organization and Architecture: Designing for Performance. 9^a Ed Upper Saddle River (NJ) : Prentice Hall, 2013 ISBN 0-273-76919-7 . 2012
- [Randal_Bryant] Randal E. Bryant, David R. O'Hallaron. Computer Systems: A Programmer's Perspective. Addison-Wesley. 2nd Edition. 2010.
- [Patt_Henn] David A. Patterson, John L. Hennessy. Computer Organization and Design. The Hardware / Software Interface. Morgan Kaufmann. 2009. Libro Standard de la mayoría de las Universidades.
- [MoMano] Computer System Architecture, Morris Mano.

x86

- [FLAGS] https://en.wikipedia.org/wiki/FLAGS_register

Programación Ensamblador

- [PGU] [Programming from the Ground Up](http://programminggroundup.blogspot.com.es/2007/01/programming-from-ground-up.html) [<http://programminggroundup.blogspot.com.es/2007/01/programming-from-ground-up.html>] Jonathan Bartlett Edited by Dominick Bruno, Jr. Copyright © 2003 by Jonathan Bartlett Published by Bartlett Publishing in Broken Arrow, Oklahoma ISBN 0-9752838-4-7
- [ATT] [Oracle AT&T language](https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html) [https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html]
- [WikiBook] [Apuntes WikiBook:x86 Assembly: AT&T](https://en.wikibooks.org/wiki/X86_Assembly) [https://en.wikibooks.org/wiki/X86_Assembly]
- [pc_asm] Paul Carter. PC Assembly Language. Acceso libre. 2006.
- [asm_linux] Jeff Duntemann. Assembly Language Step-by-Step: Programming with Linux. Wiley Ed. 3rd Edition. 2009.
- [NASM_tuto] [Tutorial NASM tutorialspoint](https://www.tutorialspoint.com/assembly_programming/index.htm) [https://www.tutorialspoint.com/assembly_programming/index.htm]
- [NASM_bristol] [Apuntes Bristol Community College: Prog. NASM](http://www.c-jump.com/CIS77/CIS77syllabus.htm) [<http://www.c-jump.com/CIS77/CIS77syllabus.htm>]
- [i386] [i386 \(32 bits\)](http://www.cs.nyu.edu/~mwalfish/classes/ut/s13-cs439/ref/i386/toc.htm) [<http://www.cs.nyu.edu/~mwalfish/classes/ut/s13-cs439/ref/i386/toc.htm>]
- [A64] [amd64 \(64 bits\)](http://www.felixcloutier.com/x86/) [<http://www.felixcloutier.com/x86/>]
- [IAL] [Intel asm language](https://software.intel.com/en-us/articles/introduction-to-x64-assembly) [<https://software.intel.com/en-us/articles/introduction-to-x64-assembly>]: Intel oficial Vol 1 Basic Architecture
- [AMD] [AMD oficial](http://support.amd.com/TechDocs/24593.pdf) [<http://support.amd.com/TechDocs/24593.pdf>]. Vol 3. 2.3 Summary of Registers and Data Types
- [paul_carter] [Paul Carter PC Assembly Language. Acceso libre. 2006.](http://pacman128.github.io/piasm/) [<http://pacman128.github.io/piasm/>]: Netwide Assembler NASM, Intel language
- [j_dunte] [Jeff Duntemann](http://www.duntemann.com/) [<http://www.duntemann.com/>]. Assembly Language Step-by-Step: Programming with Linux. Wiley Ed. 3rd Edition. 2009.

- [irvine] [Kip R. Irvine](http://kipirvine.com/) [<http://kipirvine.com/>]. Assembly Language for x86 Processors. Pearson. 6th Edition. 2014.

Documentos de Programación de Bajo Nivel

- [ABI_i386] [ABI i386](https://www.uclibc.org/docs/psABI-i386.pdf) [<https://www.uclibc.org/docs/psABI-i386.pdf>]
- [ms_llamada] [Convención de llamada MicroSoft](https://docs.microsoft.com/es-es/cpp/build/x64-software-conventions?view=vs-2017) [<https://docs.microsoft.com/es-es/cpp/build/x64-software-conventions?view=vs-2017>]

Lenguaje de Programación C

- [c_king] [K.N.King](http://knking.com/books/c/) [<http://knking.com/books/c/>] C programming, a Modern Approach W.W. Norton 2^aEd. 2008.

Herramientas de Desarrollo de Programas

- [I386] [AS i386](https://sourceware.org/binutils/docs-2.26/as/i386_002dDependent.html#i386_002dDependent) [https://sourceware.org/binutils/docs-2.26/as/i386_002dDependent.html#i386_002dDependent]: syntax, mnemonics, register
- [GNU] [GNU Software Development](https://www.gnu.org/manual/manual.html) [<https://www.gnu.org/manual/manual.html>]
- [GAS] [GNU ASsembler](https://sourceware.org/binutils/docs-2.31/as/index.html) [[http://sourceware.org/binutils/docs-2.31/as/index.html](https://sourceware.org/binutils/docs-2.31/as/index.html)]
- [GDB] [debugger GDB](https://www.gnu.org/software/gdb/documentation/) [<https://www.gnu.org/software/gdb/documentation/>]
- [GCC] [Compilador GCC](https://gcc.gnu.org/onlinedocs/gcc/) [<https://gcc.gnu.org/onlinedocs/gcc/>]
- [CPP] [Preprocessor cpp](https://gcc.gnu.org/onlinedocs/cpp/) [<https://gcc.gnu.org/onlinedocs/cpp/>]
- [BiU] [herramientas GNU binutils](https://sourceware.org/binutils/docs-2.31/binutils/index.html) [[http://sourceware.org/binutils/docs-2.31/binutils/index.html](https://sourceware.org/binutils/docs-2.31/binutils/index.html)]: as, ld, objdump, ...
- [VIM] [Vim](https://www.vim.org/docs.php) [<https://www.vim.org/docs.php>]
- [EMACS] [Emacs](http://www.gnu.org/software/emacs/#Manuals) [<http://www.gnu.org/software/emacs/#Manuals>]

Artículos

- [jvn] [linuxvoice](https://www.linuxvoice.com/john-von-neumann/) [<https://www.linuxvoice.com/john-von-neumann/>]
- [w_uni] [wisconsin university RTL](http://www.cs.uwm.edu/classes/cs458/Lecture/HTML/ch04.html) [<http://www.cs.uwm.edu/classes/cs458/Lecture/HTML/ch04.html>]

X Glosario

Primer término

The corresponding (indented) definition.

Segundo término

The corresponding (indented) definition.

XI Colofón

Text at the end of a book describing facts about its production.