

Estructura de Computadores (240306)

Cándido Aramburu

2022-10-10

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 | 8 |
| 1.6. Apuntes | 10 |
| 1.7. Temario | 10 |
| 1.7.1. Bibliografía Basica | 11 |
| 1.7.2. Bibliografía Complementaria | 13 |
| 1.8. Profesorado | 13 |
| 1.9. Grado Informática | 13 |
| 1.10. Calendarios | 14 |
| 1.11. Ejercicios mediante resolución de problemas | 17 |
| 1.12. Prácticas | 17 |
| 1.12.1. Memorias | 17 |
| 1.13. Recursos Informáticos | 17 |
| 1.13.1. UPNA | 17 |
| 1.13.2. Estaciones de Trabajo: 32 y 64 bits | 18 |
| 1.13.3. Registrarse | 18 |
| 1.14. Grupos de Prácticas | 19 |
| 1.15. Metodología | 20 |
| 1.15.1. Distribución de créditos | 20 |
| 1.15.2. Distribución de créditos de las Prácticas | 21 |
| 1.16. Evaluación | 21 |
| 1.17. Exámenes | 21 |
| 2. Arquitectura Von Neumann | 22 |
| 2.1. Arquitectura Von Neumann | 22 |
| 2.1.1. Temario | 22 |
| 2.1.2. Contexto Histórico | 22 |
| 2.2. Institute Advanced Machine (IAS) : Arquitectura | 23 |
| 2.2.1. Referencia | 23 |
| 2.2.2. Ejemplo del Programa sum1toN | 23 |
| 2.3. Estructura de la computadora IAS | 25 |
| 2.3.1. Módulos | 25 |
| 2.3.2. Unidad Central de Proceso (CPU) | 27 |
| 2.3.3. Memorias | 28 |
| 2.3.4. Bus | 29 |
| 2.3.5. Input Output (I/O) | 30 |
| 2.3.6. Animación del Ciclo de Instrucción | 30 |
| 2.4. ISA: Arquitectura del Repertorio de Instrucciones de la máquina IAS | 30 |
| 2.4.1. Formato de los datos e Instrucciones de la Computadora IAS | 30 |
| 2.4.2. Repertorio ISA | 32 |
| 2.4.3. Interfaz ISA | 34 |

| | |
|---|----|
| 2.5. Programación en el Lenguaje Ensamblador IAS | 35 |
| 2.5.1. Estrategia del Desarrollo de un Programa en Lenguaje Ensamblador | 35 |
| 2.5.2. Codificación Binaria-Hexadecimal | 36 |
| 2.5.3. Ejemplo 1: sum1toN.ias | 37 |
| 2.5.4. Ejemplos de Programas en Lenguaje IASSim | 43 |
| 2.6. Operación de la Máquina IAS: Ruta de Datos | 43 |
| 2.7. Conclusiones | 44 |
| 3. Representación de los Datos | 45 |
| 3.1. Temario | 45 |
| 3.2. Objetivo | 45 |
| 3.3. Datos e Instrucciones: Codificación Binaria | 45 |
| 3.4. Bit, Byte, Palabra | 45 |
| 3.5. Números Enteros | 46 |
| 3.5.1. Base Decimal | 46 |
| 3.5.2. Base Binaria | 46 |
| 3.5.3. Conversión Decimal-Binaria | 47 |
| 3.5.4. Base Octal | 48 |
| 3.5.5. Calculadora | 48 |
| 3.5.6. Python | 48 |
| 3.5.7. Enteros con Signo | 49 |
| 3.6. Números Reales | 51 |
| 3.6.1. Coma Fija | 51 |
| 3.6.2. Coma Flotante | 52 |
| 3.7. Character Type | 54 |
| 3.7.1. ASCII | 54 |
| 3.7.2. Python | 57 |
| 3.7.3. Unicode UTF-8 | 57 |
| 3.8. ISO-8859-1 | 59 |
| 3.8.1. Programación en C | 59 |
| 3.8.2. Otros | 59 |
| 4. Operaciones Aritmeticas y Logicas | 60 |
| 4.1. Temario | 60 |
| 4.2. Objetivo | 60 |
| 4.3. Introducción | 60 |
| 4.4. Aritmetica Binaria | 60 |
| 4.4.1. Suma en módulo 2 (binaria) en binario puro (Nº NATURALES) | 60 |
| 4.4.2. Resta en módulo 2 (binaria) en binario puro | 62 |
| 4.4.3. Suma/Resta en módulo 2 (binaria) en complemento a 2 | 63 |
| 4.4.4. Suma en Módulo 16 (Hexadecimal) | 66 |
| 4.4.5. Resta en Módulo 16 (Hexadecimal) | 66 |
| 4.4.6. Tipos de variables en C | 67 |
| 4.5. Operaciones Logicas | 67 |
| 4.5.1. Operadores BITWISE | 68 |
| 4.6. Multiplicación | 68 |
| 4.7. Programación | 69 |
| 4.7.1. funciones matemáticas | 69 |
| 4.7.2. Aplicación | 69 |
| 4.8. Hardware | 69 |

| | |
|---|----|
| 4.8.1. Circuitos Digitales | 69 |
| 4.8.2. Unidad Aritmetico Lógica (ALU) | 69 |
| 4.8.3. Registro de flags EFLAG | 70 |
| 4.8.4. Float Point Unit-FPU | 72 |
| 5. Representación de las Instrucciones | 73 |
| 5.1. Temario | 73 |
| 5.1.1. Bibliografía | 73 |
| 5.2. Objetivos | 73 |
| 5.2.1. Requisitos | 73 |
| 5.3. Lenguajes de programación de alto nivel vs Lenguajes de Programación de bajo nivel | 73 |
| 5.3.1. Lenguajes de alto nivel | 73 |
| 5.3.2. El lenguaje máquina y el lenguaje ensamblador | 74 |
| 5.4. Elementos de una Instrucción Máquina | 74 |
| 5.4.1. Tipos de Arquitecturas de Operando: Ejemplos | 75 |
| 5.5. Instrucciones en lenguaje máquina de la arquitectura x86 | 76 |
| 5.6. Representación de las instrucciones en el lenguaje ensamblador (ASM) para computadoras en general | 76 |
| 5.6.1. Introducción | 76 |
| 5.6.2. Códigos de Operación | 77 |
| 5.6.3. Operandos: Modos de Direcciónamiento | 78 |
| 5.7. Lenguaje Intel versus Lenguaje AT&T | 81 |
| 5.7.1. Lenguajes ensamblador de la arquitectura i386/amd64 | 81 |
| 5.7.2. Sintaxis de las instrucciones en el lenguaje INTEL | 81 |
| 5.8. Operandos: Modos de Direcciónamiento | 82 |
| 5.8.1. Localización | 82 |
| 5.8.2. Modos de Direcciónamiento | 82 |
| 5.9. Programas en lenguaje ASM y lenguaje Binario | 84 |
| 6. Programación en Lenguaje Ensamblador (x86): Construcciones básicas de los lenguajes de alto nivel. | 85 |
| 6.1. Temario | 85 |
| 6.2. Introducción | 85 |
| 6.2.1. Objetivos | 85 |
| 6.2.2. Requisitos | 85 |
| 6.2.3. Referencias | 85 |
| 6.3. Estructura de la Computadora con Arquitectura Intel x86-64 | 86 |
| 6.3.1. Manuales Intel ISA x86 | 86 |
| 6.3.2. Intro | 86 |
| 6.3.3. CPU-Memoria | 86 |
| 6.3.4. Registros internos a la CPU x86 | 86 |
| 6.4. Representación de los datos en lenguaje ensamblador (ASM) para la arquitectura i386/amd64 | 89 |
| 6.4.1. Tipos de Datos | 89 |
| 6.4.2. Tamaño del operando x86 | 90 |
| 6.4.3. Alineamiento de Bytes: Big-Little Endian | 90 |
| 6.4.4. Ejemplo | 92 |
| 6.5. Repertorio de Instrucciones en lenguaje ensamblador (ASM) para la arquitectura i386/amd64: Operaciones | 92 |
| 6.5.1. Ejemplo | 92 |
| 6.5.2. Manual rápido | 92 |

| | |
|--|-----|
| 6.5.3. Manuales y Tablas | 92 |
| 6.5.4. Tipo de descripción de Códigos de Operación en el Manual de Intel | 92 |
| 6.6. Mnemónicos Básicos (Explicados) | 93 |
| 6.6.1. Operaciones aritméticas | 93 |
| 6.6.2. Extensión del signo | 94 |
| 6.6.3. Cambio de tamaño | 94 |
| 6.6.4. Operaciones Booleanas | 95 |
| 6.6.5. Procesamiento Condicional: CMP,TEST,SETcc | 95 |
| 6.6.6. Saltos | 96 |
| 6.6.7. Desplazamiento y rotación | 97 |
| 6.6.8. Cambiar el Endianess | 97 |
| 6.7. Formato de Instrucción: ISA Intel x86-64 | 97 |
| 6.8. Subrutinas | 97 |
| 6.8.1. Referencias | 97 |
| 6.8.2. Introducción | 97 |
| 6.8.3. Lenguaje C: Sentencia Función | 98 |
| 6.8.4. Anidamiento de Funciones | 100 |
| 6.8.5. Pila/Frame | 100 |
| 6.8.6. Argumentos de la subrutina | 100 |
| 6.8.7. Llamada a la subrutina | 101 |
| 6.8.8. Definición de la subrutina | 101 |
| 6.8.9. Registros a Preservar | 102 |
| 6.8.10. Retorno de la subrutina | 102 |
| 6.8.11. Estado de la pila | 103 |
| 6.9. Llamadas al Sistema Operativo | 104 |
| 6.9.1. Introducción | 104 |
| 6.9.2. Arquitectura amd64 | 105 |
| 6.9.3. Arquitectura i386 | 105 |
| 6.9.4. Códigos de llamada | 106 |
| 6.9.5. Ejemplo | 110 |
| II Unidades Básicas: Procesador Central, Unidad de Memoria, Mecanismos Entrada/Salida. | 113 |
| 7. Procesador Central | 114 |
| 7.1. Temario | 114 |
| 7.2. Refs | 114 |
| 7.3. Introducción | 114 |
| 7.4. Conjunto de Instrucciones | 115 |
| 7.4.1. Arquitectura (ISA) | 115 |
| 7.4.2. Ejemplos: Intel x86, Motorola 68000, MIPS, ARM | 115 |
| 7.5. La Computadora desde el punto de vista del programador | 115 |
| 7.5.1. Niveles o Capas de Abstracción | 115 |
| 7.5.2. Compatibilidad Software | 117 |
| 7.6. Fases de Ejecución de una Instrucción | 117 |
| 7.6.1. Estructura | 117 |
| 7.6.2. Ciclo / Diagrama / Fases | 118 |
| 7.6.3. Ejemplo: máquina IAS de Von-Neumann | 120 |
| 7.7. Microarquitectura: Unidades Funcionales | 121 |
| 7.7.1. Introducción | 121 |
| 7.7.2. Implementación del ciclo de instrucción | 121 |

| | |
|--|-----|
| 7.7.3. Estructura de la CPU | 122 |
| 7.7.4. Fase de Captación | 122 |
| 7.7.5. Perspectiva de la CPU | 123 |
| 7.7.6. Unidad de Control Microprogramada | 125 |
| 7.8. Arquitecturas CISC/RISC | 126 |
| 7.8.1. Introducción | 126 |
| 7.8.2. Tabla Comparativa | 126 |
| 7.9. Instruction Level Parallelism (ILP) | 127 |
| 7.9.1. VLIW vs Superscalar | 127 |
| 7.9.2. Pipeline (Segmentacion) | 128 |
| 7.10. Ejercicios | 130 |
| 7.11. Imagenes | 131 |
| 8. Mecanismos de Entrada/Salida | 132 |
| 8.1. Temario | 132 |
| 8.2. Bibliografia | 132 |
| 8.3. Periféricos | 132 |
| 8.3.1. Ejemplos | 132 |
| 8.3.2. Modelo | 132 |
| 8.4. Teclado | 133 |
| 8.5. Arquitectura Computadora | 133 |
| 8.5.1. Von Neumann | 133 |
| 8.5.2. Conexión CPU-E/S | 134 |
| 8.5.3. Controlador I/O | 134 |
| 8.5.4. Espacio de direcciones | 136 |
| 8.5.5. Buses | 137 |
| 8.5.6. Análisis: Portatil Lenovo - Disco Duro | 138 |
| 8.6. Programa E/S | 139 |
| 8.6.1. Módulo fuente | 139 |
| 8.7. Driver: Sistema Operativo | 139 |
| 8.7.1. Gestor E/S: jerarquía | 139 |
| 8.7.2. Código Fuente | 140 |
| 8.7.3. Concepto | 140 |
| 8.7.4. Utilización del Driver | 140 |
| 8.8. Mecanismos de Implementación de la Interfaz E/S | 141 |
| 8.8.1. Introducción | 141 |
| 8.8.2. Sincronización por Encuesta | 141 |
| 8.8.3. Sincronización por Interrupción | 141 |
| 8.8.4. Direct Memory Access (DMA) | 142 |
| 8.8.5. Channel I/O | 143 |
| 8.9. Sincronización por Interrupción | 144 |
| 8.9.1. Concepto | 144 |
| 8.9.2. Mecanismo de Interrupción | 145 |
| 8.9.3. Controlador de Interrupciones | 145 |
| 8.9.4. Gestor de Interrupciones | 147 |
| 8.9.5. Tipos de Interrupciones | 147 |
| 8.9.6. Tabla de los Vectores de Interrupciones | 148 |
| 8.10. Acceso Directo a Memoria DMA | 153 |
| 8.10.1. Funcionalidad | 153 |

| | |
|--|-----|
| 8.10.2. Transferencias | 153 |
| 8.10.3. Sincronización | 153 |
| 8.10.4. Operación del controlador DMA | 153 |
| 8.10.5. Problemas de coherencia en la memoria cache | 154 |
| 8.11. Buses | 154 |
| 8.11.1. ISA | 155 |
| 8.11.2. PCI | 156 |
| 8.11.3. North-South Bridge | 157 |
| 8.11.4. Chipset x58 | 158 |
| 8.12. Programacion de rutinas de entrada/salida | 158 |
| 8.12.1. Software jerarquico del sistema operativo | 158 |
| 8.12.2. Instruction Set Architecture | 159 |
| 8.12.3. Programación del Controlador de Interrupciones Programable | 160 |
| 8.12.4. Driver del Teclado | 161 |
| 8.12.5. paralell port | 161 |
| 8.12.6. Serial communication RS-232 | 162 |
| 8.13. Ejercicios | 163 |
| 9. Unidad de Memoria | 164 |
| 9.1. Introducción | 164 |
| 9.1.1. Temario | 164 |
| 9.1.2. Libro: William Stalling | 164 |
| 9.1.3. Historia | 164 |
| 9.1.4. Interés | 164 |
| 9.1.5. Perspectivas | 165 |
| 9.1.6. Jerarquía de Memoria | 166 |
| 9.2. Registros | 166 |
| 9.2.1. Arquitectura amd64 | 166 |
| 9.3. Memoria Principal (RAM Dinámica DRAM) | 169 |
| 9.3.1. Tipos de memoria de semicondutor | 169 |
| 9.3.2. Memoria principal semiconductora | 169 |
| 9.3.3. Organización avanzada de memorias DRAM | 178 |
| 9.3.4. Imagenes | 182 |
| 9.4. Memoria Cache | 182 |
| 9.4.1. Bibliografia | 182 |
| 9.4.2. Introducción | 182 |
| 9.4.3. Principios Basicos | 182 |
| 9.4.4. Elementos de Diseño de la Cache | 186 |
| 9.5. Memoria Virtual | 193 |
| 9.5.1. Bibliografia | 193 |
| 9.5.2. Sistemas Operativos: Gestión de la Memoria | 193 |
| 9.5.3. Memoria Virtual Segmentada | 196 |
| 9.5.4. Memoria Virtual Paginada | 203 |
| 9.5.5. Sistemas Operativos: Gestión de la Memoria | 219 |
| III Ejercicios de Teoría | 220 |
| 10. Ejercicios | 221 |
| 10.1. Lista de Ejercicios Mínima | 221 |
| 10.2. Arquitectura von Neumann | 221 |
| 10.2.1. Computadoras: IAS, ENIAC, | 221 |

| | |
|---|------------|
| 10.2.2. Interconexión CPU-Memoria | 225 |
| 10.3. Representación de Datos | 229 |
| 10.4. Operaciones Aritméticas | 233 |
| 10.5. Operaciones Lógicas | 235 |
| 10.6. Representación de las Instrucciones | 236 |
| 10.7. Programación asm | 254 |
| 10.7.1. Datos | 254 |
| 10.7.2. Modos de Direccionamiento | 255 |
| 10.7.3. Aritmética | 256 |
| 10.7.4. Saltos | 258 |
| 10.7.5. If-Then-Else | 259 |
| 10.7.6. Do-While Loops | 260 |
| 10.8. Lenguaje de Programación C | 262 |
| 10.8.1. Punteros | 262 |
| 10.9. Capítulo 4: Memoria Cache | 264 |
| 10.10. Capítulo 5: Memoria Sincrona Dinámica RAM (SDRAM) | 267 |
| 10.11. Capítulo 7: Sistemas Entrada/Salida | 270 |
| 10.12. Capítulo 8: Operating System | 274 |
| 10.13. Capítulo 12: Processor Structure and Function (Capítulo 14 en 9 ^a Ed) | 281 |
| 10.14. Capítulo 13: Reduces Instruction Set Computer (Capítulo 15 en 9 ^a Ed) | 288 |
| IV Autoevaluación Teoría | 296 |
| 11. Teoría: Cuestionario | 297 |
| 11.1. Arquitectura von Neumann | 297 |
| V Guiones de Prácticas: Programación Ensamblador x86 | 298 |
| 12. Introducción a la Programación en Lenguaje Ensamblador AT&T x86-32 | 299 |
| 12.1. Introducción | 299 |
| 12.1.1. Objetivos | 299 |
| 12.1.2. Requisitos | 299 |
| 12.2. LEEME | 299 |
| 12.3. Cuestiones | 300 |
| 12.4. Estación de Trabajo | 300 |
| 12.5. Programación sum1toN.c | 300 |
| 12.5.1. Algoritmo | 300 |
| 12.5.2. Edición del Módulo fuente: sum1toN.c | 300 |
| 12.5.3. Compilación | 301 |
| 12.5.4. Análisis de los módulos | 301 |
| 12.5.5. Ejecución | 301 |
| 12.5.6. Depuración | 302 |
| 12.5.7. Recordatorio: Documento Memoria | 304 |
| 12.5.8. Continuamos con más ejercicios | 304 |
| 12.6. Programación sum1toN.s | 305 |
| 12.6.1. Algoritmo | 305 |
| 12.6.2. Edición del Módulo fuente: sum1toN.s | 305 |
| 12.6.3. Compilación | 306 |
| 12.6.4. Ejecución | 306 |
| 12.6.5. Análisis del módulo Fuente | 306 |
| 12.6.6. Depuración | 306 |
| 12.7. Arquitectura amd64 | 307 |

| | |
|---|-----|
| 13. Representación de los Datos | 308 |
| 13.1. Introducción | 308 |
| 13.1.1. Objetivos | 308 |
| 13.1.2. Módulos fuente: características | 308 |
| 13.1.3. Requisitos | 308 |
| 13.2. LEEME | 309 |
| 13.3. Cuestiones Opcionales | 309 |
| 13.4. Registros internos de la CPU | 309 |
| 13.5. Tamaño de los datos y variables | 309 |
| 13.5.1. Algoritmo | 309 |
| 13.5.2. Edición del Módulo fuente: datos_size.s | 309 |
| 13.5.3. Compilación | 310 |
| 13.5.4. Ejecución | 310 |
| 13.5.5. Análisis del módulo fuente | 310 |
| 13.5.6. GDB: Observaciones | 311 |
| 13.5.7. GDB:Ejecución paso a paso | 311 |
| 13.6. Tamaño de los Operandos | 314 |
| 13.6.1. Edición del Módulo fuente: datos_sufijos.s | 314 |
| 13.6.2. Compilación | 315 |
| 13.6.3. Ejecución | 316 |
| 13.6.4. Análisis del módulo fuente asm | 316 |
| 13.6.5. Deducción del tamaño del operando en una instrucción asm | 316 |
| 13.6.6. GDB:Ejecución paso a paso | 316 |
| 13.7. Modos de Direcciónamiento | 317 |
| 13.7.1. Edición del Módulo fuente: datos_direccionamiento.s | 317 |
| 13.7.2. Compilación | 319 |
| 13.7.3. Ejecución | 319 |
| 13.7.4. Análisis del módulo fuente asm | 319 |
| 13.7.5. GDB: Ejecución paso a paso | 319 |
| 14. Operaciones Aritméticas y Lógicas | 321 |
| 14.1. Introducción | 321 |
| 14.1.1. Objetivos | 321 |
| 14.1.2. Conceptos de Arquitectura | 321 |
| 14.1.3. Módulos fuente | 321 |
| 14.1.4. Requisitos | 321 |
| 14.2. LEEME | 321 |
| 14.3. Cuestiones | 321 |
| 14.4. Registros internos de la CPU | 322 |
| 14.5. Operaciones Aritméticas y Lógicas con Números Enteros con Signo | 322 |
| 14.5.1. Edición del Módulo fuente: op_arit_log.s | 322 |
| 14.5.2. Compilación | 324 |
| 14.5.3. Ejecución | 324 |
| 14.5.4. Análisis del módulo fuente | 324 |
| 14.5.5. Ejecución paso a paso | 324 |
| 15. Instrucciones de Saltos Condicionales | 326 |
| 15.1. Introducción | 326 |
| 15.1.1. Objetivos | 326 |
| 15.1.2. Requisitos | 326 |

| | |
|---|-----|
| 15.2. LEEME | 326 |
| 15.3. Cuestiones | 326 |
| 15.4. Saltos Condicionales | 326 |
| 15.4.1. Algoritmo | 326 |
| 15.4.2. Edición del Módulo fuente: saltos.s | 326 |
| 15.4.3. Compilación | 329 |
| 15.4.4. Ejecución | 329 |
| 15.4.5. Análisis del módulo fuente | 330 |
| 15.4.6. Ejecución paso a paso | 330 |
| 15.5. Mnemónicos Utilizados | 331 |
| 16. Llamadas al Sistema Operativo (Kernel) | 332 |
| 16.1. Introducción | 332 |
| 16.1.1. Qué son las llamadas al sistema | 332 |
| 16.1.2. Manuales de las llamadas al sistema | 332 |
| 16.1.3. Códigos de las llamadas | 332 |
| 16.1.4. Cómo pasar los argumentos directamente al Kernel | 332 |
| 16.1.5. Como pasar los argumentos indirectamente a través de funciones libc | 333 |
| 16.2. LEEME | 333 |
| 16.3. Cuestiones | 333 |
| 16.4. Llamada Exit | 333 |
| 16.4.1. Edición del Módulo fuente:salida.c / salida.s | 333 |
| 16.5. LLamar a la librería de C desde código ensamblador | 334 |
| 16.5.1. imprimir.s: printf | 334 |
| 16.6. Llamadas al Sistema en la Arquitectura AMD64 | 335 |
| 17. Subrutinas | 337 |
| 17.1. Introducción | 337 |
| 17.1.1. Objetivos | 337 |
| 17.2. Módulo Fuente | 337 |
| 17.3. Requisitos | 337 |
| 17.4. LEEME | 337 |
| 17.5. Cuestiones | 338 |
| 17.6. Tamaño de los datos y variables | 338 |
| 17.6.1. Algoritmo | 338 |
| 17.6.2. Edición del Módulo fuente: sumMtoN.s | 338 |
| 17.6.3. Compilación | 340 |
| 17.6.4. Ejecución | 340 |
| 17.6.5. Análisis del módulo fuente | 340 |
| 18. Imágenes: Bit Map Portable | 342 |
| 18.1. Introducción | 342 |
| 18.2. Aplicación | 342 |
| 18.2.1. Ficheros incluidos | 342 |
| 18.2.2. Ejemplo | 342 |
| 18.3. Formato BMP | 343 |
| 18.3.1. Codificación | 343 |
| 18.3.2. Mapa de memoria | 343 |
| 18.3.3. Fichero | 344 |
| 18.4. Módulo Fuente bitmap_gen_test.c | 344 |
| 18.4.1. Descripción | 344 |

| | |
|---|-----|
| 18.4.2. Funciones | 344 |
| VI Hojas de Referencia Rápida | 345 |
| 19. Programación Ensamblador AT&T x86 | 346 |
| 19.1. Programas x86-32 | 346 |
| 19.1.1. Programa Minimalista | 346 |
| 19.1.2. Ejemplo Básico | 347 |
| 19.2. Directivas Assembler AS | 349 |
| 19.3. Repertorio de Instrucciones Ensamblador | 350 |
| 19.3.1. TRANSFERENCIA | 350 |
| 19.3.2. ARITMÉTICOS | 351 |
| 19.3.3. LÓGICOS | 352 |
| 19.3.4. MISCELÁNEOS | 353 |
| 19.3.5. SALTOS (generales) | 353 |
| 19.3.6. SALTOS Sin Signo (Cardinal) SALTOS Con Signo (Integer) | 353 |
| 19.3.7. FLAGS (ODITSZAPC) | 354 |
| 19.4. Registros | 354 |
| 19.4.1. Visión completa | 354 |
| 19.4.2. Registros visibles al programador | 356 |
| 19.4.3. Compatibilidad 32-64 | 357 |
| 19.4.4. Control Flag Register | 357 |
| 19.5. GDB | 360 |
| VII Autoevaluación Prácticas | 362 |
| 20. Prácticas: Cuestionario | 363 |
| 20.1. Práctica 1 ^a : Introducción a la Programación en Lenguaje Ensamblador AT&T x86-32 | 363 |
| 20.1.1. Cuestiones teóricas | 363 |
| 20.1.2. Cuestiones prácticas | 363 |
| 20.2. Práctica 2 ^a : Representación de los Datos | 364 |
| 20.2.1. Módulo datos_size.s | 364 |
| 20.2.2. Módulo datos_sufijos.s | 364 |
| 20.2.3. Módulo datos_direccionamiento.s | 364 |
| 20.3. Práctica 3 ^o : Operaciones Aritmetico-Lógicas e Instrucciones de Salto Condicionales | 364 |
| 20.3.1. Módulo op_arit_log.s | 364 |
| 20.3.2. Módulo saltos.s | 365 |
| 20.4. Práctica 4: LLamadas al Sistema Operativo | 365 |
| 20.4.1. Módulo syscall_write_puts.c | 365 |
| 20.4.2. Módulo syscall_write_puts.s | 367 |
| 20.5. Práctica 5: LLamadas a una Subrutina | 367 |
| 20.5.1. Módulo sumMtoN_aviso.c | 367 |
| 20.5.2. Módulo sumMtoN_aviso.s | 368 |
| 20.6. Práctica 6: Imagen Bit Map Portable | 368 |
| 20.6.1. Programación en C | 368 |
| 20.6.2. Programación en ASM | 368 |
| 20.6.3. GDB | 368 |
| VIII Apéndices | 369 |
| 21. Arquitectura de una Computadora | 370 |
| 21.1. Estructura de la Computadora | 370 |
| 21.1.1. Contexto | 370 |
| 21.1.2. Arquitectura HW | 370 |

| | |
|--|-----|
| 21.1.3. CPU | 370 |
| 21.1.4. Memoria | 372 |
| 21.2. Instruction Set Architecture (ISA) | 374 |
| 21.2.1. Ejemplos: Intel x86, Motorola 68000, MIPS, ARM | 375 |
| 21.3. Procesadores Intel con arquitectura x86 | 375 |
| 21.3.1. Nomenclatura | 375 |
| 22. RTL Register Transfer Language | 377 |
| 22.1. Lenguaje RTL | 377 |
| 22.1.1. Introducción | 377 |
| 22.1.2. Registros | 377 |
| 22.1.3. Símbolos | 379 |
| 22.1.4. Sentencias RTL | 379 |
| 22.1.5. Ejemplos RTL con expresiones aritmético-lógicas | 380 |
| 23. Programas ensamblador IASSim | 381 |
| 23.1. Ejemplo 1: sum1toN.ias | 381 |
| 23.1.1. Ejemplo 2: Producto/Cociente | 382 |
| 23.1.2. Ejemplo 3: Vectores | 386 |
| 24. Simulador IASSim | 391 |
| 24.1. Máquina Virtual Java JVM | 391 |
| 24.2. Simulador IAS | 391 |
| 24.3. Simulación/Depuración | 392 |
| 25. Lenguajes de programación de Alto y Bajo Nivel | 394 |
| 25.1. Lenguajes de programación de alto nivel vs lenguajes de bajo nivel | 394 |
| 25.2. Ejemplo sum1toN en distintos lenguajes de Programación | 394 |
| 26. Lenguajes de programación en Ensamblador | 399 |
| 26.1. Manuales de referencia | 399 |
| 26.1.1. Lenguaje Intel | 399 |
| 26.1.2. lenguaje AT&T | 399 |
| 26.1.3. Características arquitectura i386 | 399 |
| 26.1.4. Assembler (Traductor Ensamblador): Directivas | 399 |
| 26.1.5. Discusión por qué ASM AT&T | 399 |
| 26.1.6. TRANSFERENCIA | 399 |
| 26.1.7. ARITMÉTICOS | 400 |
| 26.1.8. LÓGICOS | 402 |
| 26.1.9. MISCELÁNEOS | 402 |
| 26.1.10. SALTOS (generales) | 402 |
| 26.1.11. SALTOS Sin Signo (Cardinal) SALTOS Con Signo (Integer) | 403 |
| 26.1.12. FLAGS (ODITSZAPC) | 403 |
| 26.1.13. Sufijos | 403 |
| 26.2. Intel x86-32 /i386 | 404 |
| 26.2.1. sum1toN.s | 404 |
| 26.2.2. hola_mundo.s | 404 |
| 26.2.3. hola_mundo: Código Máquina Binario | 405 |
| 26.3. Intel x86-64 / AMD 64 | 408 |
| 26.3.1. sum1toN.s | 408 |
| 26.3.2. Hola Mundo | 409 |
| 26.3.3. Miscellaneous | 410 |
| 26.3.4. sum1toN.s: lenguaje intel | 411 |

| | |
|--|-----|
| 26.4. Arquitectura ARM | 412 |
| 26.4.1.Hola Mundo | 412 |
| 26.4.2. ISA | 412 |
| 26.5. Motorola 68000 | 413 |
| 26.5.1. Hola Mundo | 413 |
| 26.5.2. ISA | 413 |
| 26.6. Arquitectura MIPS | 414 |
| 26.6.1. ISA | 414 |
| 27. Toolchain: Cadena de Herramientas en el proceso de compilación | 416 |
| 27.1. Toolchain | 416 |
| 27.2. Proceso de compilación de un programa en lenguaje C | 416 |
| 27.3. Traductores del proceso de ensamblaje | 417 |
| 27.4. Assembler "as" | 418 |
| 27.4.1. Directivas | 418 |
| 28. Practicando la Programación desde el principio | 419 |
| 28.1. Documentación: guiones, bibliografía, apuntes | 419 |
| 28.2. Plataforma de Desarrollo | 419 |
| 28.2.1. Herramientas | 419 |
| 28.2.2. Programación online | 420 |
| 28.2.3. Referencias | 420 |
| 28.3. Documento Memoria: Contenido y Formato | 420 |
| 28.3.1. Contenido | 420 |
| 28.3.2. Formato | 421 |
| 28.3.3. Entrega del Documento Memoria | 421 |
| 28.4. Evaluación | 421 |
| 28.5. Programación | 422 |
| 28.5.1. Metodología | 422 |
| 28.6. Compilación | 422 |
| 28.6.1. Módulo fuente en lenguaje C | 422 |
| 28.6.2. Punto de entrada al programa: main vs _start | 422 |
| 28.6.3. Fases de la compilación | 423 |
| 28.6.4. Toolchain | 424 |
| 28.6.5. módulo fuente en lenguaje ensamblador | 424 |
| 28.7. Depuración | 425 |
| 28.8. Errores Comunes | 425 |
| 28.8.1. gcc | 425 |
| 28.8.2. gdb | 425 |
| 28.9. Programar y Depurar desde cero | 425 |
| 28.9.1. Empezando: ASM | 426 |
| 28.9.2. Empezando: C | 428 |
| 29. Llamadas al Sistema Operativo | 430 |
| 29.1. Introducción | 430 |
| 29.2. Manuales de las llamadas | 431 |
| 29.3. Llamada INDIRECTA | 431 |
| 29.4. LLamada DIRECTA | 431 |
| 29.4.1. Argumentos de la llamada directa | 431 |
| 29.4.2. Códigos de la llamada directa | 432 |
| 29.5. Ejemplos: lenguaje C | 432 |

| | |
|---|-----|
| 29.6. Ejemplos: ASM INDIRECTO | 432 |
| 29.7. Ejemplos: ASM DIRECTO | 433 |
| 29.8. Línea de Comandos | 433 |
| 29.8.1. Procedimiento | 433 |
| 29.8.2. Stack Initialization | 434 |
| 29.8.3. Rutina principal con Retorno | 435 |
| 29.8.4. Ejercicios: suma_linea_com.s ,maximum_linea_com.s | 436 |
| 30. Pila | 439 |
| 30.1. Concepto | 439 |
| 30.2. Anchura | 439 |
| 30.3. Frame: frame pointer y stack pointer | 440 |
| 30.4. Instrucciones Ensamblador Push-Pop | 441 |
| 30.4.1. Anidamiento de llamadas | 442 |
| 31. Lenguaje de Programación C | 443 |
| 31.1. Introducción | 443 |
| 31.2. Casting | 443 |
| 31.2.1. Concepto | 443 |
| 31.2.2. Ejemplo | 443 |
| 31.3. Puntero | 443 |
| 31.3.1. Referencias | 443 |
| 31.3.2. Introducción | 443 |
| 31.3.3. Concepto | 444 |
| 31.3.4. Módulo Ilustrativo | 448 |
| 31.3.5. Declaración | 449 |
| 31.3.6. Operador Dirección | 449 |
| 31.3.7. Operador Indirección o Dereferencia | 450 |
| 31.3.8. Ejemplo | 450 |
| 31.3.9. Aplicaciones de los punteros | 450 |
| 31.3.10. String Literal | 451 |
| 31.3.11. Puntero a Puntero | 452 |
| 31.3.12. String Variable | 452 |
| 31.3.13. Funciones | 453 |
| 32. FPU x87 | 454 |
| 32.1. FPU x87 | 454 |
| 32.1.1. Resumen | 454 |
| 32.1.2. Refs | 455 |
| 33. Exámenes de Cursos Anteriores | 456 |
| 33.1. Año 2018 | 456 |
| 33.1.1. Noviembre | 456 |
| 33.2. Año 2017 | 458 |
| 34. Miaulario: Videoconferencia | 467 |
| 34.1. Introducción | 467 |
| 34.2. Instalación de Zoom | 467 |
| 34.3. Guía de usuario Zoom | 467 |
| 34.3.1. Configuración | 467 |
| 34.4. Sesión de videoconferencia | 467 |
| IX Bibliografía | 468 |
| X Glosario | 470 |

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](#) 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)

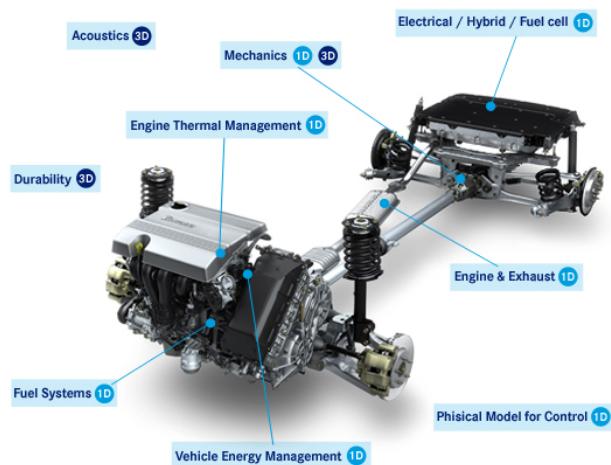


Figure 1. Estructura del Automóvil

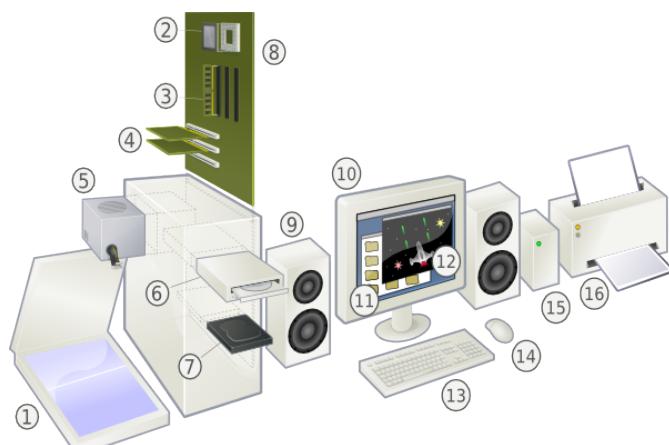


Figure 2. Personal Computer

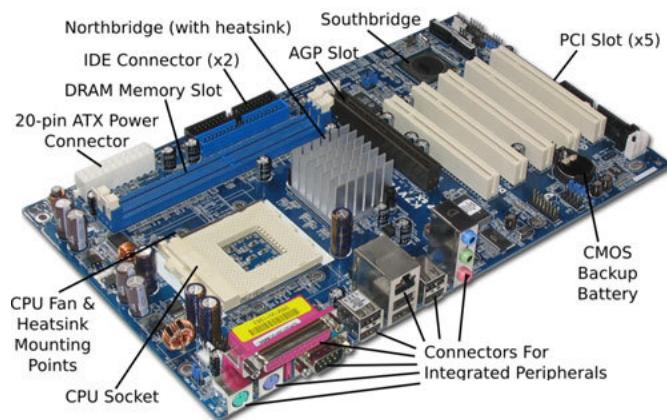


Figure 3. Placa Base

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

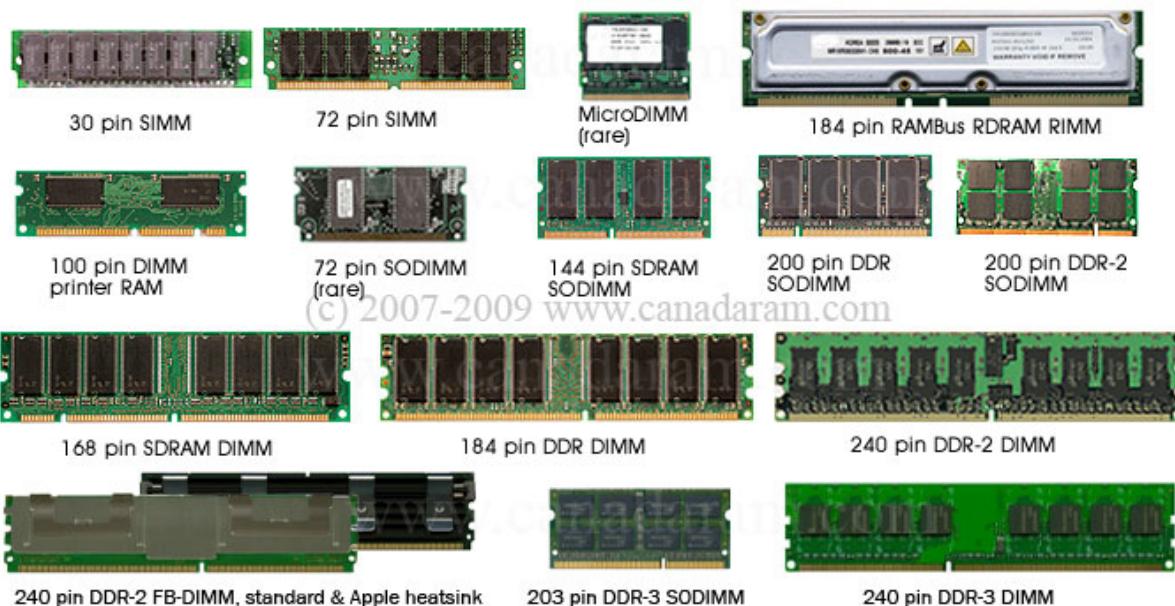


Figure 5. Memoria DRAM



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



Figure 7. Periféricos: Disco Duro Seagate



Figure 8. Periféricos: Memoria M2M

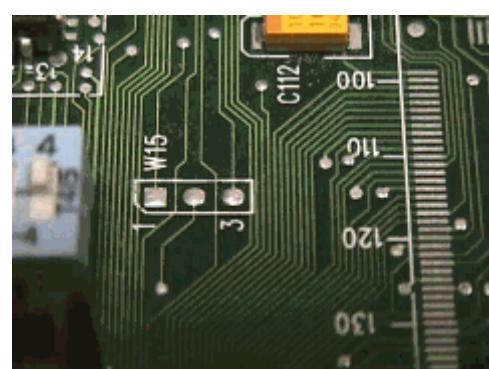


Figure 9. Bus de la Placa Base

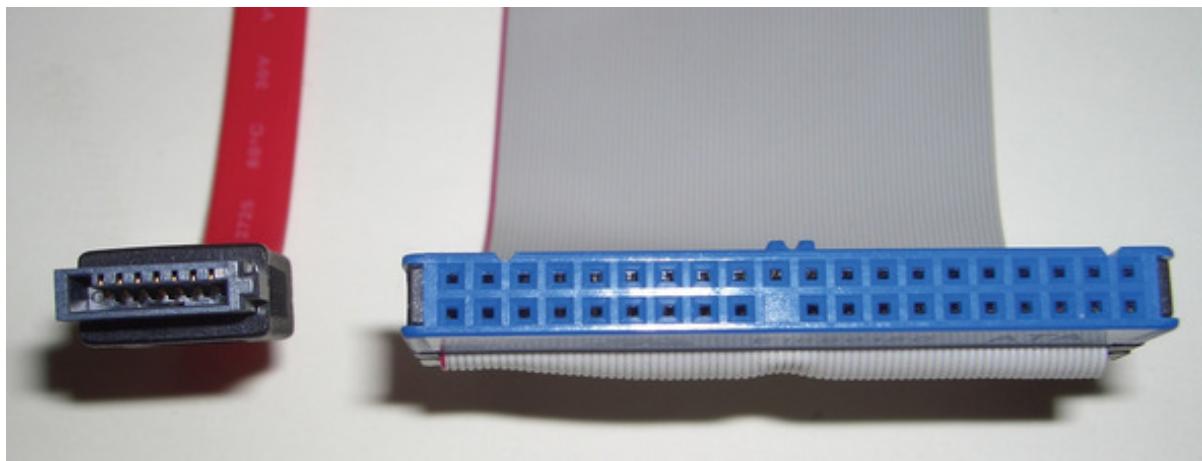


Figure 10. Bus Cableado

1.4. Lenguajes de Programación y Lenguaje Máquina

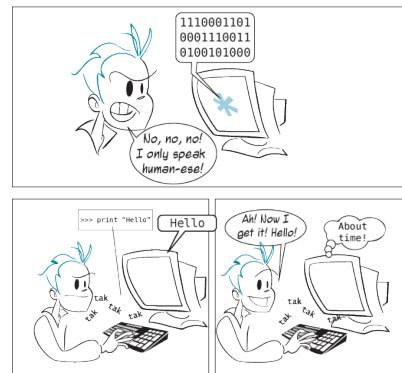


Figure 11. Lenguaje Máquina: Binario

- 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

```

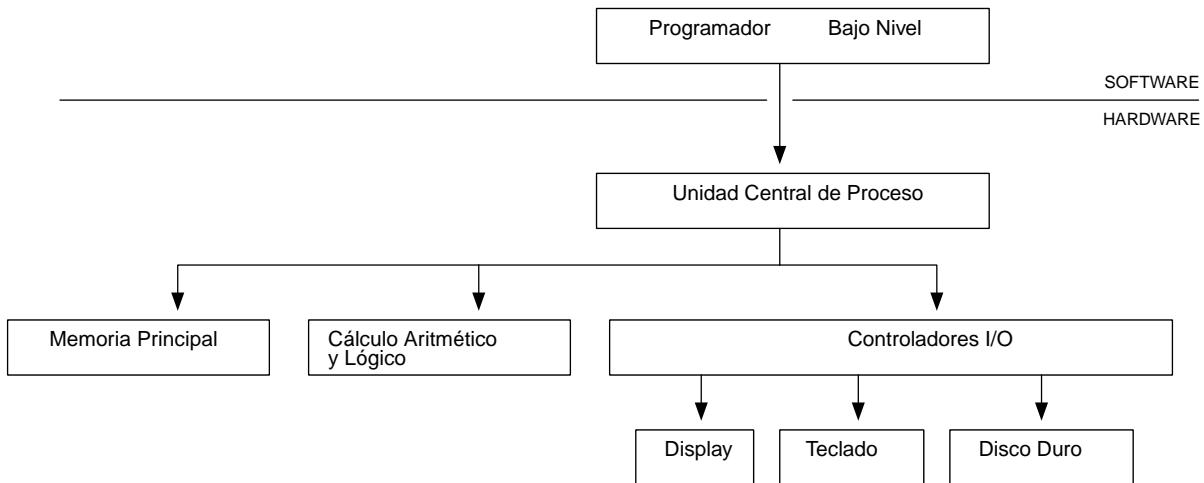
# [2] begin
0000000004001a0 <PASCALMAIN>:
4001a0: 55                      push   %rbp
4001a1: 48 89 e5                mov    %rsp,%rbp
4001a4: 48 83 ec 10             sub    $0x10,%rsp
4001a8: 48 89 5d f8             mov    %rbx,-0x8(%rbp)
4001ac: e8 b7 3e 01 00          callq  414068 <FPC_INITIALIZEUNITS>
# [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

```

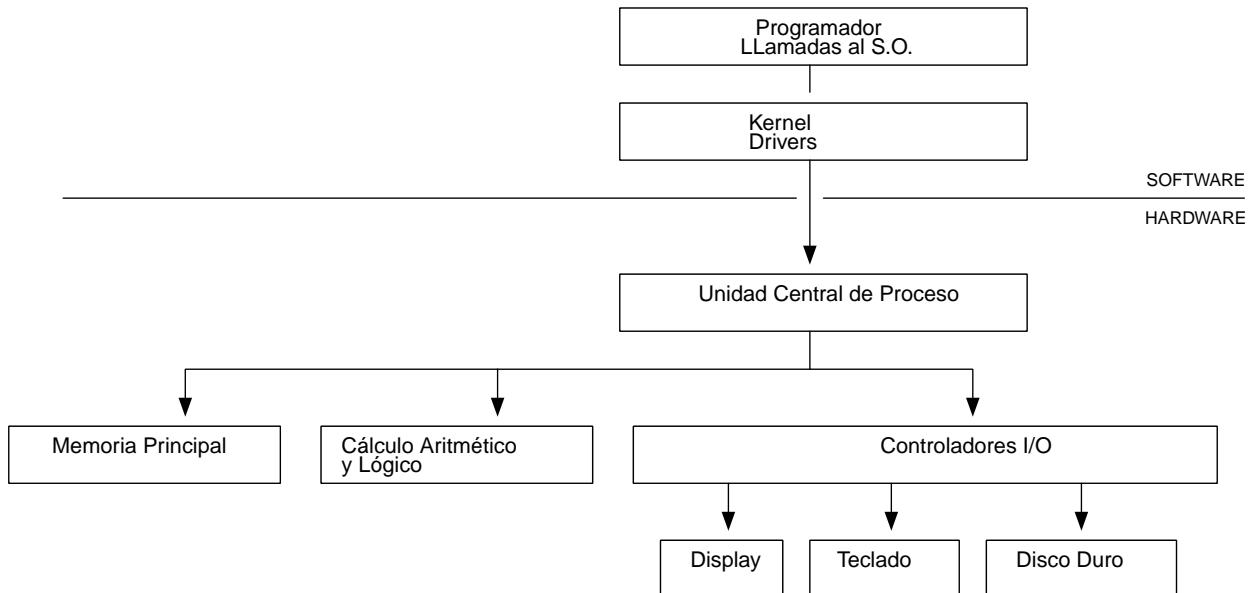
- Lenguaje de programación humano (texto) vs Lenguaje de programación máquina (binario)
- Traducir → Compilar
- Interpretar
- Lenguajes de programación: javascript, ruby, python, java, bash, C, assembly
 - Niveles de abstracción: humano/máquina

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. Apuntes

- [Apuntes On Line](#)



Figure 12. Apuntes On Line

1.7. Temario

- [Web Estructura de Ordenadores](#)

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

1.7.1. Bibliografia Basica

- Teoría
 - [William Stallings](#)

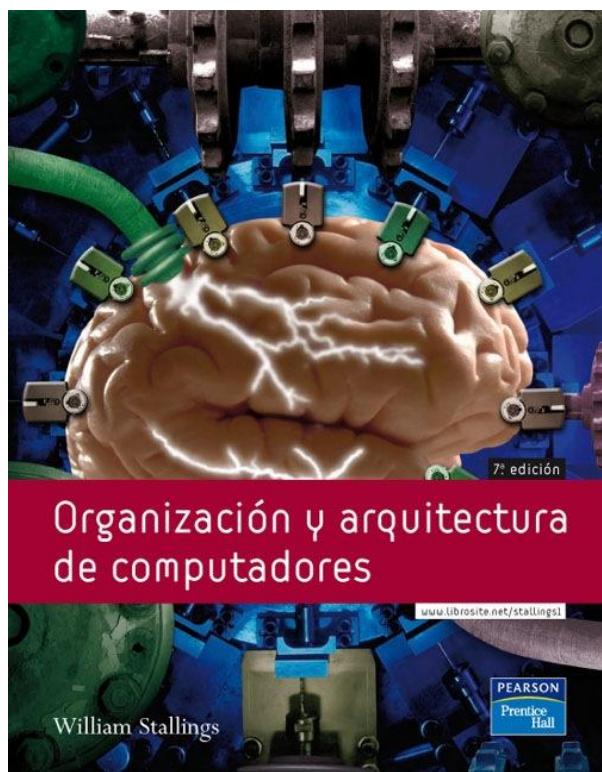


Figure 13. William Stallings Book

Organización y arquitectura de computadores .William Stallings
Edición 7, reimprima Pearson Prentice Hall
ISBN 8489660824, 9788489660823 . 2006

Computer Organization and Architecture: Designing for Performance.
William Stallings

- COA 7^a
- 11^o Ed 2019
- Prácticas:

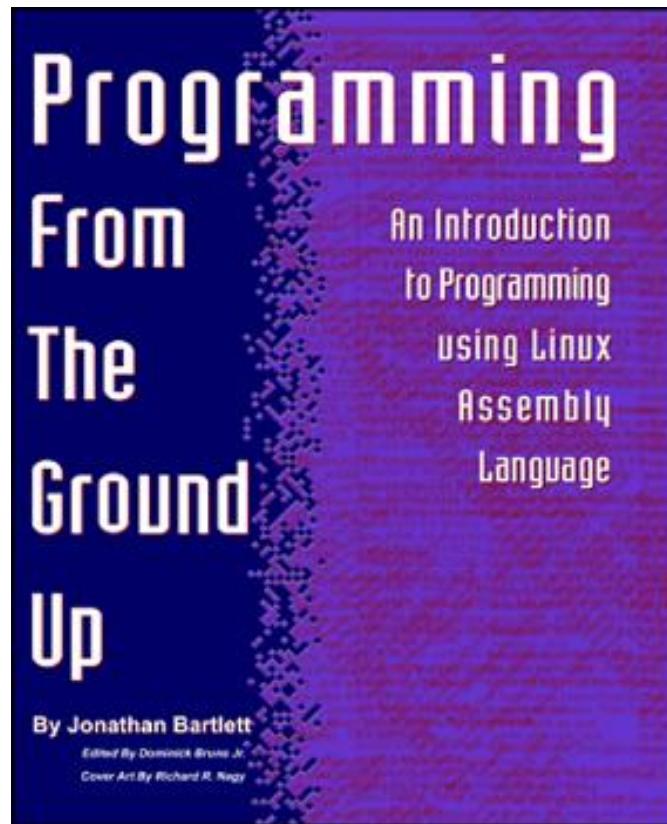


Figure 14. Programming Assembly

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
 - PGU book home
 - pdf
- El ensamblador... pero si es muy fácil: 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 la sesiones de trabajo.

1.7.2. Bibliografía Complementaria

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

1.8. Profesorado

- Cándido Aramburu Mayoz.
 - Doctor Ingeniero Telecomunicación. Profesor Titular de Universidad.
 - Edificio los Tejos, planta 2^a. Despacho 2028.
 - correo electrónico interno: a través del servidor Miaulario.
 - [PDI Profesorado](#)
 - Tutorías

Tutoría online (cita previa) y Presencial

Lugar: Edificio Los Tejos 2^a Planta, Despacho 2028 (Prof. Cándido Aramburu)

Lunes : 9:30-11:00-12:00-13:30

Jueves: 9:30-11:00-12:30-13:30

- 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](#)
- Carlos Juan de Dios Ursúa
 - 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

1.9. Grado Informática

- [Grado Informatica](#)
 - 200 - Escuela Técnica Superior de Ingeniería Industrial, Informática y de Telecomunicación
 - 240 - Graduado o Graduada en Ingeniería Informática por la Universidad Pública de Navarra

1.10. Calendarios

- [Calendario administrativo](#)
 - Teoría
 - Grupo1 GG Jueves 15:00 A125
 - Grupo2 GG Jueves 17:00 A234
 - Grupo91 GG Lunes 15:00 A135
 - Prácticas
 - [Calendario Aulas Informática 1º Cuatrimestre](#)
 - [Calendario Aulas Informática 2º Cuatrimestre](#)
 - 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

- [Horarios y Aulas](#)

- Teoría-Prácticas

| | Lunes | Martes | Miércoles | Jueves | Viernes |
|-------|----------|--------|------------|---------|---------|
| 15:00 | T91-A135 | | PG1G2-A303 | T1-A125 | |
| 17:00 | P91-A327 | | PG2-A306 | T2-A234 | |
| 19:00 | PG1-A336 | | | | |

- Prácticas

| | |
|----------------|------------------|
| A336 P91 17:00 | A303 PG1G2 15:00 |
| P327 PG1 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

- * 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

Table 1. Calendario Semestre Otoño 2022

| 2022 / 2023 | LUNES | MARTES | MIÉRCOLE S | JUEVES | VIERNES | SÁBADO | DOMINGO |
|----------------|--|--------|--|---|---|--------|---------|
| | 29 | 30 | 31 | 1 | 2 | 3 | 4 |
| SEPTIEMB RE | 1. Saioa (T1, T2*) G91 (A135) 15:00 5 | 6 | 7 | Sesión 1 (T1, T2*) G1 (A125) 15:00 G2 (A234) 17:00 8 | 9 | 10 | 11 |
| | 2. Saioa (T2) 12 | 13 | 14 | Sesión 2 (T2) 15 | 16 | 17 | 18 |
| | 3. Saioa (T3*)4. Saioa (A227)(T3/T 4*) 19 | 20 | Sesión 3 (T3*) G1 (A318) 15:00 G2 (A318) 17:00 21 | Sesión 4 (T3/T4*) 22 | 23 | 24 | 25 |
| | 5. Saioa (T5) 6. Saioa(A227 (T5/T6*) 26 | 27 | Sesión 5 (T5) G1 (A318) 15:00 G2 (A318) 17:00 26 | Sesión 6 (T5/T6*) 29 | 30 | 1 | 2 |
| OCTUBRE | Práctica 1 PG1 (A336) 19:00 PG91(A327) 17:00 3 | 4 | Práctica 1 PG1G2 (A303) 15:00 PG2 (A306) 17:00 5 | Sesión 7 (T6) 6 | 7 | 8 | 9 |
| | 10 | 11 | 12 | Sesión 8 (T6) 13 | Examen parcial 1 (A125 10:00)(T1- T6*) Programaci ón papel 14 | 15 | 16 |
| | Práctica 2 17 | 18 | Práctica 2 19 | Sesión 9 (T6) 20 | 21 | 22 | 23 |

| 2022 / 2023 | LUNES | MARTES | MIÉRCOLE S | JUEVES | VIERNES | SÁBADO | DOMINGO |
|---------------|------------------|--------|------------------|---|--|--------|---------|
| | Práctica 3 24 | 25 | 26 | Sesión 10 27 | Examen de prácticas 1 Aula A307-A308-A015 8:00AM (P1-P2) 28 | 29 | 30 |
| | 31 | 1 | Práctica 3 2 | Sesión 11 3 | 4 | 5 | 6 |
| NOVIEMBR E | Práctica 4 7 | 8 | 9 | Sesión 12 10 | 11 | 12 | 13 |
| | 14 | 15 | Práctica 4 16 | Sesión 13 17 | 18 | 19 | 20 |
| | Práctica 5 21 | 22 | Práctica 5 23 | Sesión 14 24 | 25 | 26 | 27 |
| | Práctica 6 28 | 29 | Práctica 6 30 | Sesión 15 1 | 2 | 3 | 4 |
| DICIEMBRE | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| ENERO | 9 | 10 | 11 | Examen parcial 2 (T6-T9) Examen Prácticas 2 12 | 13 | 14 | 15 |

CAUTION

En el primer parcial de teoría del tema 6 NO entran las subrutinas y las llamadas al sistema, el resto sí.

- 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).

- Semanas del curso :
 - 15 semanas: Duración del curso 2 de Septiembre al 17 de Diciembre
- [calendario exámenes](#)
 - [otoño 2023](#)

1.11. Ejercicios mediante resolución de problemas

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

1.12. Prácticas

1.12.1. 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.

1.13. Recursos Informáticos

1.13.1. UPNA

servicio informático

- Servicio Informático UPNA
 - Reserva Aulas
 - Aulas VDI
 - Manual Acceso Remoto VDI
 - Escritorio Virtual
- Laboratorio Remoto de Computación
 - acceso general → ¿?

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](#) → 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](#): Instancias DE1-SoC. Interfaz Web.
 - Se instalará su acceso a través de miaulario mediante el botón EDA.
 - Herramientas de diseño dispositivos digitales

1.13.2. 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://www.gnu.org/software/\[GNU\]](https://www.gnu.org/software/[GNU]) mediante la distribución distribución [Ubuntu](#). 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, [Virtualio](#), 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..)

1.13.3. Registrarse

miaulario

- [Miaulario](#)

Github

- Es necesario tener una cuenta abierta en la plataforma de repositorios [Github](#)

Google

- [Google account](#)
- [Navegador Google Chrome](#): permite seleccionar el idioma

1.14. Grupos de Prácticas

Table 2. Grupos de Prácticas

| Grupo P1 | Grupo P2 | Grupo P1P2 |
|------------------------------------|----------------------------------|------------------------------|
| Alén Urra, Saúl | Elcid Beperet, Iker | Goikoetxea Macua, Jon |
| Alonso Gómez, Ana | Fernandez Illera, Iker | Goñi Lara, Iker |
| Altamirano Trujillo, Ruth Nazareth | Fernández Picorelli, Marina | Isturitz Sesma, Eneko |
| Álvarez Alonso, Markel | Flamarique Arellano, Aritz | Orradre Berdusán, Joaquín |
| Andreu Mangado, Alvaro | Fortun Iñurrieta, Lucas Nicolas | Pascal Alegría, 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 David | Solaegui Garralda, Beñat |
| Bayona Restrepo, Karol Juliana | Hualde Romero, Israel | Taberna Maceira, Alain |
| Bellera Alsina, Alberto | Iribarren Ruiz, Beñat | Zamboran Maldonado, Andrea |
| Cardenas Curicho, Sol. Samanta | Juanotena Ezkurra, Joseba | Martínez Sesma, Álvaro |
| Calleja Pascual, Vidal | Juárez Jiménez, Adrián | Meléndez Uriz, Alejandro |
| Cascan Ustarroz, Eduardo | Labat García, Pablo | Mellado Ilundain, Iñaki |
| Cerezo Uriz, Iñaki | Labiano Garcia, Martin Javier | Molina Puyuelo, Alejandro |
| Chacón Flores, Malena Paola | Larrayoz Díaz, Asier | Monreal Ayanz, Ander |
| Cordido Pérez, Elena | Larrayoz Urria, Carlota | Moral Garcia, Haizea |
| Couceiro Eizaguirre, Javier | Latasa Sancha, Iván | Oroz Azcárate, Ángel |
| Cuello Tejero, Jorge | Liébana Revuelta, Diego | Pérez Álvarez, Ángel |
| de la Ossa Goñi, Miguel | Longás Saragüeta, Endika Jacinto | Redrejo Fernández, Misael |
| Díaz Ochoa, Alex | Lumbreras Corredor, Imanol | Saiz Larraz, Eder |
| Dobromirova Karailieva, Zhaklina | Martínez Arpón, Alain | Santos Garzon, Jaider Felipe |
| Ezponda Igea, Eduardo | Martínez de Goñi, Unai | Sanz Sanz, Iván |
| Goicoechea Elío, María | | Sola Alba, Alejandro |
| Ridruejo Mayor, Lara | | Tellechea Zamanillo, Sergio |
| | | Turrillas Remiro, Ethan |
| | | Urroz Velasco, Ibai |
| | | Vadillo Navarro, Asier |

| Grupo P1 | Grupo P2 | Grupo P1P2 |
|----------|----------|-----------------------------|
| | | Yaniz Ibañez, Asier |
| | | Yarhui Sarate, Yerson Caleb |
| | | Zheng , Yushan |

1.15. Metodología

- Teoría, Ejercicios, Prácticas y Exámenes.

Table 3. 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 |
| 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)

1.15.1. 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

1.15.2. 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.

1.16. Evaluación

• Web Estructura de Ordenadores

- prácticas: 2 pruebas parciales. Primera prueba parcial el 28 de Octubre (prácticas 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 (Temas 1 y 6. Del tema 6 no entran subrutinas y llamadas al sistema) . Segunda prueba parcial en la convocatoria final ordinaria el 12/01/2023 (Temas 6-7-8 y 9). 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.

1.17. 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. Del tema 6 no entran subrutinas y llamadas al sistema.

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

convocatoria ordinaria: 2º parcial (Temas 6,7,8 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](#)

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)

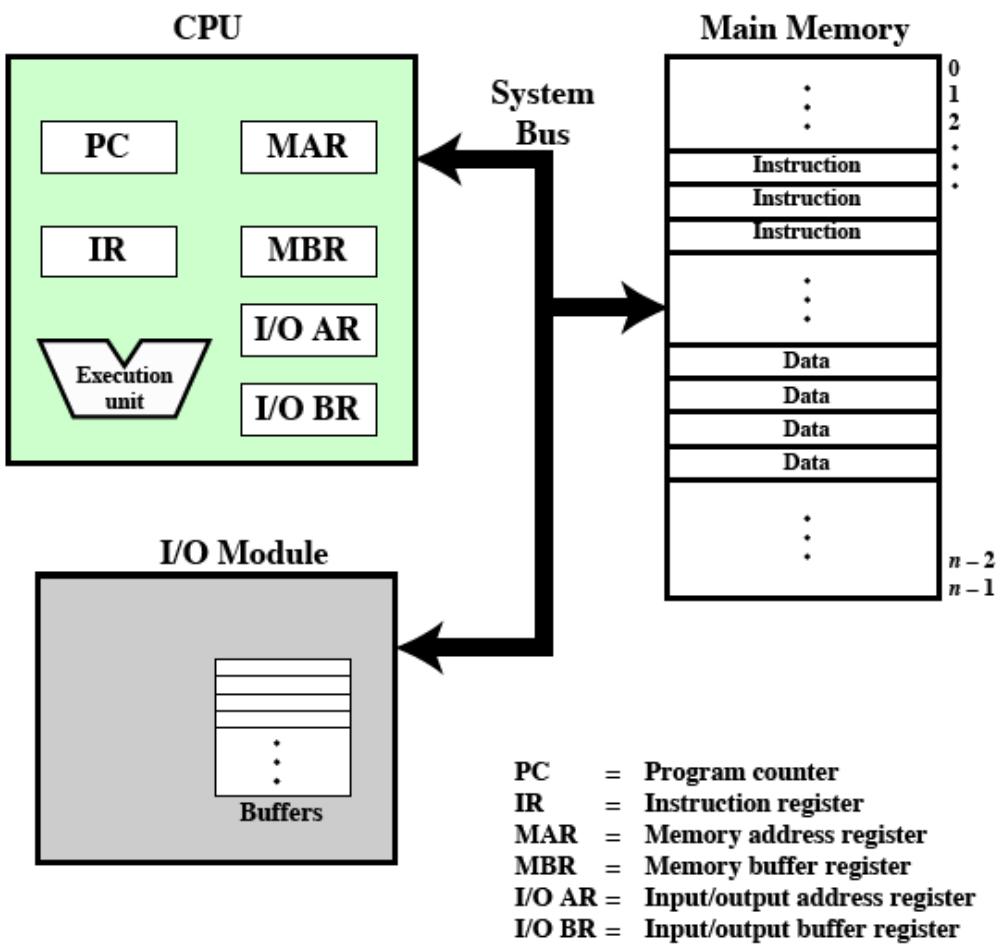


Figure 16. Arquitectura de la máquina IAS

- Arquitectura Interna de la CPU : Microarquitectura

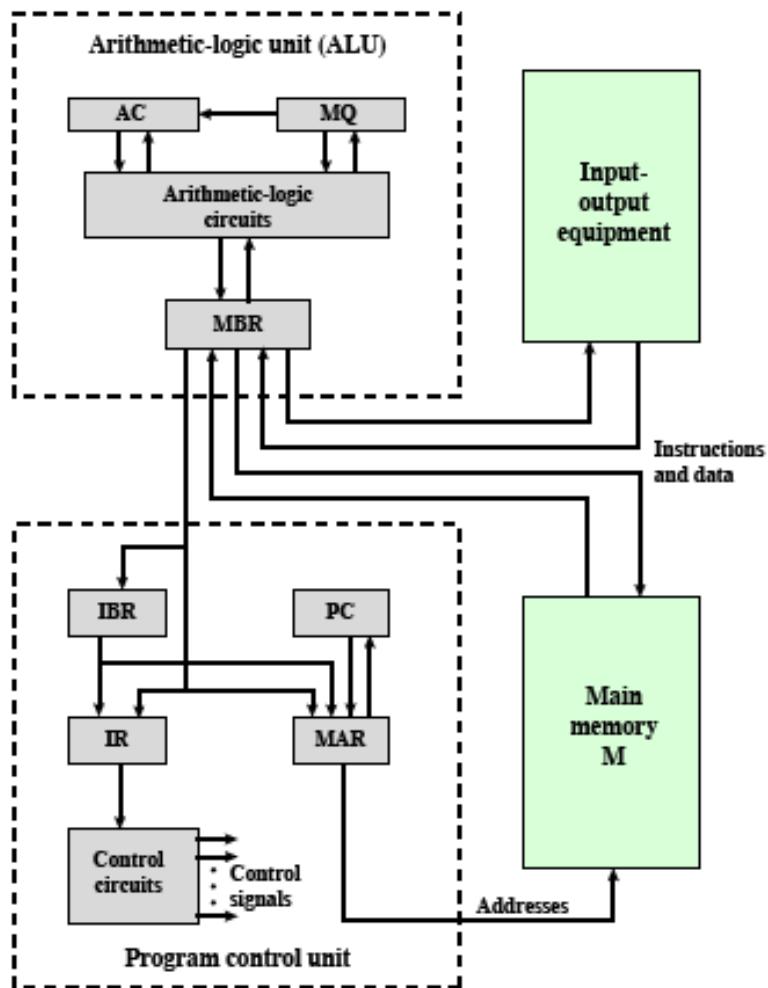


Figure 2.3 Expanded Structure of IAS Computer

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**.

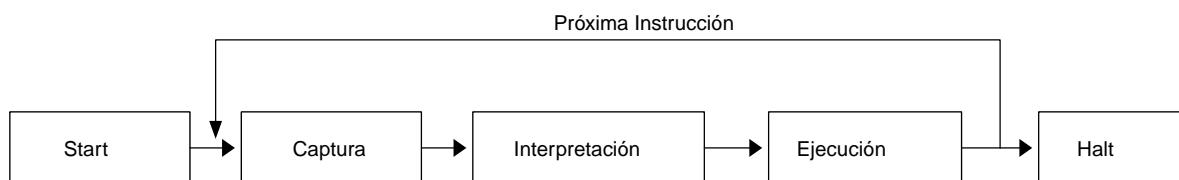


Figure 18. Ciclo de Instrucción

- 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. Direccionamiento 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.

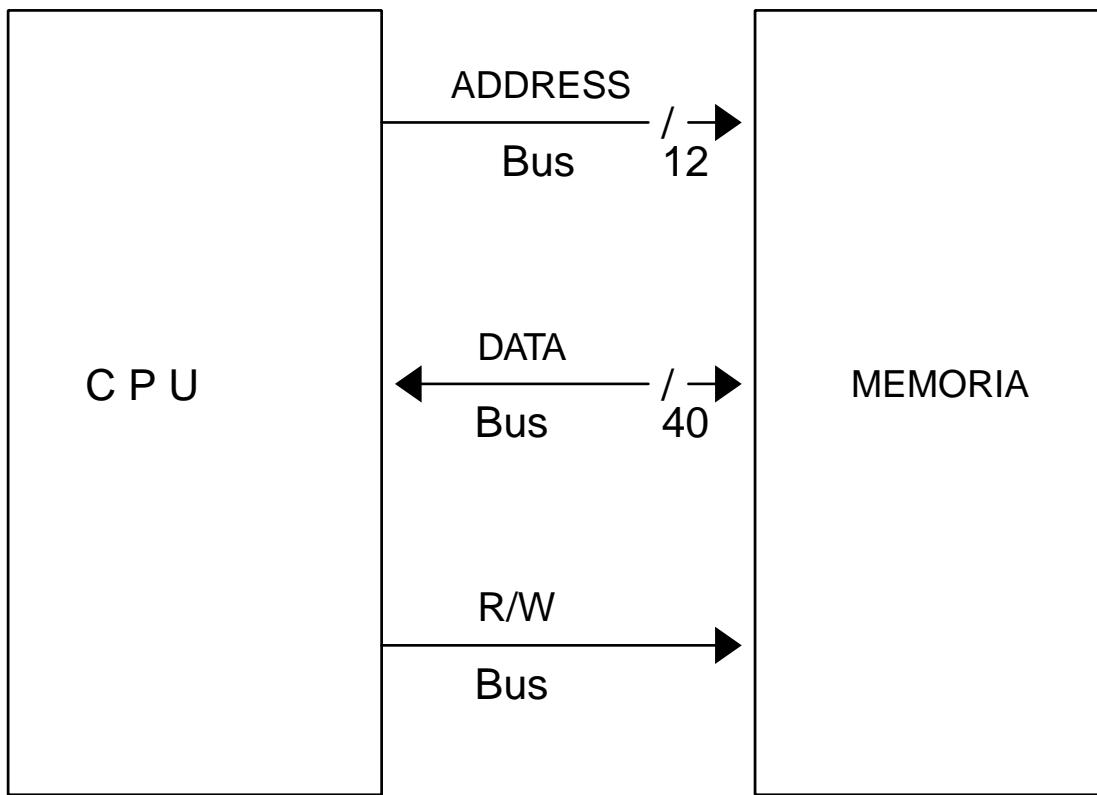


Figure 20. Conexión CPU-Memoria Principal

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](#)

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*

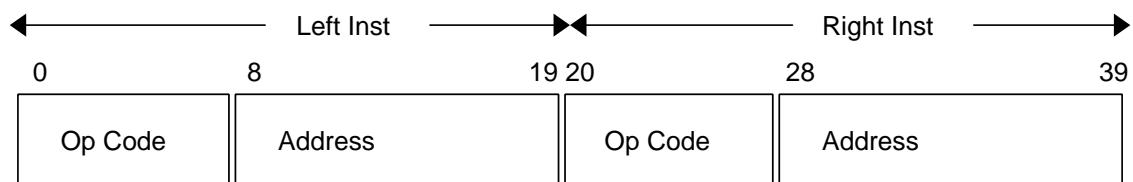


Figure 22. Formato de las instrucciones

- 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 diseña 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
- El lenguaje de transferencia entre registros permite describir programas a nivel de microoperaciones al igual que los lenguajes máquina binario y lenguaje ensamblador. La ventaja del lenguaje RTL es que su sintaxis es independiente de la arquitectura de la computadora, es decir, es un lenguaje Universal. Por lo tanto si describimos un programa en el lenguaje RTL su transcripción a un lenguaje ensamblador de arquitectura específica como la arquitectura amd64,ARM, RISC-V,etc resulta mucho más sencilla.

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 4. Instruction Set

| Instruction name | Op | Description | Register Transfer Language (RTL) |
|------------------|-----------|---|----------------------------------|
| S(x)→A C+ | LOAD M(X) | copy the number in Selectron location x into AC | AC ← M[x] |

| Instruction name | Instruction name | Op | Description | Register Transfer Language (RTL) |
|------------------|---------------------|----|--|----------------------------------|
| S(x)→A c- | LOAD -M(X) | 2 | same as #1 but copy the negative of the number | $AC \leftarrow \sim M[x] + 1$ |
| S(x)→A cM | LOAD M(X) | 3 | same as #1 but copy the absolute value | $AC \leftarrow M[x] $ |
| S(x)→A c-M | LOAD - M(X) | 4 | same as #1 but subtract the absolute value | $AC \leftarrow AC - M[x] $ |
| S(x)→A h+ | ADD M(X) | 5 | add the number in Selectron location x into AC | |
| S(x)→A h- | SUB M(X) | 6 | subtract the number in Selectron location x from AC | |
| S(X)→ AhM | ADD M(X) | 7 | same as #5, but add the absolute value | |
| S(X)→ Ah-M | SUB M(X) | 8 | same as #7, but subtract the absolute value | |
| S(x)→ R | LOAD MQ,M(X) | 9 | copy the number in Selectron location x into AR | |
| R→A | LOAD MQ | A | copy the number in AR to AC | |
| S(x)*R →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) →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→S(x) | JUMP M(X,0:19) | D | Continue execution at the left-hand instruction of the pair at Selectron location x | |
| Cu`→S (x) | JUMP M(X,20:39) | E | Continue execution at the right-hand instruction of the pair at Selectron location x | |
| Cc→S(x) | JUMP+ M(X,0:19) | F | If the number in AC is ≥ 0 , continue as in #D. Otherwise, continue normally. | |
| Cc`→S(x) | JUMP+ M(X,20:39) | 10 | If the number in AC is ≥ 0 , continue as in #E. Otherwise, continue normally. | |
| 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) | |

| Instruction name | Instruction name | Op | Description | Register Transfer Language (RTL) |
|------------------|------------------|----|--|----------------------------------|
| halt | | 0 | Halt the program (see paragraph 6.8.5 of the IAS report) | |

- Instruction Set (William Stalling)

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

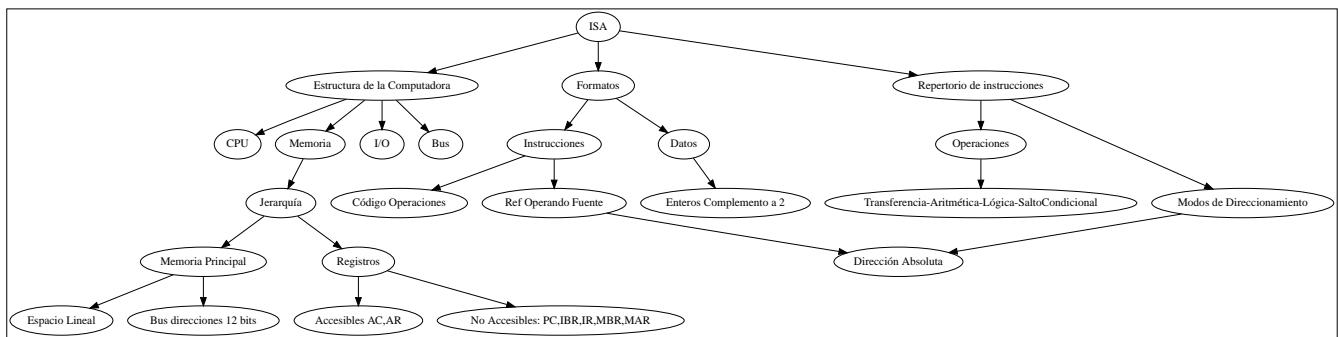


Figure 25. Instruction Set Architecture (ISA)

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

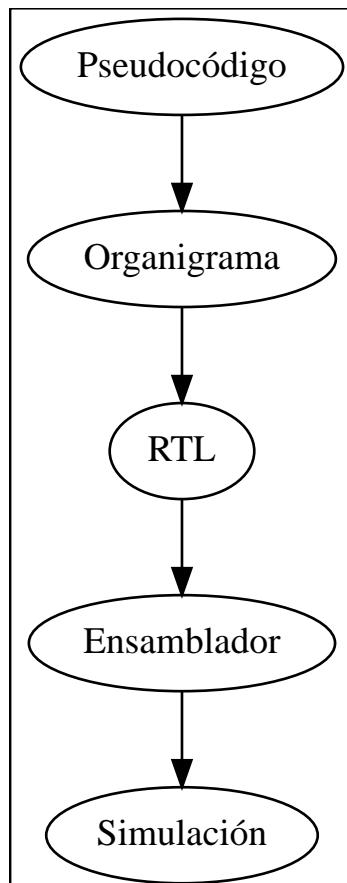


Figure 26. Fases de la Programación



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. Codificación Binaria-Hexadecimal

- Ejecutar el programa en lenguaje python:

```
# Tabla decimal-binario-hexadecimal

for i in range(256):
    print(str(i)+" 0b"+'{:b}'.format(i).zfill(8))
    print(str(i)+" 0b"+'{:9_b}'.format(i)+" 0x"+'{:_x}'.format(i))
    print()
```

- los dígitos binarios son 2: el 0 y el 1
- los dígitos hexadecimales son 16: 0-1-2-3-4-5-6-7-8-9-A-B-C-D-E-F. Que se corresponden con los valores 0-1-2... hasta el valor 15.
- El valor 9 se representan en binario como 1001 donde el peso de cada dígito viene determinado por su posición: $2^3, 2^2, 2^1, 2^0$
 - El valor del número 1001 es la suma ponderada del los digitos del número con su peso: $1*2^3+0*2^2+0*2^1+1*2^0$
- En hexadecimal
 - el número 0xA tiene el valor 10, el 0xB 11, el 0xC 12
 - el número 0x10 tiene el valor $1*16^1+0*16^0 = 16$
 - el número 0xFD tiene el valor $15*16^1+13*16^0 = 240+13 = 253$
- Relación hexadecimal-binario
 - el número 0xF6 se convierte en binario mediante la conversión de cada dígito hexadecimal en un grupo de 4 dígitos binarios. F (valor 15) en binario 1111 y 6 (valor 6) en binario 0110. Por lo que al número hexadecimal 0xF6 le corresponde el binario 0b11110110
- Ejemplos de números de 1 byte: decimal-binario-hexadecimal

```
16 0b0001_0000 0x10 17 0b0001_0001 0x11 18 0b0001_0010 0x12 19 0b0001_0011 0x13 20
0b0001_0100 0x14 21 0b0001_0101 0x15
22 0b0001_0110 0x16 23 0b0001_0111 0x17 24 0b0001_1000 0x18 25 0b0001_1001 0x19 26
0b0001_1010 0x1a 27 0b0001_1011 0x1b
28 0b0001_1100 0x1c 29 0b0001_1101 0x1d 30 0b0001_1110 0x1e 31 0b0001_1111 0x1f

192 0b1100_0000 0xc0 193 0b1100_0001 0xc1 194 0b1100_0010 0xc2 195 0b1100_0011 0xc3
196 0b1100_0100 0xc4 197 0b1100_0101 0xc5
198 0b1100_0110 0xc6 199 0b1100_0111 0xc7 200 0b1100_1000 0xc8 201 0b1100_1001 0xc9
202 0b1100_1010 0xca 203 0b1100_1011 0xcb
204 0b1100_1100 0xcc 205 0b1100_1101 0xcd 206 0b1100_1110 0xce 207 0b1100_1111 0xcf
```

2.5.3. 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

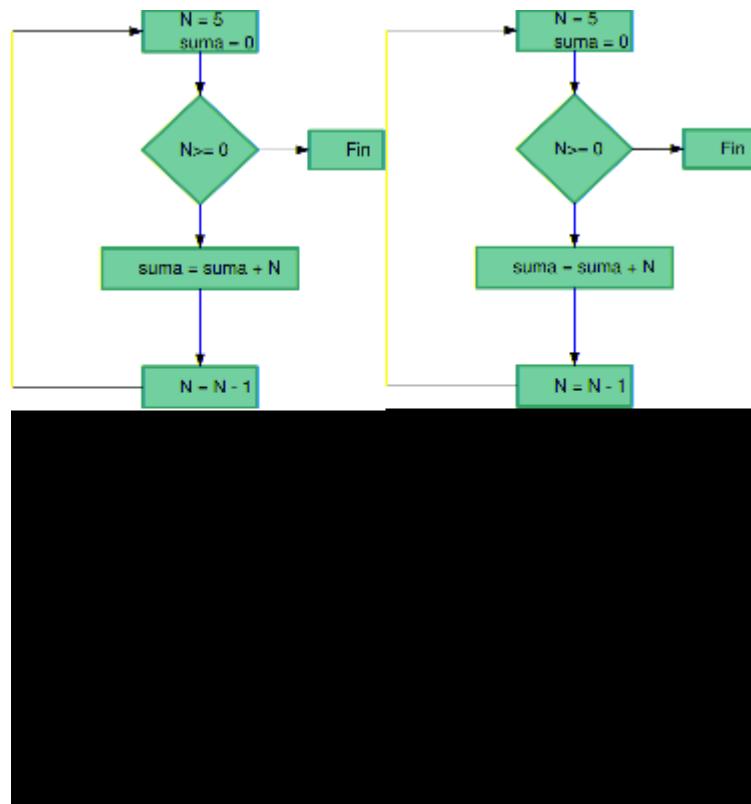


Figure 27. Organigrama: Diagrama de Flujo

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 generación de sumandos
bucle: AC <- M[n] ; cargar sumando
    AC>=0 : PC <- sumar ; si el sumando < 0 fin del bucle y si es >0 salto a
sumar
    ; 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]

```

```

JMP+    sumar      ; Si AC >= 0, salto a sumar
; fin del bucle
HALT      ; stop

; realizar la suma
sumar: ADD M(suma)      ; AC <- AC+M(suma)
STOR  M(suma)      ; M[suma] <- AC
; actualizar sumando
LOAD  M(n)       ; AC <- M[n]
SUB   M(uno)      ; AC <- AC-M[uno]
STOR  M(n)       ; M[suma] <- AC
; comprobar condición bucle
JMP   bucle      ; salto incondicional

```

```

;;;;;;;;;;;;;;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

- módulo fuente *sum1toN_A.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
 - Sólo puede haber etiquetas que apunten a instrucciones ubicadas al **derecha** de una palabra.
 - 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 para conseguir que el número de instrucciones del programa sea **par**: utilizar la directiva **.empty**

- La sección de instrucciones debe de ir previamente a la sección de datos

```

; CABECERA
; Este código necesita ser DEPURADO
; 1ª version : sum1toN_A.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
- **FALTA:**
 - añadir las instrucciones para salvar los operandos en la dirección de memoria "n" en cada iteracción del bucle
 - añadir las instrucciones para realizar la suma parcial en cada iteracción del bucle
 - Ver versión del programa en lenguaje RTL ():

```

SECCION INSTRUCCIONES
bucle: AC <- M[n] ; cargar sumando
        AC>=0 : PC <- sumar ; si el sumando < 0 fin del bucle y si no salto a

```

```

    sumar
        stop
        ; realizar la suma
    sumar: AC <- AC + M[suma]
        M[suma] <- AC
        ; actualizar sumando
        AC <- M[n]
        AC <- AC - M[uno]
        M[n] <- AC
        PC <- bucle ; salto incondicional

    SECCION DATOS
    n: 5      ; variable sumando e inicializacion
    uno: 1     ; variable sumando e inicializacion
    suma: 0    ; variable suma parcial y final

```

- Versión rectificada

```

; CABECERA
; Código rectificado
; Version : sum1toN_A.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 maquina 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]
    Cc->S(x)  sumar ; si el sumando < 0 fin del bucle y si no salto a sumar
    ; fin del bucle
    halt          ; stop
    .empty
    ; realizar la suma
sumar: S(x)->Ah+ suma      ; AC <- AC + M[suma]
    At->S(x)  suma      ; suma <- AC
    ; actualizar sumando
    S(x)->Ac+ n      ; AC <- M[n]
    S(x)->Ah- uno      ; AC <- AC - M[uno]
    At->S(x)  n       ; n <- AC
    Cu->S(x)  bucle     ; salto incondicional a la instrucción izda de bucle

;;;;;;;;;;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

```

- Versión B

```

; Version : sum1toN_B.ias
; Calcula la suma de una secuencia de numeros enteros: suma = 1+2+..+n
loop:   S(x)->Ac+ n      ;load n into AC
        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
        Cc'->S(x) loop   ;if AC >= 0, jump to pos right instruction
        halt            ;otherwise done

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

```

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

2.5.4. Ejemplos de Programas en Lenguaje IASSim

- Más ejemplos en el [Apéndice](#).

2.6. Operación de la Máquina IAS: Ruta de Datos

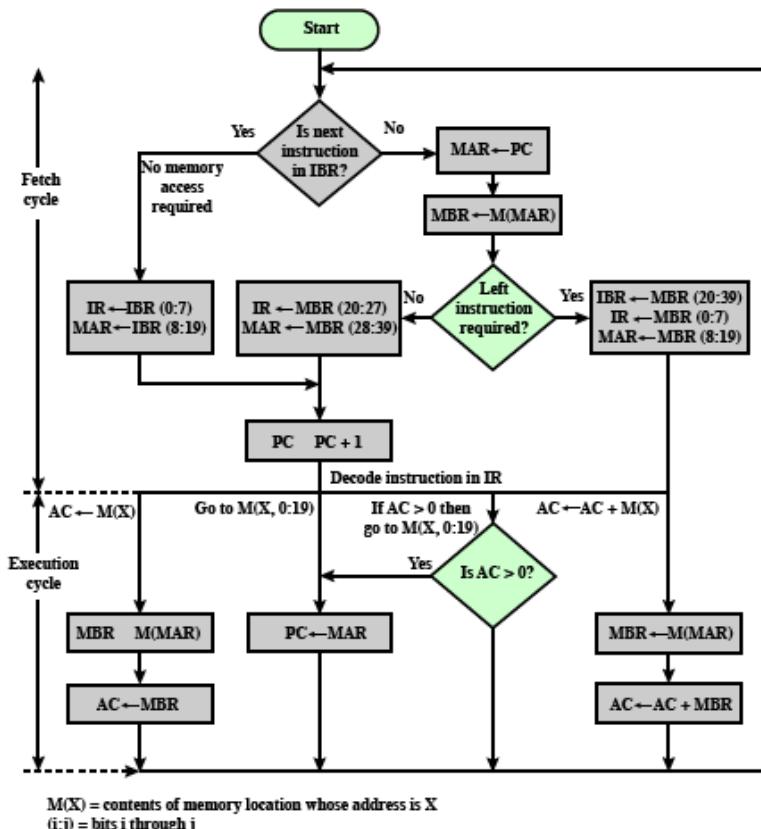


Figure 2.4 Partial Flowchart of IAS Operation

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

- Binary 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
```

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 $10^{\text{posición}}$ → ..., Centenas, decenas, unidades
 - Valor representado = sumatorio de dígitos 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 $2^{\text{posición}}$ → ... $2^5, 2^4, 2^3, 2^2, 2^1, 2^0, \dots, 64, 32, 16, 8, 4, 2, 1$
 - 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 | $1*8$ | $0*4$ | $1*2$ | $0*1$ |

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

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

- La Rueda: representación y valor de los números sin signo con 3 bits.

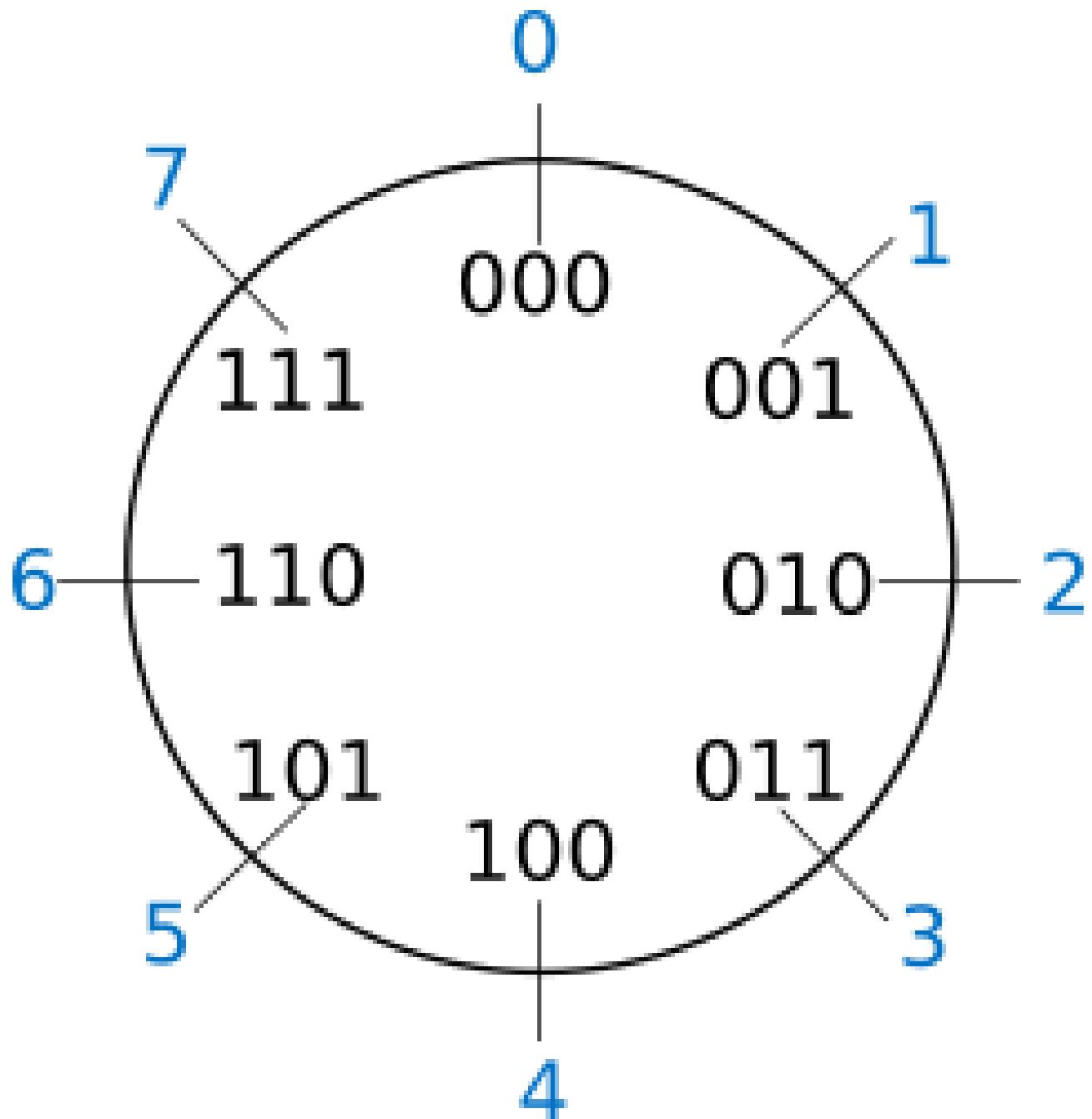


Figure 29. Representación Números sin Signo

3.5.3. Conversión Decimal-Binaria

- Divisiones sucesivas / 2 → Dividendo₁ = 2*Cociente₁ + Resto₁
- Cociente₁ = 2*Cociente₂ + Resto₂ → Dividendo₁ = 2 * (2*Cociente₂ + Resto₂) + Resto₁ = Resto₁*2⁰ + Resto₂*2¹ + Cociente₂*2²
- 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 5. 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 ^o Div | | |
|--------|--------------------|-------|-----|--------------------|-----|-------|--------------------|-------|-----|---------------------|-----|-------|
| | Coc | Resto | Coc | Resto | 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 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: representación y valor de los números con signo con 3 bits.

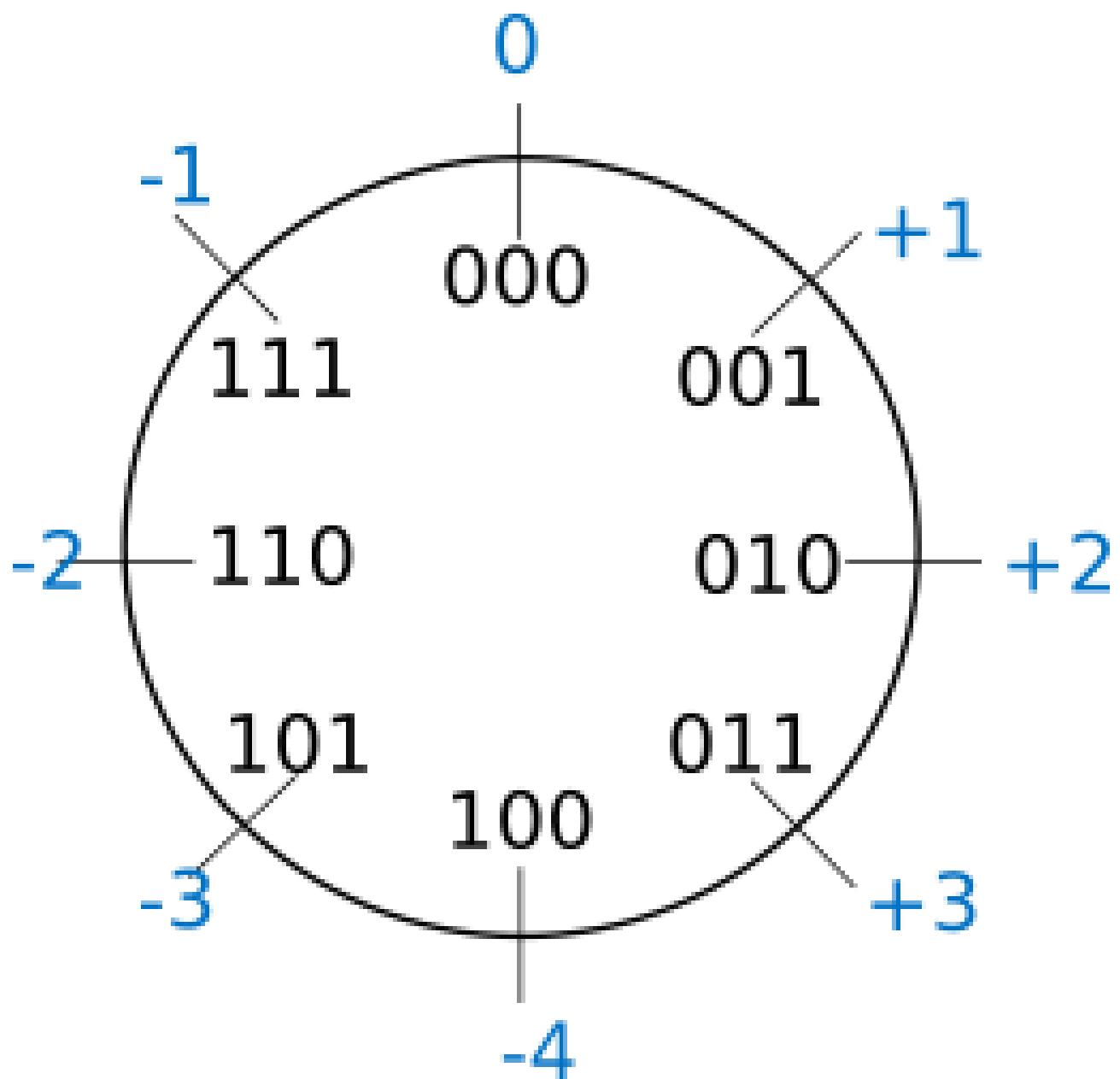


Figure 30. Representación Complemento a 2

- 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:
 - 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
 - C2(X)=1001100011101010101111+1=100110001110101010110000 → negativo por tener el bit más significativo (MSB) uno
 - 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
 - C2 (X) = 10011000111010101-10000
 - Método 3: Realizar la operación aritmética 0-X
 - X = 0110011100010101010000
 - $0-X=00000000000000000000000000 - 0110011100010101010000 = 1001100011101010110000$

- 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
 - $C2(n=10000000) = 01000000 = 2^7$, 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 $-23.4567 \cdot 10^{-34}$
 - La **mantisa o significando** es el número que multiplica a la potencia → -23.4567
 - Mantisa **normalizada** : La mantisa tiene como parte entera un número entero de un dígito distinto de cero. → $-2.34567 \cdot 10^{-33}$
 - 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
  
```

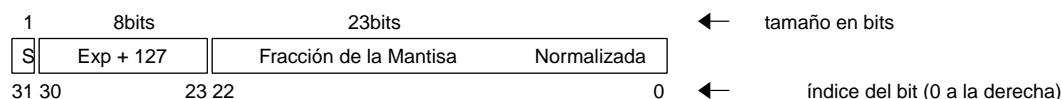
- fracción redondeada 0.1001
- fracción sin redondear 0.10010001011000010
- fracción redondeada 0.100100011
- Código Binario coma fija: 10011010010.10010001011000010
- Notación científica: $1.001101001010010001011000010 \cdot 2^{+10}$ → coma flotante → la parte entera siempre vale 1.

Precisión

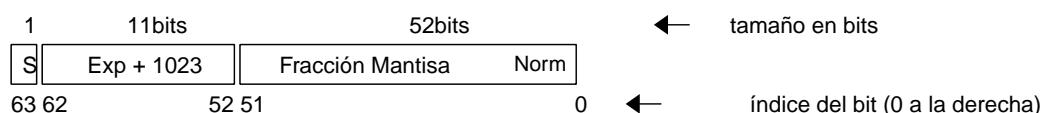
- Es el número de dígitos significantes
- Se dice que el número q es una aproximación del número p != 0 con una precisión de, al menos, **m** cifras significativas en la base b, siempre que el error relativo $|p-q|/p \leq 0.5 \cdot b^{-m+1}$
 - Cuando m es el mayor entero para el que se cumple la desigualdad anterior, se dice que q approxima a p con m cifras significativas.
- Ejemplo
 - a. p = 1E0 y q = .9999E0 → Error relativo=0.1E-3<0.5E(-4+1) → precisión de 4 cifras significativas
 - b. Una calculadora, A, trabaja en base 2 con mantisa de 22 bits y otra, B, trabaja en base 16 con 6 dígitos de precision (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 longitudes 1/8/23 bits



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



- [conversion manual](#)
- [wiki](#)
- 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](#): tipos char,short,int,float,double
 - [conversor ieee754](#)
 - [IEEE 754 single precision](#): 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 6. Single precision

| Números | Exp | Fracción |
|---------|------|----------|
| Ceros | 0x00 | 0 |

| Números | Exp | Fracción |
|-------------------------|-----------|---------------|
| 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
- Infinito

Referencia

- numerical analysis: programas ejemplo sencillos
- IEEE
- William Kahan
- Yale: C programming. float.c.
- Bruce Dawson blog
 - <https://randomascii.wordpress.com/2012/01/11/tricks-with-the-floating-point-format/>
- wikipedia
 - <http://www.validlab.com/goldberg/paper.pdf>
 - The pitfalls of verifying floating-point computations
 - el mismo?
- cprogramming
- code tips
- c review: practicas

3.7. Character Type

3.7.1. ASCII

- Codificación ASCII
 - American Standard Code International Interchange: 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 | |

| Carácter | ASCII hex | Control (Secuencia de Escape) |
|----------|-----------|-------------------------------------|
| 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
- `man iso_8859-1 | grep ñ`
- <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](#)
- 0
- ~

- ñ
- 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](#)
- 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 UTF-8 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 (ascii extendido) 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**:
 - 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](#)
- la ñ tiene **unicode point U+00F1 ó hex_code_utf8 0xC3B1**
 - en la wikipedia utf-8 explica cómo pasar 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
^J 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 Significand Byte.
177 0261 0xb1  -> LSB: Less Significand Byte
hex_code_utf8    -> 0xC3B1
  
```

- 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"
- URL → en la barra de direcciones de un navegador poner camión → enter
 - copiar la URL y copiarla en una nueva barra → <https://www.google.com/search?channel=fs&client=ubuntu&q=cami%C3%B3n>
 - %C3%B3 es el UTF-8 hex_code de ó
- Sistema operativo : variables del entorno
 - [env | grep LC_](#)
- [Unicode chart](#)
 - Colocando el puntero sobre la categoría se visualiza el rango hexadecimal del charset
 - Symbols Punctuation:
 - Punctuation: ASCII Punctuation: [U0000.pdf](#)
 - Find chart by hex code: 278a
 - Pictographs: Dingbats: x278a → ☰ → [U2700.pdf](#)

- Mathematical symbols: Mathematical Operators:
 - [wikipedia](#)
 - [U2200](#): Mathematical Operators Range: 2200–22FF
 - hexadecimal x2228 → √ ó en decimal 8744 √
 - x22bc → ∞
 - x22bd → ∇
 - x22a6 → ⊢
- otros
 - x1f60b → 🕵
 - x00f1 → ñ
 - 241 → ñ
 - x2190 → ←
 - x2192 → →
- https://wiki.mozilla.org/Help:Special_characters#Unicode
- info detallada sobre un carácter unicode: Pej U+0305
- Microsoft Office
- Overline o suprarayado
 - 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
 - [locale -a](#) → C.UTF8
 - ñ → [env printf \u00f1 \n](#) : incluir las simples comillas
 - [printf invocation](#)

Chapter 4. Operaciones Aritmeticas y Logicas

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.
 - Ejemplo: Representación de los números binarios en módulo 8. Suma binaria en módulo 8. Representación gráfica para números binarios de 3 bits → módulo= $2^3 = 8$

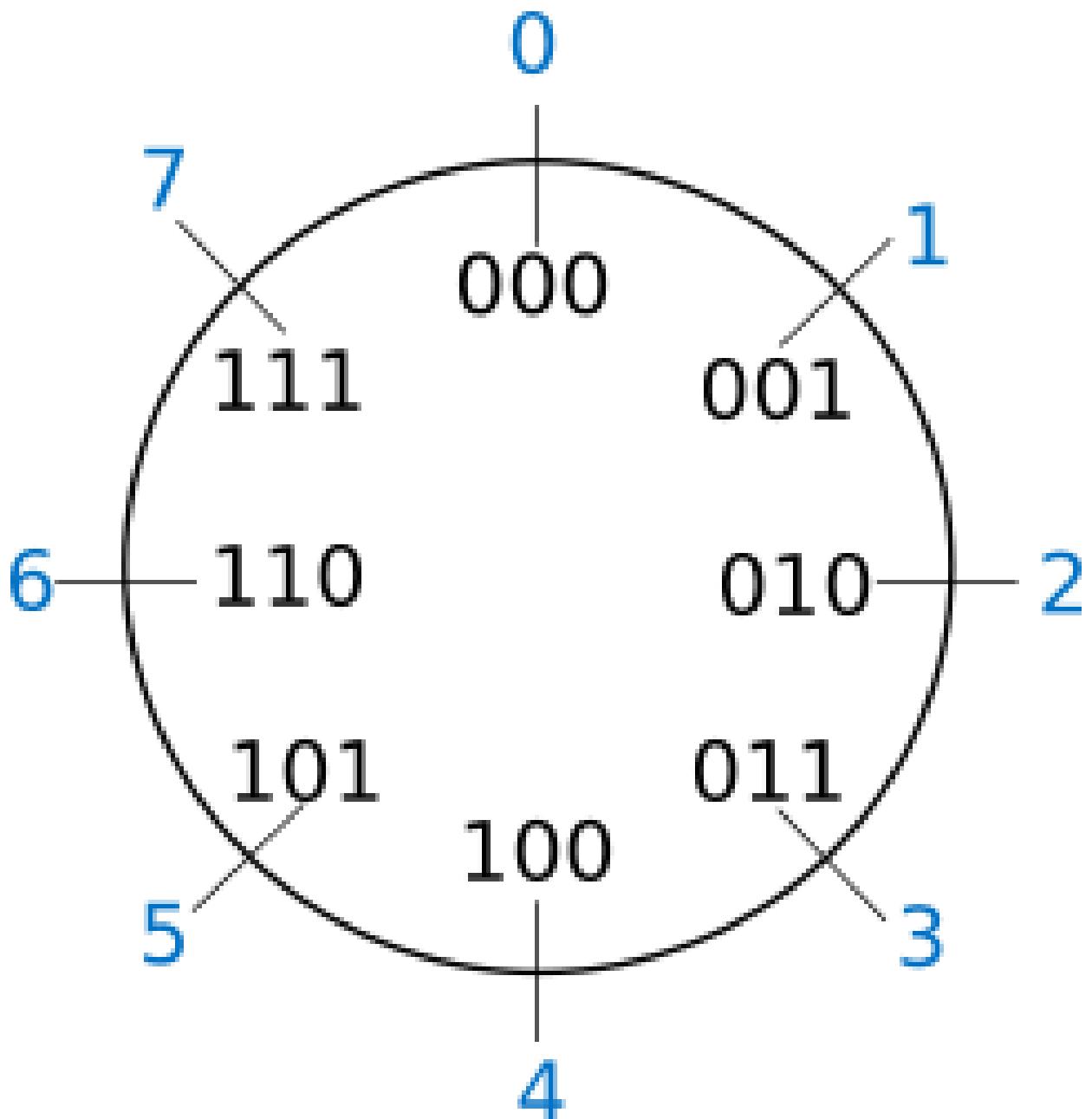


Figure 31. Representación Unsigned Number

- Suma: Gráficamente se puede ver que $7+1=0$. La ALU al realizar la operación suma $111+001$ da como resultado 000. $7+1$ da como resultado el valor 8 que en módulo 8 es 0.
- Suma: ¿Cuánto sería $7+7$? $\rightarrow 14$ en módulo 8 es 6. Si el resultado es igual o superior al módulo hay que restarle el módulo tantas veces como sea necesario.
- Suma: Cuánto vale 33 en módulo 8 $\rightarrow 33-8*4=1$. El 1 está en el rango (0 , 7).
- Resta: Gráficamente se puede ver que $0-1=7$. La ALU al realizar la operación resta $000-001$ da como resultado 111
 - El **acarreo** (llevada) se produce al llegar o pasar el valor 2.
 - $1+1=\text{uno más uno} = \text{dos} >= 2 \rightarrow$ 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.
 - $1+1+1=\text{uno más uno más uno} = \text{tres} >= 2 \rightarrow$ 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 0000000 y 1111111, 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 \rightarrow 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 $0+módulo-1=0+2-1=1$. 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 $0+módulo-1-1=0+2-1-1=0$ y llevada 1 que sumaremos a la posición siguiente del sustraendo.
- posición "p": $1-1-1 \rightarrow$ en el sustraendo $1+módulo-1-1=1+2-1-1=1$ y llevada 1 que sumaremos a la posición siguiente del sustraendo.
- $10110110 - 10011011 = 00011011$

Sumar crédito al minuendo

2 2 2 2

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

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

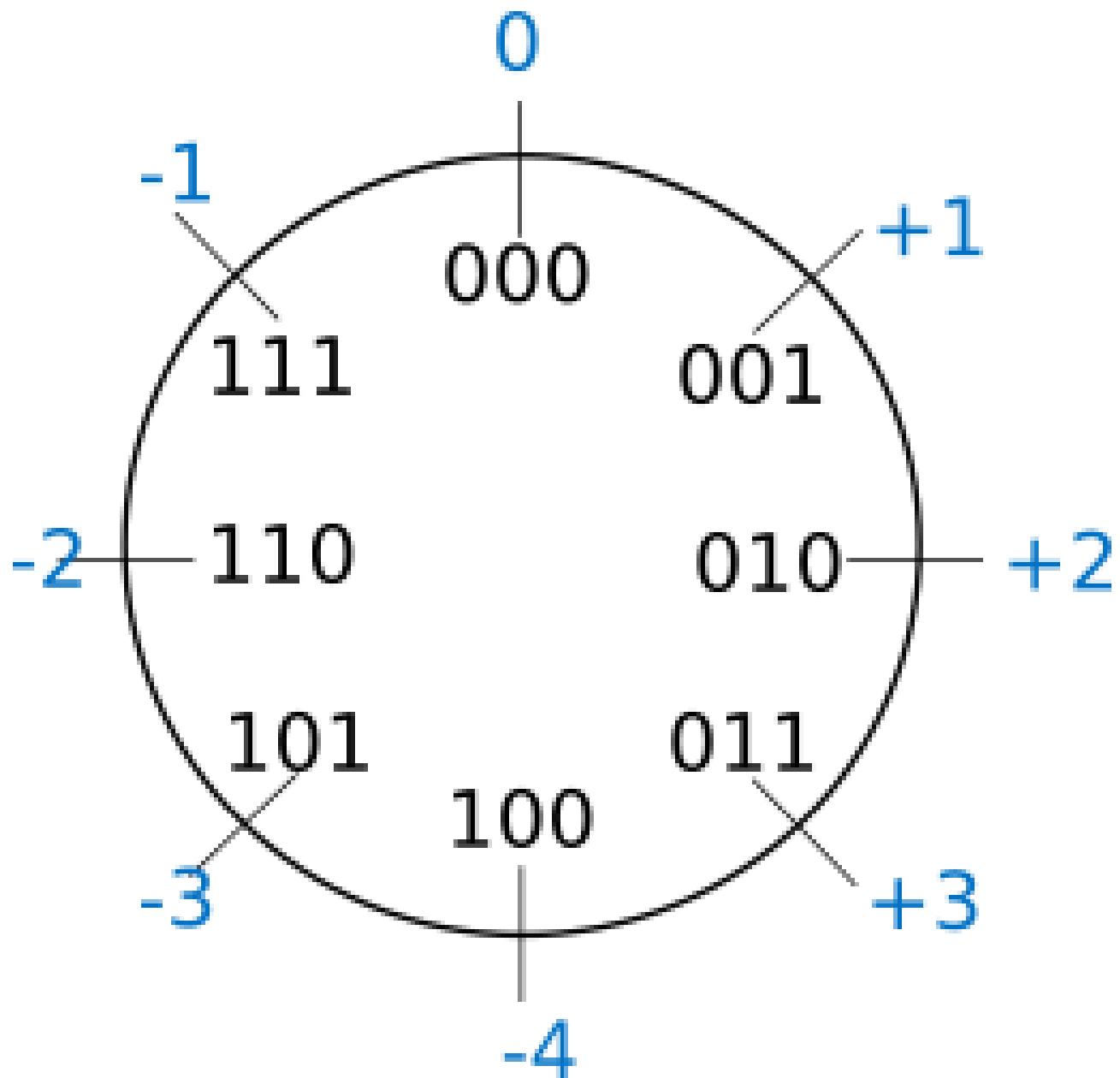


Figure 32. Representación Complemento a 2

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

$$\begin{array}{r} 10011011 \\ +00011011 \\ \hline \end{array} \quad \begin{matrix} <\text{--AL} \\ <\text{--BL} \end{matrix}$$

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 cambiarlo 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 10110110 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 <-- 0000000000011011 (+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 = dieciséis $\geq 16 \rightarrow$ al resultado dieciséis 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 dieciséis le resto 16 ($16-16=0$) y me llevo una.

4.4.5. Resta en Módulo 16 (Hexadecimal)

- Todo lo visto anteriormente para números binarios se puede realizar en cualquier otra base, por ejemplo en números codificados en hexadecimal.
- Resta en módulo 16:
 - el acarreo se produce cuando una posición p del minuendo es inferior a la misma posición p del sustraendo, en cuyo caso, es necesario sumar el módulo 16 al minuendo y la llevada a la posición 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

```

- $8-C \rightarrow 8 + \text{módulo } 16 - 12 = 8 + 16 - 12 = 12 = 0xC$ y llevada 1 a la posición siguiente
- $0-B - \text{Llevada} \rightarrow 0 + \text{módulo } 16 - 11 - 1 = 0 + 16 - 11 - 1 = 4 = 0x4$ y llevada 1 a la posición siguiente
- $3-A - \text{Llevada} \rightarrow 3 + \text{módulo } 16 - 10 - 1 = 3 + 16 - 10 = 8 = 0x8$ y llevada 1 a la posición 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 números 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
- algebra symbols
 - Bitwise operator: and &, or |, xor ^, not ~
 - Shift operator: left <<, right signed >>, right unsigned >>>

| Operador | Algebra | C |
|-------------------------|-----------------|-----------|
| NOT | \neg \sim | \sim |
| OR | \vee | $ $ |
| AND | \wedge | $\&$ |
| XOR | \oplus \vee | \wedge |
| NOR | $\bar{\vee}$ | |
| NAND | $\bar{\wedge}$ | |
| Left SHIFT | | $x << m$ |
| Right SHIFT signed | | $x >> m$ |
| Right SHIFT unsigned | | $x >>> m$ |

Tablas de la Verdad

| x | y | $z=x \vee y$ | $z=x \wedge y$ | $z=x \oplus y$ |
|---|---|--------------|----------------|----------------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 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
 - not, and, or, xor

Complejos

- half adder, full adder
- multiplicador
 - 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.

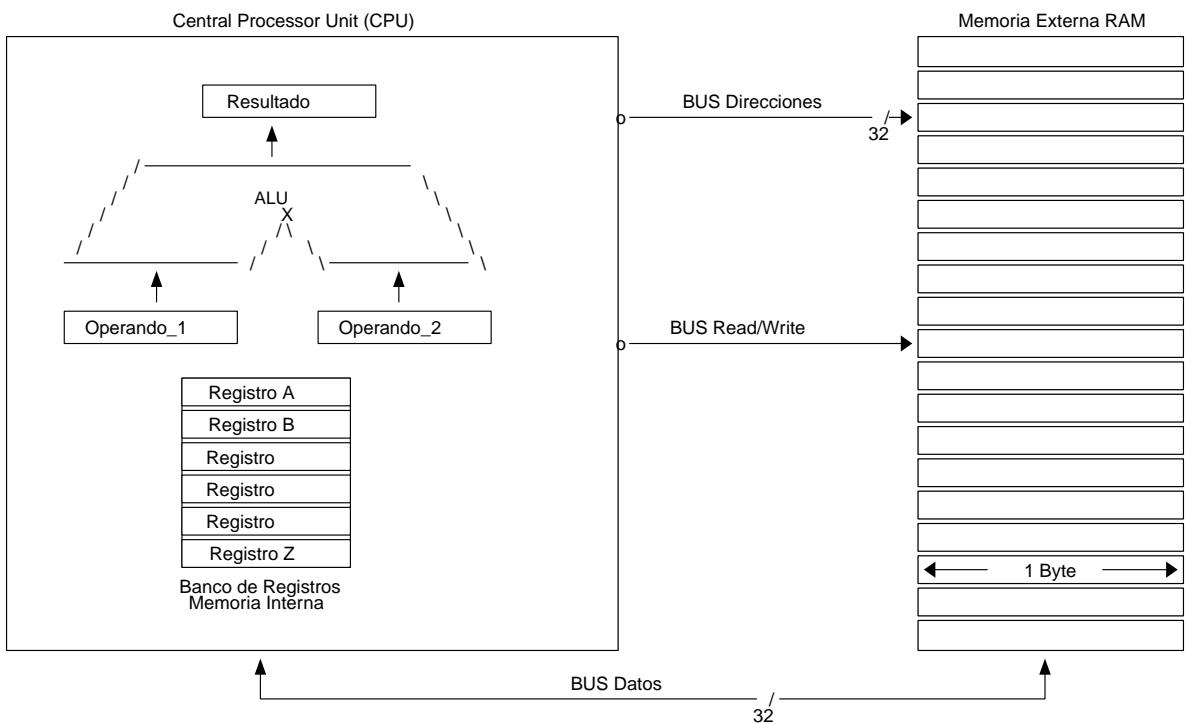


Figure 33. Arquitectura Intel x86 de 32 bits

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 7. 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:



Hay que diferenciar los casos suma CON signo y suma SIN signo. En el primer caso detectamos el error matemático únicamente con el flag OF y en el segundo caso detectamos el error matemático únicamente con el flag CF.

- Números CON signo (complemento a 2):
 - para saber si hay overflow siempre se suma...una resta se puede convertir en suma
 - El carrier flag CF no tiene sentido. Únicamente interpreto OF para saber si hay error en la operación aritmética.

$$\begin{array}{r} 11111111 \\ + 00000001 \\ \hline \end{array}$$

100000000 -> Esta suma NO es correcta, ya que para representar el resultado con 9 bits los operandos tienen que ser de 9 bits y por lo tanto hay que extender el bit de signo de los operandos de 8 bits. La suma con 9 bits sería:

operando y resultado con 9 bits:

$$\begin{array}{r} 11111111 \\ + 00000001 \\ \hline \end{array}$$

000000000 -> NO hay Overflow ya que los operandos de la suma son de distinto signo

operando y resultado con 8 bits:

$$\begin{array}{r} 1111111 \\ + 00000001 \\ \hline \end{array}$$

00000000 -> NO hay Overflow ya que los operandos de la suma son de distinto signo

Nunca va haber overflow si sumamos datos de signo contrario

Resta A-B donde

$$A=11110000$$

$$B=00010100$$

$A-B=A+(-B)$ -> Convierto la resta en suma

$$A : 11110000$$

$$-B : +11101100$$

$A-B: 11011100$ -> Hay acarreo pero NO overflow. La suma 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.

- números SIN signo
 - El overflow flag OF no tiene sentido. Unicamente interpreto CF para saber si hay error en la operación aritmética.

$$\begin{array}{r} 11111111 \\ + 00000001 \\ \hline \end{array}$$

00000000 -> Hay acarreo en el bit más significativo luego CF=1.
Conceptualmente hay overflow por lo que el resultado que obtiene la ALU aunque electrónicamente es correcto, no lo es matemáticamente ($511+1=0$). El efecto overflow lo detecto con CF=1.

Extiendo los operandos de la operación anterior con 1 bit.

$$\begin{array}{r} 01111111 \\ + 00000001 \\ \hline \end{array}$$

100000000 -> No hay acarreo -> CF=0. Esta suma es correcta matemáticamente ya que $511+1=512$ y por lo tanto no hay overflow -> CF=0

- 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 10: Conjuntos de Instrucciones : Características y Funciones (Datos, Operando y Operaciones)
 - Capítulo 11: 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. Lenguajes de programación de alto nivel vs Lenguajes de Programación de bajo nivel

5.3.1. Lenguajes de alto nivel

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 BINARIO, 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.

- Ejemplos de lenguajes de Programación de alto y bajo nivel: [Apéndice](#).

5.3.2. El lenguaje máquina y el lenguaje ensamblador

- En este tema se trata de la representación e interpretación de las instrucciones en lenguaje máquina y lenguaje ensamblador.
- 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, como es el texto, tiene como objetivo facilitar la tarea del programador en la interpretación de las instrucciones y en el desarrollo de programas en lenguaje ensamblador.
- El estudio de los formatos de las instrucciones máquina de un procesador específico se enmarca dentro del concepto ISA de la Arquitectura del Procesador [Ver apéndice](#)

5.4. Elementos de una Instrucción Máquina

- Una instrucción máquina se estructura en diferentes campos: campo de operaciones, campo de operando, etc ... El número de campos dependerá del procesador que se esté diseñando.



- En el caso de la máquina IAS el formato de instrucción tiene únicamente dos campos: el código de operación y el campo del operando.
- Código de Operaciones:
 - La instrucción debe de especificar que operación debe de realizar la CPU. Operaciones cómo 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, la operación suma ADD requiere de dos operandos, etc
- 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.

- Direcciones del operando implícitas: Direcciones que no aparecen explícitamente en la instrucción.
Ejemplos:
 - La Próxima instrucción es la dirección almacenada en otro registro: el Contador de Programa
 - El Resultado de la operación se guarda en otro registro: el Acumulador
 - etc

5.4.1. 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 operación **(A-B)/(DxE+C)** según 4 arquitecturas ISA diferentes: arquitectura con 3 operandos referenciados, con 2 operandos referenciados, con 1 operando referenciado y ningún operando referenciado

| Instruction | Comment |
|--------------------|---------------------------|
| 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

| Instruction | Comment |
|--------------------|---------------------------|
| 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 | $Y \leftarrow Y \div T$ |

Instruction Comment

| | |
|--------|-----------------------------|
| 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 | $Y \leftarrow AC$ |

- 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;

5.5. Instrucciones en lenguaje máquina de la arquitectura x86

- Ver el apéndice [Apéndice](#) a modo de comprender un ejemplo. No es posible programar manualmente en lenguaje máquina en una computadora actual.

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

5.6.1. Introducción

- Los dos campos más importantes y casi únicos del formato de instrucción son: El código de operación y los modos de direccionamiento de los campos de operandos. El número de operandos puede ser 0,1,2,3,etc

5.6.2. Códigos de Operación

- La codificación del conjunto de operaciones depende de cada arquitectura ISA.
- Categorías según el tipo de operaciones:
 - 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
 - El repertorio puede ser: reducido/extenso, complejo/sencillo.
- En el lenguaje ensamblador a la palabra que indica el tipo de operación, por ejemplo ADD para una suma, se le denomina **mnemónico** y suele estar en lengua inglesa, permitiendo intuir fácilmente de qué operación se trata.
- La mejor forma de practicar con los mnemónicos del lenguaje ensamblador es programando, lo cual se verá en el siguiente tema [lenguaje ensamblador x86](#)

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 |
| | Rotate | Left (right) shift operand, with wraparound 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 |

5.6.3. Operандos: Modos de Direcciónamiento

Localización

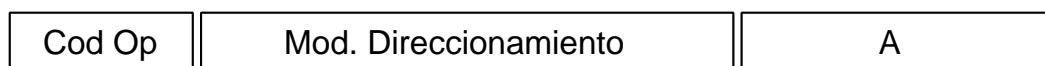
- Posibles ubicaciones de los operandos.

- En la propia instrucción
- Memoria interna: registros CPU
- Memoria Principal: memoria DRAM
- Memoria i/o: registros en controladores de entrada/salidas denominados puertos.
- La instrucción tiene que hacer referencia de alguna forma (modo de direccionamiento) a la ubicación del operando.

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 refiere 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

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.

5.7. Lenguaje Intel versus Lenguaje AT&T

5.7.1. Lenguajes ensamblador de la arquitectura i386/amd64

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

5.7.2. 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 8. Sintaxis Intel

| label: | op_mnemonic | operand_destination | , | operand_source | #comment |
|--------|-------------|---------------------|---|----------------|----------|
|--------|-------------|---------------------|---|----------------|----------|

- Ejemplo:

- bucle: sub rsp,16 ;RSP ← RSP-16. Comienzo del bucle con la operación subtraction
- je bucle ;je: jump equal: salto si la última operación dió resultado cero
- suma: add eax,esi ;EAX ← EAX+M[ESI] . Sumar
- mov ax,[resultado] ;AX ← M[resultado]. Copiar el resultado



La sintaxis del lenguaje ensamblador depende del traductor del proceso de ensamblaje (assembler) utilizado, en este caso, se utiliza el assembler NASM. Ver un ejemplo en el [Apéndice](#) de un programa en lenguaje ensamblador intel y assembler "NetWide Asm" (nasm)

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 9. Sintaxis AT&T

| label: | op_mnemonic | operand_source | , | operand_destination | ;comment |
|--------|-------------|----------------|---|---------------------|----------|
|--------|-------------|----------------|---|---------------------|----------|

- Ejemplo:

- bucle: subq \$16,%rsp ;RSP ← RSP-16. Comienzo del bucle con la operación subtraction
- je bucle ;je: jump equal: salto si la última operación dió resultado cero
- suma: addl %esi,%eax ;EAX ← EAX+M[ESI] . Sumar
- movw %ax,resultado ;AX ← M[resultado]. Copiar el resultado

- ETIQUETA

- Se especifica en la primera columna. Tiene el sufijo :

- CODIGO DE OPERACION: Se utilizan símbolos *mнемónicos* 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,..
 - 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.

5.8. Operandos: Modos de Direccionamiento

5.8.1. 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".

5.8.2. 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. Únicamente 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. |

| | |
|--|---|
| | <p>sintaxis: Nombre del registro con el prefijo %.</p> <p>ejemplo: <code>movl %eax, %ebx</code>. El operando fuente es el REGISTRO EAX y el destino es el REGISTRO EBX</p> |
| 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</p> <p>ejemplo: <code>je somePlace .</code> Salto a la dirección marcada por la etiqueta somePlace 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.</p> <p>$EA = Offset + R_Base + R_índice * Scale$</p> |
| | <p>sintaxis: lista de valores separados por coma y entre paréntesis (base_register, index_register, scale) y precedido por un offset.</p> <p>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)</p> <p>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.</p> <p>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 |

- $M[\text{loop_exit}]$: directo ya que `loop_exit` es una dirección de memoria externa y M indica la memoria externa.
- $M[\text{RBX}]$: 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.

5.9. Programas en lenguaje ASM y lenguaje Binario

- Ejemplos en el [Apéndice](#)

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_AS(gas).
- Capacidad para desarrollar pequeños programas en lenguaje ensamblador para la arquitectura x86 (32 bits) tanto en papel impreso como en un entorno de desarrollo computacional bien en el ordenador personal y también en las estaciones de trabajo del Laboratorio de Informática.

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
 - [Sintaxis y Direccionamiento ASM x86](#): Sintaxis de las instrucciones en lenguaje ensamblador GnuAS y modos de direccionamiento para la arquitectura x86.

6.2.3. Referencias

- Manual Assembler as: [Apéndice: Cadena de herramientas Toolchain "as i386"](#)
- **Leerse** los ocho primeros apartados del manual "Assembler as": [Apéndice: Características "as i386"](#)
- Manual Assembler as: [Apéndice: Directivas "as i386"](#)
- Referencias a manuales en el [Apéndice](#)
- Bibliografía: [Programación Ensamblador](#).
- Apuntes completos [WikiBook](#) de programación en lenguaje AT&T con diversidad de aspectos.

6.3. Estructura de la Computadora con Arquitectura Intel x86-64

6.3.1. Manuales Intel ISA x86

- <https://www.intel.es/content/www/es/es/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html>
- 8 Volúmenes: 1,2A,2B,2C,3A,3B,3C,3D
 - Volume 1: Basic Architecture
 - Volumes 2A, 2B and 2C: Instruction Set Reference
 - Volumes 3A, 3B, 3C and 3D: System Programming Guide

6.3.2. Intro

- Para poder programar en lenguaje ensamblador GAS x86 eficazmente es necesario tener las nociones básicas de la ISA x86.
- Se va analizar la arquitectura amd64 de 64 bits que abarca también a la arquitectura i386 de 32 bits.
- Las empresas Intel y AMD comparten dichas arquitecturas cuyo mercado abarca tanto ordenadores de sobremesa como servidores.

6.3.3. CPU-Memoria

- Ver la estructura general de una computadora en el [Apéndice](#)

6.3.4. Registros internos a la CPU x86

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 | cx | ch | cl |
| rdx | edx | dx | dh | dl |
| rsi | esi | si | | sil |
| rdi | edi | di | | dil |
| rbp | ebp | bp | | bpl |

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

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](#)
- Únicamente nos fijamos en los flags OSZAPC.

Table 11. 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:

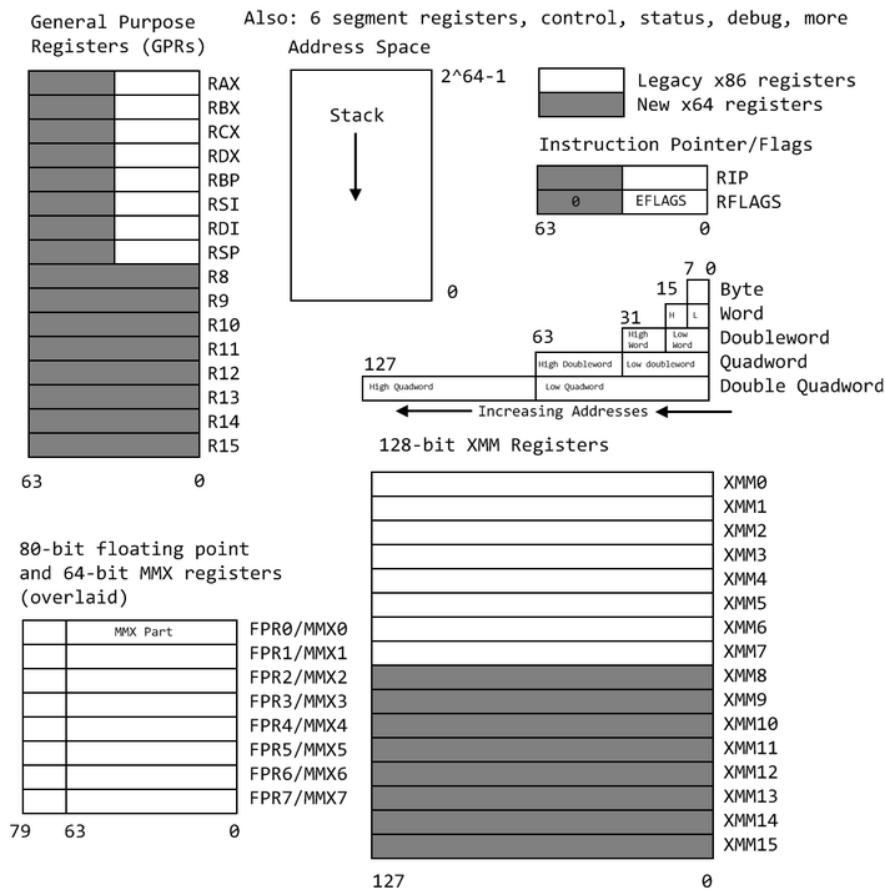
- se activa si teniendo en cuenta el bit de mayor peso MSB (aunque esté fuera el word size) indica error en la operación aritmética con números enteros con signo. Si no se tiene en cuenta el MSB fuera del word size, la operación es correcta.
- Parity Even flag:
 - indica si el número de bits del byte 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

Casos de Overflow



Ejemplos de errores matemáticos por overflow: activación de los flags CF y OF

Otros Registros internos a la CPU x86



- Segment Registers: CS,DS,ES,FS,GS,SS
 - https://en.wikipedia.org/wiki/X86_memory_segmentation
 - Normalmente se utilizan de forma implícita: las instrucciones están en el segmento de código en direcciones relativas al registro CS, los datos están en el segment data en direcciones relativas al registro DS, la pila está en el segment stack en direcciones relativas al registro SS.
 - utilización explícita:
 - `movl $42, %fs:(%eax) ; M[fs:eax]←42) ; eax contiene la dirección relativa a la`

dirección FS

- El Sistema Operativo utiliza los registros segmento en la gestión de memoria virtual mediante los mecanismos de paginación y segmentación: <https://nixhacker.com/segmentation-in-intel-64-bit/> : La gestión de la memoria es un tema de la segunda parte de la asignatura.
- 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. Representación de los datos en lenguaje ensamblador (ASM) para la arquitectura i386/amd64

6.4.1. 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

- Referencia [Apéndice](#)

Table 12. 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 |
| .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 carácter NULL de final de cadena |
| .byte 3,7,-10,0b1010,0xFF,0777 | tamaño 1Byte y formatos decimal,decimal,decimal,binario,hexadecimal,octal |

| Directivas | descripción |
|---------------------------------|---|
| .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 múltiplo de n bytes. |
| .end | fin del ensamblaje |

- Et: Etiqueta

6.4.2. Tamaño del operando x86

- Tamaño del operando: sufijos de los MNEMÓNICOS.

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

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

6.4.3. Alineamiento de Bytes: Big-LittleEndian

- Los bytes de un dato de varios bytes se pueden almacenar en memoria en sentido MSB-LSB ó MSB-MSB

- 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 | |
| 0x00003 | |
| 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 Little Endian.
- Formato BigEndian
 - 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.

6.4.4. Ejemplo

- Analizar el código del programa [sum1toN att x86-32](#)

6.5. Repertorio de Instrucciones en lenguaje ensamblador (ASM) para la arquitectura i386/amd64: Operaciones

6.5.1. Ejemplo

- En cada apartado que se estudia a continuación analizar el código del programa [sum1toN att x86-32](#) interpretando los códigos de operación.

6.5.2. Manual rápido

- [manual Intel quick](#): recomendado

6.5.3. Manuales y Tablas

- Referencias a manuales en el [Apéndice](#)

6.5.4. Tipo de descripción de Códigos de Operación en el Manual de Intel

Operación MOV

- [MOV](#)

MOV -- Move Data

| Opcode | Instruction | Clocks | Description |
|--------|----------------|--------------|-----------------------------------|
| 88 /r | MOV r/m8,r8 | 2/2 | Move byte register to r/m byte |
| 89 /r | MOV r/m16,r16 | 2/2 | Move word register to r/m word |
| 89 /r | MOV r/m32,r32 | 2/2 | Move dword register to r/m dword |
| 8A /r | MOV r8,r/m8 | 2/4 | Move r/m byte to byte register |
| 8B /r | MOV r16,r/m16 | 2/4 | Move r/m word to word register |
| 8B /r | MOV r32,r/m32 | 2/4 | Move r/m dword to dword register |
| 8C /r | MOV r/m16,Sreg | 2/2 | Move segment register to r/m word |
| 8D /r | MOV Sreg,r/m16 | 2/5,pm=18/19 | Move r/m word to segment register |

| | | | |
|---------|-----------------|-----|-----------------------------------|
| A0 | MOV AL,moffs8 | 4 | Move byte at (seg:offset) to AL |
| A1 | MOV AX,moffs16 | 4 | Move word at (seg:offset) to AX |
| A1 | MOV EAX,moffs32 | 4 | Move dword at (seg:offset) to EAX |
| A2 | MOV moffs8,AL | 2 | Move AL to (seg:offset) |
| A3 | MOV moffs16,AX | 2 | Move AX to (seg:offset) |
| A3 | MOV moffs32,EAX | 2 | Move EAX to (seg:offset) |
| B0 + rb | MOV reg8,imm8 | 2 | Move immediate byte to register |
| B8 + rw | MOV reg16,imm16 | 2 | Move immediate word to register |
| B8 + rd | MOV reg32,imm32 | 2 | Move immediate dword to register |
| C6 | MOV r/m8,imm8 | 2/2 | Move immediate byte to r/m byte |
| C7 | MOV r/m16,imm16 | 2/2 | Move immediate word to r/m word |
| C7 | MOV r/m32,imm32 | 2/2 | Move immediate dword to r/m dword |

- MOV NO afecta a ningún flag
- dword :double word: 32 bits
- r8: registro de 8 bits
- r/m8 : registro de cualquier tamaño o posición de memoria de 8 bits
- imm8 : operando inmediato de 8 bits
- reg8 : registro de 8 bits
- Sreg : registros segmento → CS,DS,ES,FS,GS,SS
- The moffs8, moffs16, and moffs32 operands specify a simple offset relative to the segment base, where 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

<https://www.intel.es/content/www/es/es/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html>

3.7.4 Specifying a Segment Selector

The segment selector can be specified either implicitly or explicitly. [...] The processor automatically chooses a segment according to the rules given in Table 3-5.

SS Any memory reference which uses the ESP or EBP register as a base register.

DS All data references, except when relative to stack or string destination.



la sintaxis del lenguaje ASM de los manuales no es la sintaxis de AT&T sino Intel → mnemónico operando_destino, operando_fuente

6.6. Mnemónicos Básicos (Explicados)

6.6.1. 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

- **imul**: multiplicación de números enteros con signo
 - Puede tener 1,2 o 3 operandos
 - **imull Etiqueta**: $R[\%edx]:R[\%eax] \leftarrow M[Etiqueta] \times R[\%eax]$

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

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

- **div**: división de números naturales, sin signo
- **idiv**: división de números enteros con signo
 - Puede tener 1,2 o 3 operandos

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

6.6.2. Extensión del 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

6.6.3. Cambio de tamaño

- **movzbw src,Reg** → Mov Byte to Word
- **movzbl src,Reg** → Mov Byte to Long
- **movzwl src,Reg** → Mov Word to Long

6.6.4. Operaciones Booleanas

- [NOT](#)
 - no flags
- [AND](#)
 - Clear CF,OF
 - Modifica SF,ZF,PF
- [OR](#)
 - Clear CF,OF
- [XOR](#)
 - Clear CF,OF
 - Modifica SF,ZF,PF

6.6.5. Procesamiento Condicional: CMP,TEST,SETcc

CMP

- [CMP](#)
- Modifica CF,OF,SF,ZF,PF,AF
- The condition codes used by the Jcc, CMOVcc, and SETcc instructions are based on the results of a CMP instruction
- Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the **SUB instruction**. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.
 - Es decir, hay que saber interpretar la instrucción de resta SUB y en concreto el flag OF para signed overflow y CF para unsigned overflow.
 - Ejercicios:
 - Realizar el [ejercicio instrucción CMP](#) y también en el guión de prácticas que se estudia los saltos realizar la interpretación manual de las instrucciones CMP.

SUB

- [SUB](#)
- Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.
- The SUB instruction performs integer subtraction.
- It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.
 - Es decir, realiza la resta tanto interpretando los operandos como números naturales y números con signo ya que la representación binaria del resultado es la misma. En caso de overflow utiliza el flag OF para la interpretación con signo y para el overflow sin signo el flag CF.

TEST

- TEST

- Computes the bit-wise logical **AND** of first operand (source 1 operand) and the second operand (source 2 operand)
- Clear CF,OF
- Modifica SF,ZF,PF

SETcc

- SETcc

- SETcc operand
- No modifica ningún flag. Modifica el operando si se cumple la condición.

MOV

- MOV

- La instrucción MOV no afecta a ningún flag del registro EFLAG

6.6.6. Saltos

Condicionales: Jcc

- Jcc

- Comprueba la condición y si se cumple se ejecuta el salto a la dirección referenciada en el campo de operando.
- jump short: el valor del operando es relativo al PC
- Chequea los flags CF, OF, PF, SF, and ZF.
- "less" and "greater": compara números con signo: jl, jle, jg, jge, etc...
- "above" and "below" : compara números sin signo: ja, jae, jb, jbe, etc...

- Antes de un salto condicional es necesario ejecutar algún tipo de instrucción donde la relación (aritmética, lógica, etc) de los operandos sea condición para la ejecución del salto. Instrucciones previas pueden ser: CMP y TEST.

Indirectos

Símbolo *:

El Símbolo asterisco para indicar indirección en los saltos y diferenciarlos del direccionamiento relativo.

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

En cambio en los movimientos MOV no hace falta el símbolo asterisco ya que no hay movimientos con direccionamiento relativo.

6.6.7. 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 bit entrante es el bit de signo el operando en EDX.
- **shr,shl**
- desplazamiento lógico: entran ceros
 - Ejemplos de multiplicación y división
- ROL,ROR : rotación hacia la izquierda y rotación hacia la derecha.
 - el bit que sale fuera se copia en CF
 - Aplicación: conversión endianess

6.6.8. 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
```

6.7. Formato de Instrucción: ISA Intel x86-64

- Apéndice [Formato Instrucción](#)

6.8. Subrutinas

6.8.1. Referencias

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

6.8.2. 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.
- En la sesión de prácticas 5 se programarán subrutinas.

6.8.3. 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 shrot.
 - 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  
https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Debugging-.html
```

```
Options.html#Debugging-Options
*/
// 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, N=1, longitud;
    // Llamada a las funciones printf() y sumMtoN()
    longitud=printf("El resultado de la suma es %d \n", sumMtoN(M,N));
    printf("El número de caracteres de la cadena anterior impresa es
%d\n",longitud);
    // La evaluación de sumMtoN consiste en: llamar a la función y capturar el
    valor de retorno.
    // Llamada a la función exit
    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;
}
```

- La **evaluación** de sumMtoN() consiste en obtener el **valor de retorno** de la ejecución de la función *sumMtoN()*
- printf → sumMtoN : printf llama a sumMtoN , se evalua sumMtoN y se imprime el resultado de evaluar la función *sumMtoN()*.
- El valor de retorno de la función printf() es el número de caracteres de la cadena impresa. Abrir un terminal y ejecutar el manual "man 3" para leer el apartado "RETURN VALUE".

6.8.4. Anidamiento de Funciones

- Anidamiento de llamadas: init() → main() → sumMtoN() → printf() → write()
- El shell el sistema operativo GNU/linux llama a la función principal main() del programa de usuario, que a su vez llama a la función printf() de la librería libc , la cual a su vez llama a la función de usuario sumMtoN() y a la función write() del sistema operativo.
- Retorno:



6.8.5. Pila/Frame

- Ver concepto de pila en el [Apéndice](#).
- La pila es una *sección* del programa en ejecución en la memoria principal. A diferencia de la sección de datos y la sección de instrucciones la pila se crea en tiempo de ejecución ,no durante la carga en memoria.
- Los argumentos M y N de la función *sumMtoN()* se pasan de la función *main()* a la función *sumMtoN()* a través de la pila.
- Partición de la pila en frames: Cada función del programa tiene su zona limitada dentro del *segmento* de pila. A cada zona limitada asociada a cada función se le denomina **frame**. Por lo tanto en la sección pila se irán anidando frames cada vez que se llame a un función, e irán desapareciendo frames cada vez que se retorne de un función.
 - La función *main()* crea su frame cuando es llamada por el shell y la función *sumMtoN* crea su propio frame cuando es llamada por *main()*.
 - Los frames se apilan según se anidan las llamadas a subrutinas. Y se desactivan según retornan.
 - 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 la parte alta del frame (top) por el puntero ESP. Los términos bajo y alto hacen referencia a la dirección del **apilamiento** y no a direcciones de memoria. Parte baja y alta del apilamiento.

6.8.6. 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
```

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

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

6.8.8. 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

```

6.8.9. Registros a Preservar

Rutina Llamante: arquitectura i386

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: arquitectura i386

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.

6.8.10. Retorno de la subrutina

- La última instrucción de la subrutina es **RET** cuya ejecución por la Unidad de Control de la CPU realiza las siguientes órdenes:
 - a. **PC ← M[ESP]** : extrae de la pila la dirección de retorno guardada por la instrucción **CALL** y la carga en el Contador de Programa, por lo que se ejecutará el ciclo de instrucción de la instrucción posterior a **call sumMtoN**
 - b. Actualiza el stack pointer: **ESP ← ESP + 4**
- Es necesario que en el epílogo de la subrutina, antes de la ejecución de RET el stack pointer apunte a la dirección de la pila donde está almacenada la dirección de retorno.

6.8.11. Estado de la pila

Análisis

- La pila es una estructura dinámica cuyo estado (registros puntero EIP,EBP,ESP) cambian según se realizan llamadas y retornos de subrutinas. Razonar el contenido de dichos registros y verificarlo con el depurador GDB.

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
```

- Analizar el contenido de los registros EIP,EBP,ESP:
 - EIP: instruction pointer →
 - EBP: stack bottom pointer →
 - ESP: stack top pointer →

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*
- Analizar el contenido de los registros EIP,EBP,ESP:
 - EIP: instruction pointer →
 - EBP: stack bottom pointer →
 - ESP: stack top pointer →

Creación del nuevo frame *sumMtoN*

- Estado de la pila después de ejecutar:

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

- Analizar el contenido de los registros EIP,EBP,ESP:
 - EIP: instruction pointer →
 - EBP: stack bottom pointer →
 - ESP: stack top pointer →

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*
- Analizar el contenido de los registros EIP,EBP,ESP:
 - EIP: instruction pointer →
 - EBP: stack bottom pointer →
 - ESP: stack top pointer →

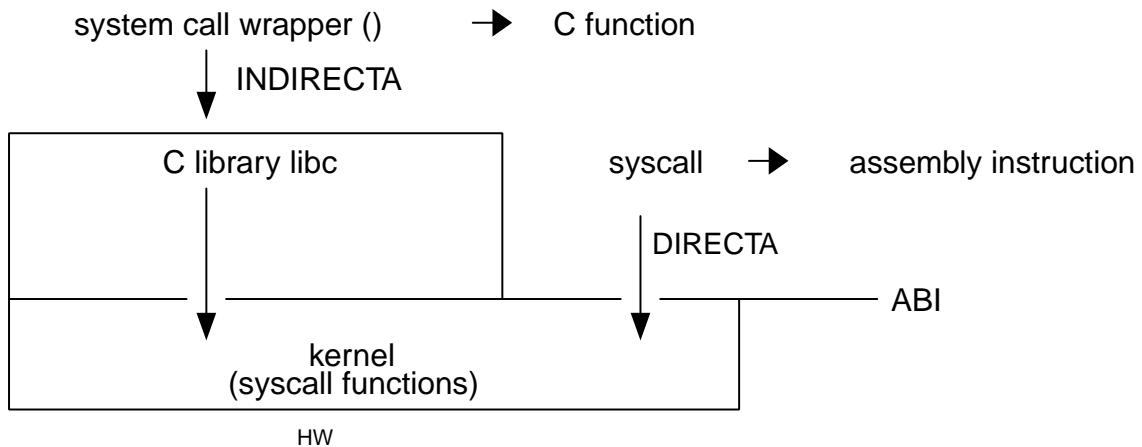
Posterior al salto de retorno

- 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`.
- Analizar el contenido de los registros EIP,EBP,ESP:
 - EIP: instruction pointer →
 - EBP: stack bottom pointer →
 - ESP: stack top pointer →

6.9. Llamadas al Sistema Operativo

6.9.1. 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



6.9.2. Arquitectura amd64

Códigos de llamada

- */usr/include/asm/unistd_64.h*: declaración de macros con el código de la llamada en la arquitectura x86-64
- */usr/include/bits/syscall.h*: macros antiguas también válidas en la arquitectura x86-32

Argumentos

- Ejemplo arquitectura **amd64**
 - Para pasar los argumentos no se utiliza la pila (memoria externa) sino los registros RPG (memoria interna de la CPU)
 - Los 6 primeros argumentos se pasan a través de los registros: [RAX-RDI-RSI-RDX-R10-R8-R9] previamente a la instrucción de la llamada **syscall**

```

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

Ejemplos amd64

- Ver en el [Apéndice](#)

6.9.3. Arquitectura i386

Norma

- ABI SystemV i386: Application Binary Interface (ABI)
 - La norma ABI SystemV i386 es la norma oficial donde se describe la interfaz binaria, es decir, como se comunican los distintos módulos de un programa a nivel binario, a nivel de lenguaje máquina.
 - La llamada al kernel se realiza mediante la instrucción **int \$0x80** donde int es el mnemónico de interrupción (se interrumpe el programa de usuario en ejecución para llamar a una función del kernel)

6.9.4. Códigos de llamada

- `/usr/include/asm/unistd_32.h`: declaración de macros con el código de la llamada
 - `/usr/include/bits/syscall.h`: macros antiguas también válidas
- Llamadas típicas:
 - **exit-fork-read-write-open-close-...**

| exit - terminate current process | | |
|---|-----------------------------|--|
| In | eax | 1 |
| | ebx | return code |
| Out | (This call does not return) | |
| fork - create child process | | |
| In | eax | 2 |
| Out | eax | 0 in the clone; process id of clone or EAGAIN or ENOMEM in caller |
| read - read from file or device | | |
| In | eax | 3 |
| | ebx | file descriptor |
| | ecx | address of the buffer to read into |
| | edx | maximum number of bytes to read |
| Out | eax | number of bytes actually read : EAGAIN : EBADF : EFAULT : EINTR : EINVAL : EIO : EISDIR |
| write - write to file or device | | |
| In | eax | 4 |
| | ebx | file descriptor |
| | ecx | address of the buffer to write from |
| | edx | maximum number of bytes to write |
| Out | eax | number of bytes actually sent : EAGAIN : EBADF : EFAULT : EINTR : EINVAL : EIO : ENOSPC : EPIPE |
| open - open, create, or truncate a file or device | | |
| In | eax | 5 |
| | ebx | address of zero-terminated pathname |
| | ecx | file access bits |
| | edx | file permission mode |
| Out | eax | file descriptor of opened file : EACCESS : EEXIST : EFAULT : EISDIR : ELOOP : EMFILE, : ENAMETOOLONG : ENFILE : ENOENT : ENODEV : ENODIR : ENOMEM : ENOSPC : ENXIO : EROFS : ETXTBSY |
| close - close a file or device | | |
| In | eax | 6 |
| | ebx | file descriptor |

| exit - terminate current process | | |
|---|---|---|
| Out | eax | zero for success : EBADF |
| waitpid - wait for a processes to terminate | | |
| In | eax | 7 |
| | ebx | process id of the process to wait for |
| | ecx | 0, or address of buffer to hold exit state |
| | edx | option flags : 0 : WNOHANG : WUNTRACED |
| Out | eax | pid of finished process : ECHILD : EINVAL : ERESTART |
| | ecx | exit state of finished process, if non-zero value was input in ecx |
| create - create a file | | |
| In | eax | 8 |
| | ebx | address of zero-terminated pathname |
| | ecx | file permission mode |
| Out | eax | file descriptor of opened file : EACCESS : EEXIST :EFAULT : EISDIR : ELOOP : EMFILE, : ENAMETOOLONG : ENFILE : ENOENT : ENODEV : ENODIR : ENOMEM : ENOSPC : ENXIO : EROFS : ETXTBSY |
| Note | This call is identical to calling open with access bits O_CREATE : O_WRONLY : O_TRUNC | |
| link - create a hard link to a file | | |
| In | eax | 9 |
| | ebx | address of zero-terminated pathname of existing file name |
| | ecx | address of zero-terminated pathname of new name |
| Out | eax | 0 : EACCESS : EIO : EPERM : EEXIST :EFAULT : ELOOP : EMLINK : ENAMETOOLONG : ENOENT : ENOMEM : ENOSPC : ENOTDIR : EPERM : EROFS : EXDEV |
| unlink - delete a name and remove file when not busy | | |
| In | eax | 10 |
| | ebx | address of zero-terminated pathname of existing file name |
| Out | eax | 0 : EACCES :EFAULT : EIO : EISDIR : ELOOP : ENAMETOOLONG : ENOENT : ENOMEM : ENOTDIR : EPERM : EROFS |
| execve - execute a program | | |
| In | eax | 11 |
| | ebx | address of zero-terminated pathname of program |
| | ecx | address of zero-terminated list of addresses of zero-terminated argument strings |
| | edx | address of zero-terminated list of addresses of zero-terminated environment strings |

| exit - terminate current process | | |
|----------------------------------|-----|--|
| Out | eax | If success, no return because the new program inherits resources and overwrites caller; otherwise: E2BIG : EACCES : EINVAL : EIO : EISDIR : ELIBBAD : ELOOP : ENFILE : ENOEXEC : ENOENT : ENOMEM : ENOTDIR :EFAULT : ENAMETOOLONG : EPERM : ETXTBUSY |
| chdir - change working directory | | |
| In | eax | 12 |
| | ebx | address of zero-terminated pathname of existing directory |
| Out | eax | 0 : EACCES : EBADF :EFAULT : EIO : ELOOP : ENAMETOOLONG : ENOENT : ENOMEM : ENOTDIR |

- File descriptors

- 0 (STDIN): The standard input for the terminal device (normally the keyboard). Diferenciar la macro STDOUT_FILENO de la macro stdin.
- 1 (STDOUT): The standard output for the terminal device (normally the terminal screen). Diferenciar la macro STDIN_FILENO de la macro stdout.
- 2 (STDERR): The standard error output for the terminal device (normally the terminal screen)

Como pasar los argumentos

- Para pasar los argumentos no se utiliza la pila (memoria externa) sino los registros RPG (memoria interna de la CPU)
- Los 6 primeros argumentos se pasan a través de los registros: [EBX-ECX-EDX-ESI-EDI-EBP] previamente a la instrucción de la llamada `int $0x80`

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

- Pasar los argumentos 1º-2º-3º-4º-5º-6º a través de los registros EBX-ECX-EDX-ESI-EDI-EBP en el orden indicado en el prototipo de la función en C:
- Ej: llamada a la *syscall exit* del kernel desde un módulo fuente en lenguaje ASM:

- `man 3 exit`
- `syscall(exit_code,int status)`
- módulo asm

```
mov $1,%eax
mov $status_value,%ebx
int $0x80
```

- Ej: llamada a la *syscall write* del kernel desde un módulo fuente en lenguaje ASM:
- `man 2 write`
- `syscall(write_code,int fd, const void *buf, size_t count)`

- módulo asm

```
mov $4,%eax
mov $1,%ebx
mov $buffer_address_label,%ecx
mov size,%edx
int $0x80
```

- Valor de Retorno
 - El valor de retorno se pasa a través del registro *EAX*

LLamadas a las funciones de la librería standard de C

- Desde ASM se puede llamar a las funciones de la librería de C instalada en linux: *libc*
- Para llamar a la función *printf()* utilizamos la instrucción **call printf**
- Los argumentos de la función se pasan previamente a la llamada.
 - Los argumentos se pasan a través de la **pila** en el sentido Dcha→Izda a como estan definidos.
- Es necesario linkar el módulo objeto con la librería *libc*
- Ej:
 - Programación C

```
planetas = 9;
printf("El número de planetas es %d \n", planetas);
```

- Programación ASM

```
.section .data
cadena:
.asciz "El número de planetas es %d \n"
planetas:
.long 0
.section .text
_start:
    movl $9,planetas
    push planetas
    push cadena
    call printf
    call exit
.end
```

Línea de comandos

- Process Initialization
- en el kernel está declarada la función main(): **extern int main (int argc , char* argv[] , char* envp[])** ;
 - declaración y definición del módulo principal

- *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.
- Stack Initialization

Table 13. Convenio ABI: Stack

| Stack Reference | Interpretation |
|-----------------|---------------------------------|
| | arguments strings |
| 4n(%esp) | - pointer to nº string |
| 8(%esp) | - pointer to 2º argument string |
| 4(%esp) | - pointer to 1º argument string |
| 0(%esp) | - argc |

6.9.5. Ejemplo

- Introducir los datos del programa suma.s a través de a línea de comandos

```
### función: sumar dos números enteros de un dígito.
###      los sumandos se pasan a través de la línea de comandos
## gcc -m32 -nostartfiles -g -o sum_input sum_input.s
## run 5 7
## x /a *(char**)(%esp+4) -> 0xfffffd0a4: 0xffffd26e
## x /c *(char**)(%esp+4) -> 0xfffffd26e: 47 '/'
## x /s *(char**)(%esp+4) -> 0xfffffd26e:
"/home/candido/tutoriales/as_tutorial/algoritmos_x86-32/basicos/sum_input"
## p /s *(char**)(%esp+4) -> 0xfffffd26e
"/home/candido/tutoriales/as_tutorial/algoritmos_x86-32/basicos/sum_input"
## x /s *(char**)(%esp+8) -> 0xfffffd2b7: "5"
## x /s *(char**)(%esp+12) -> 0xfffffd2b9: "7"

.section .text
.globl _start
_start:

## instrucciones aclaratorias

leal 8(%esp),%eax      #eax contiene argv[1] la dirección de la pila que
contiene el pointer al argumento string
movl 8(%esp),%ebx      #ebx tiene el contenido de la pila= dirección del
string
xor %ecx,%ecx
movb (%ebx),%cl        #caracter ASCII

## string argument pointers
movl 8(%esp),%eax      #eax tiene el contenido de la pila= dirección del
```

```

string. argv[1]
    movl 12(%esp),%ebx      #eax tiene el contenido de la pila= dirección del
string. argv[2]
    ## fetch string indirect
    ## convert ascii numbers to values
    xor %ecx,%ecx
    xor %edx,%edx
    movb (%eax),%cl        # indirección para acceder al string referenciado
por argv[1]
    movb (%ebx),%dl        # indirección para acceder al string referenciado
por argv[1]
    subl $0x30,%ecx
    subl $0x30,%edx

    push %ecx
    push %edx

    call suma

## salida
    movl %eax,%ebx
    movl $1, %eax      #1 is the exit() syscall
    int $0x80

```

```

### Función que calcula el máximo entre dos valores
.type suma, @function
.section .text
suma:
    ## prologo
    push %ebp
    movl %esp,%ebp
    subl $1,%esp      #reserva de memoria
    push %ebx
    push %edi
    push %esi
    ## captura de argumentos
    movl 8(%ebp),%eax      #1º argumento
    movl 12(%ebp),%ecx      #2º argumento
    ## cuerpo
    addl %ecx,%eax      #
    ## guardar resultado
    ## el resultado está en EAX
salto:
    ## epilogo
    pop %esi
    pop %edi
    pop %ebx
    mov %ebp,%esp      # frame anterior
    pop %ebp

```

ret

recuperar dirección de retorno

II Unidades Básicas: Procesador Central, Unidad de Memoria, Mecanismos Entrada/Salida.

Chapter 7. Procesador Central

7.1. Temario

- 7. Conjunto de instrucciones
 - a. Arquitecturas CISC, RISC y VLIW
 - b. Fases de ejecución de una instrucción
 - c. Ruta de datos

7.2. Refs

- Apuntes : Tema 2: Arquitectura von Neumann (unidad de control)
- Libro de Texto: Estructura y Organización de Computadores .William Stalling. Capítulo 12.

7.3. Introducción

- El objetivo principal de la CPU es la implementación del *ciclo de instrucción*. Es el soporte hardware para poder llevar a cabo todas las operaciones que conllevan las instrucciones de un programa.
- Unidad Central de Proceso (CPU) o Procesador.
 - Nombres: Procesador, microprocesador (El componente electrónico básico es el transistor con un tamaño en sus orígenes del orden de la micra), ..
 - Central porque la computadora tiene varios procesadores: Por ejemplo el controlador de la memoria y los controladores de los periféricos.
- Arquitectura Von-Neumann.
 - La CPU es una de las unidades básicas que conforman la arquitectura Von-Neumann (CPU-MP-IO) y Buses.
- *Microarquitectura*
 - Las unidades básicas de la CPU son: Unidad de Control (UC), y la Ruta de Datos (ALU,FPU,MMU,Registros y circuitos de enrutamiento como multiplexores, comutadores, etc).
- El ciclo de instrucción puede ser secuencial o segmentado, permitiendo el solapamiento en el tiempo de la ejecución de más de una instrucción (técnicas de paralelismo a nivel de instrucción, ILP)
- La CPU se puede ver desde el punto de vista del programador o desde el punto de vista del diseñador electrónico.
- Desde el punto de vista del *programador* interesa conocer:
 - La Arquitectura del Repertorio de Instrucciones (ISA)
 - Registros: registros de propósito general accesibles por el programador (acumulador, registro índice, punteros pila, etc), registro de estado, registros de coma flotante, registros multimedia, registros de segmentación de memoria, registros no accesibles como el contador de programa, tamaño de los registros, etc
 - Modos de Funcionamiento de la CPU: modo superusuario, modo usuario, modo interrupción
- Técnicas HW de optimización de la ejecución de un programa (**Performance**)
 - **Segmentación-Pipelining**: organizar el ciclo de instrucción en fases o segmentos y ejecutarlos en paralelo.
 - **Ejecución fuera de Orden OoO**: Run time
 - **Renombre de Registros**: Compiler & Run time

- Branch Predictor: Run time

7.4. Conjunto de Instrucciones

7.4.1. Arquitectura (ISA)

- Recordatorio de la primera parte de la asignatura:
 - Temas: arquitectura von neumann, representación de datos, operaciones aritmético-lógicas, representación de las instrucciones y programación en lenguaje ensamblador.
- Instruction Set Arquitecture (ISA)
 - La arquitectura del repertorio de instrucciones define: códigos de operación, tipos de operando, modos de direccionamiento, etc
 - Son las instrucciones máquina ejecutables directamente por la CPU en código binario.: *lenguaje máquina*
 - La instrucción a ejecutar está almacenada en código binario en el registro RI de la Unidad de Control.
- El repertorio de instrucciones está especificado en el manual del programador de la CPU:
 - Programamos en *lenguaje Ensamblador* en lugar de en *lenguaje máquina*
 - El manual contiene la definición de la Arquitectura del Repertorio de Instrucciones.
 - el listado y descripción de todas las instrucciones ejecutables por el microprocesador
 - categorías de las instrucciones: transferencia(mov), control(jmpz,loop),aritméticas(add), lógicas(xor), i/o (in/out)
 - Mnemónicos del código de operación
 - Modos de direccionamiento: inmediato, directo, indirecto, desplazamiento
 - Tipos de datos: entero, real, alfanumérico
 - Formatos binarios
 - De las instrucciones: campos de operación, operando, modo direccionamiento
 - De los datos: complemento a 2, coma flotante

7.4.2. Ejemplos: Intel x86, Motorola 68000, MIPS, ARM

- Ver Apéndice Lenguajes Ensamblador

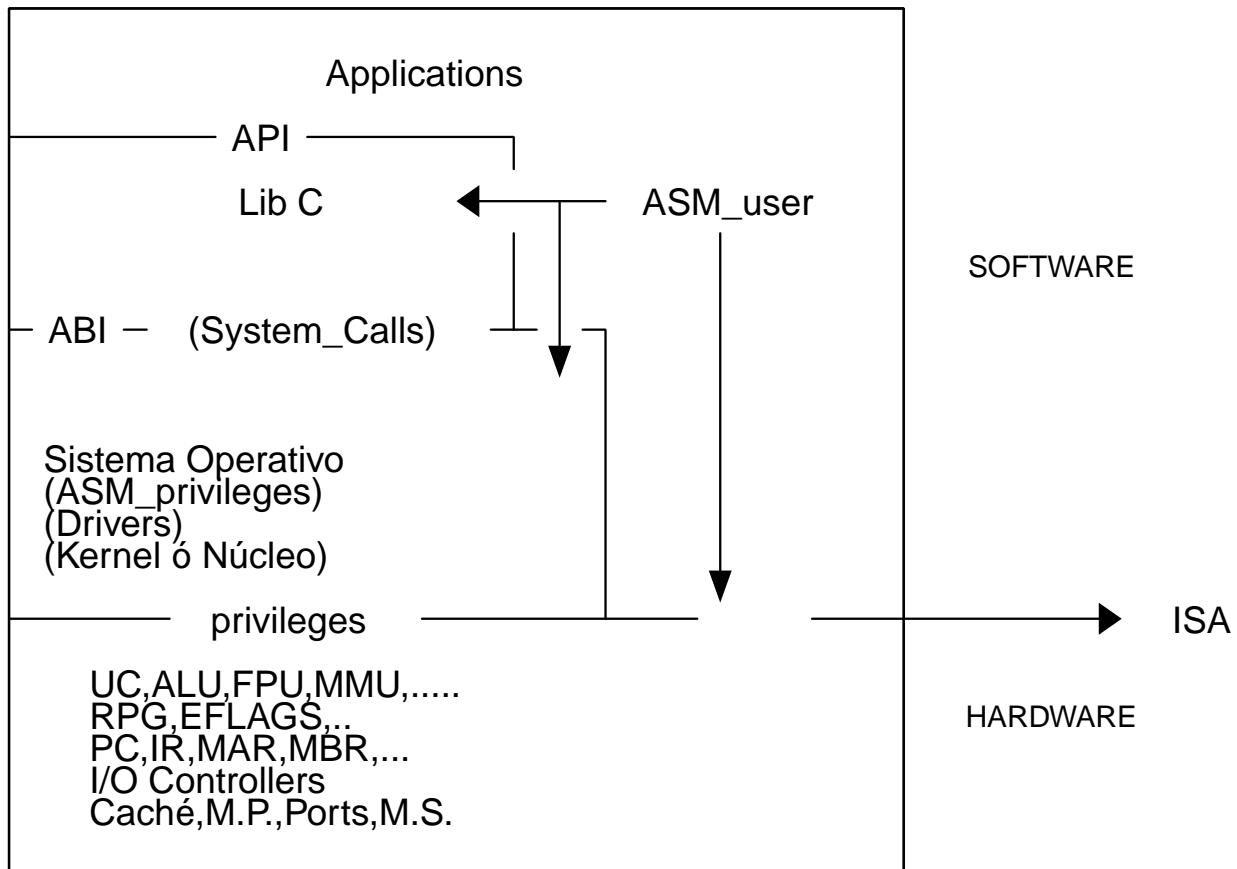
7.5. La Computadora desde el punto de vista del programador

7.5.1. Niveles o Capas de Abstracción

- El programador se abstrae (en parte) de la implementación del Hardware gracias al Sistema Operativo. El programador interactúa con el Sistema Operativo para acceder a los recursos HW de la computadora.
- Abstracción de la Máquina : mediante las instrucciones ISA / especificaciones ABI
 - **ABI** : "Application Binary Interface" . Es un documento que especifica las características binarias del software, es decir, el nivel más bajo del software. Por ejemplo especifica el convenio de llamadas a subrutinas, cómo está estructurada la pila, cómo realizar las llamadas al sistema, el formato binario del módulo objeto ejecutable, etc ... El compilador, linker y loader han de conocer la interfaz ABI con todo detalle.
 - Desde el punto de vista del programador de aplicaciones de bajo nivel: La interfaz con la máquina son

las llamadas al sistema (ABI) y el repertorio "user ISA"

- Abstracción a niveles superiores
 - La interfaz con la librería son los prototipos de las funciones de la librería (Application Programming Interface - **API**)
- Esquema con las Interfaces de las aplicaciones desarrolladas en **lenguajes de bajo nivel**:



- Desde el punto de vista del sistema operativo S.O.:
 - La interfaz con la máquina es **ISA (system isa y user isa)**
 - La interfaz con el programador es **ABI**
- Desde el punto de vista del programador
 - si no hay S.O. la interfaz con la máquina será equivalente a la del S.O.
 - si hay S.O. y librerías la interfaz con la máquina:
 - en lenguaje C : **API** y **ABI** específicos de C
 - en lenguaje ASM: **API** © y **ABI** específico de asm
 - La programación de bajo nivel requiere tener algunos conocimientos del Hardware de la máquina no siendo posible su completa abstracción. Por lo tanto es necesario estudiar la CPU desde el punto de vista del programador.
 - ¿Cuál sería el esquema de niveles o capas visto por los siguientes niveles de abstracción superiores?
- Escritorio
- Lenguaje de Programación Java

7.5.2. Compatibilidad Software

Compatibilidad

- Cada procesador tiene su repertorio de instrucciones
- Si dos procesadores tienen el mismo repertorio de instrucciones, es decir, la misma arquitectura, el módulo fuente en lenguaje ensamblador será compatible para los dos procesadores aunque la estructura interna de la CPU sea diferente: Ejemplo: Intel IA64 y AMD64

Ejemplos

- El programador necesita conocer el trío ARCH-KERNEL-LIBC
 - Arch se refiere a la arquitectura de la computadora → ISA
 - Kernel: núcleo del sistema operativo. Implementa las llamadas del sistema
 - Libc: librería para el programador de aplicaciones. Implementa las llamadas al sistema
 - Tanto el Kernel como la Librería tienen asociados sus interfaces de nivel alto (API) como de nivel bajo (ABI)
- Ejemplos arch/kernel/libc
 - amd64-linux-gnu
 - arm-linux-gnueabi

7.6. Fases de Ejecución de una Instrucción

7.6.1. Estructura

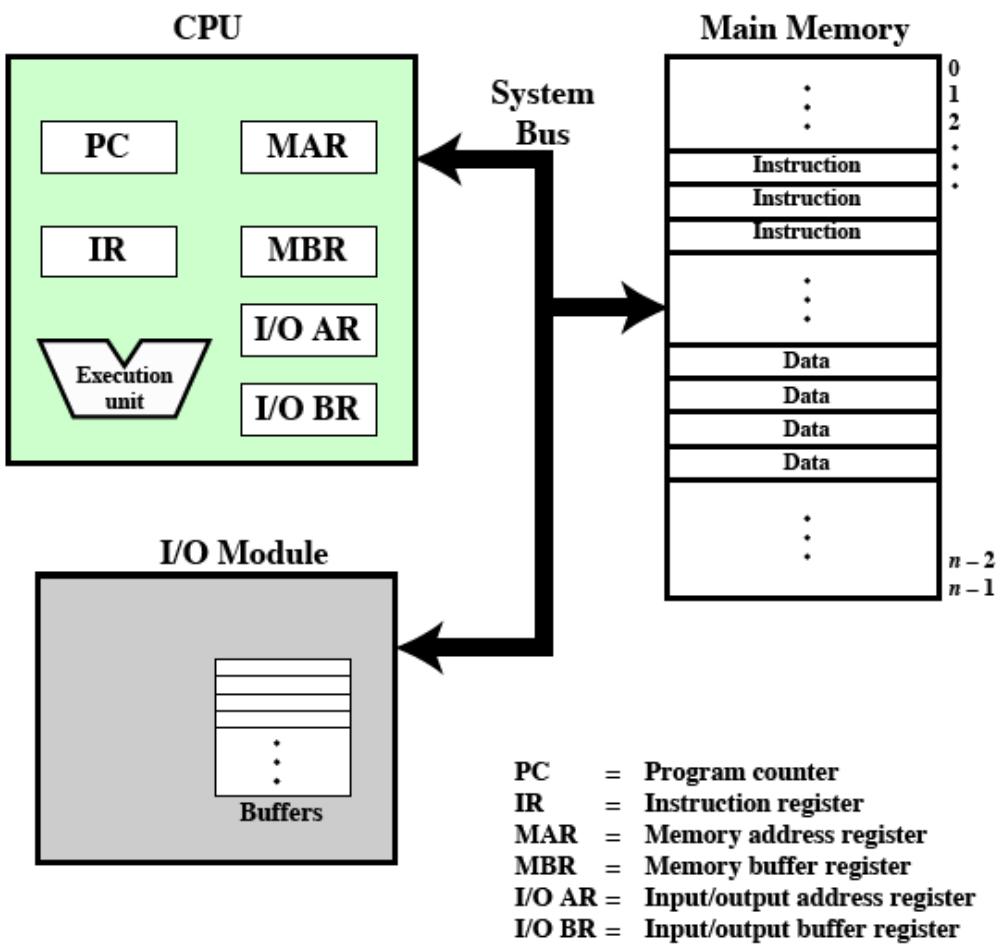


Figure 34. IAS_Architecture

7.6.2. Ciclo / Diagrama / Fases

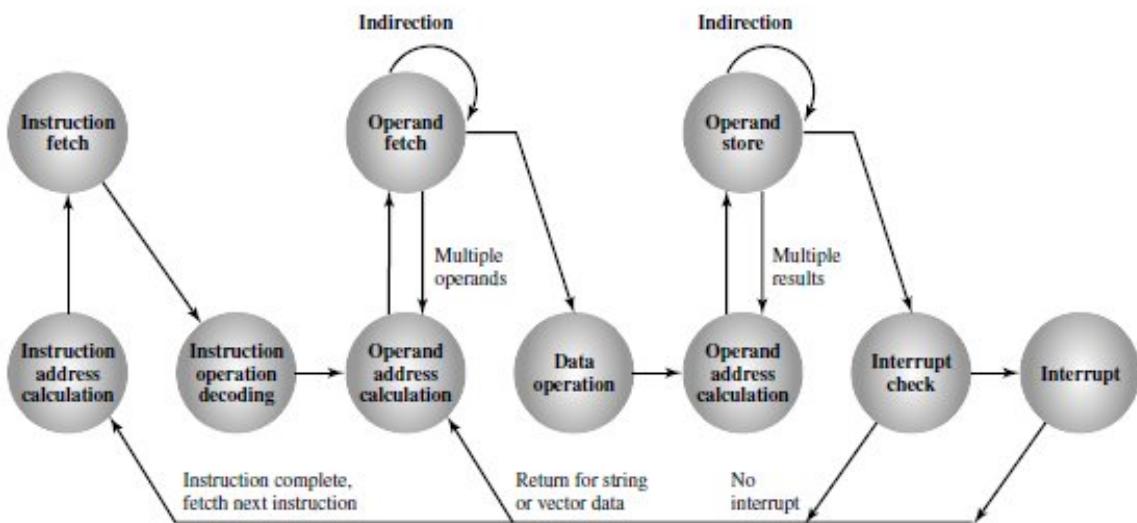


Figure 12.5 Instruction Cycle State Diagram

Figure 35. Diagrama de Estados

- Fases del ciclo de instrucción

- Fetch Instruction : FI

- Inicialmente hay que volcar al bus de direcciones de memoria el contenido del Contador de Programa (PC)
- Captar la instrucción
- $PC \leftarrow PC + 1$

- Instruction Decode : ID

- interpretar la instrucción

- Fetch Operand : OF

- captar datos, captar los operandos
- resolver la dirección efectiva

- Execute Instruction : EI

- procesar la instrucción con los datos

- Write Operand: WO

- almacencar el resultado
- resolver la dirección efectiva

- Interruption : II

- El programa puede ser interrumpido por la prioridad de ejecutar otro programa de atención a periféricos, etc.. Una vez atendida la interrupción el programa continua con el siguiente ciclo de instrucción.

- Next Instruction : NI

- Ciclo de instrucción

- Después de la fase de captación de la instrucción (FI) le sigue la fase de Ejecución (EI) ó la Fase de

determinación de la Dirección Efectiva del Operando y Obtención del operando (OF)

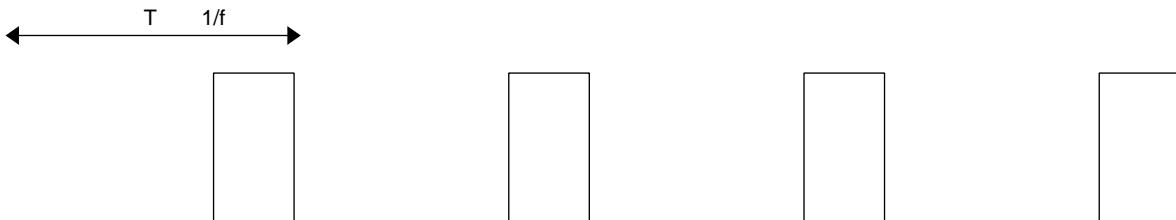
- Después de la fase de ejecución puede haber un ciclo de atención a una interrupción.

7.6.3. Ejemplo: máquina IAS de Von-Neumann

- Tema 2: [Arquitectura Von-Neumann](#)
- Cada instrucción de la computadora IAS se ejecuta siguiendo una secuencia de fases. Dicha secuencia se repite para cada instrucción y se conoce como el ciclo de instrucción de la unidad central de proceso (CPU).
- La unidad de control es la unidad de la CPU que implementa cada fase del ciclo de instrucción.
- La unidad de control controla la ruta de datos de la CPU mediante microordenes.
- Internamente está formada por el circuito generador de microordenes y por los registros : contador de programa y registro de instrucción.

Diagrama de Microoperaciones

- Microoperaciones: operaciones realizadas por la CPU internamente, al ejecutar una Instrucción Máquina.
 - Ejemplos: escribir en el registro MAR, orden de lectura a la MPrincipal, leer de MBR, interpretar de IR, incrementar PC, etc
 - Ejecución Síncrona con el reloj de la CPU:



- Flancos de reloj: Cambio de nivel $0 \rightarrow 1$ (positivos) o $1 \rightarrow 0$ (negativos)
- IAS no es síncrona: una microoperación no comienza con ningún patrón de tiempos.
- Descripción de las micro-operaciones: Register Transfer Language (RTL)

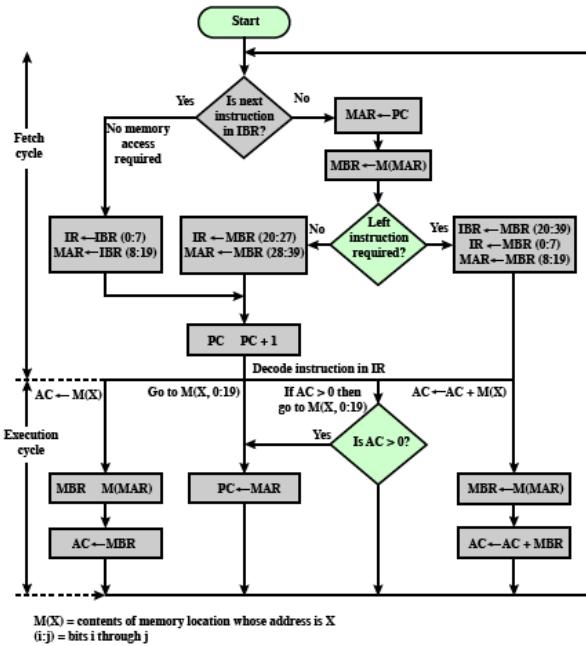


Figure 2.4 Partial Flowchart of IAS Operation

Figure 36. 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
 - $AC \leftarrow AC + M(x)$.

7.7. Microarquitectura: Unidades Funcionales

7.7.1. Introducción

- Se conoce con el nombre microarquitectura a la arquitectura interna del microprocesador.
 - La microarquitectura es el diseño e implementación del ciclo de instrucción del conjunto de instrucciones definido por ISA.
 - Ejemplos
 - Intel: 8051, x86
 - AMD: x86
 - ARM: Cortex

7.7.2. Implementación del ciclo de instrucción

- ¿Cómo implementar el ciclo de instrucción?

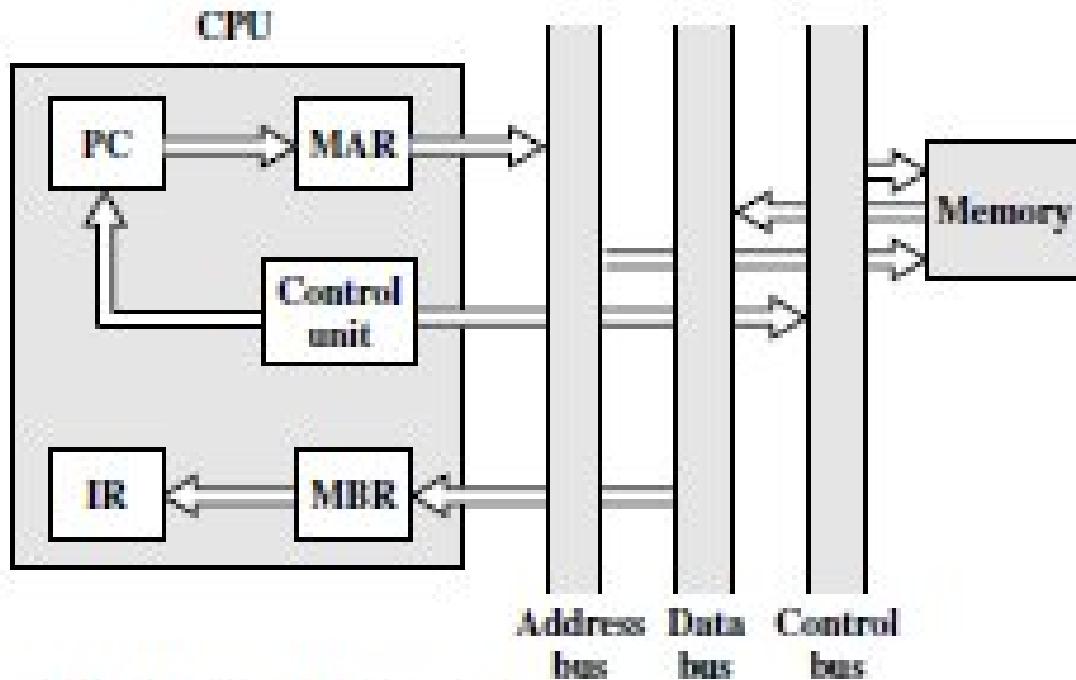
- Mediante un Circuito Electrónico Digital secuencial: Máquina de estados finitos FSM que implementa la secuencia del diagrama de estados y que recibe el nombre de Unidad de Control.
- La Unidad de Control es una secuencia de estados que van realizando las distintas fases del ciclo de instrucción.
- Las distintas fases del ciclo de instrucción utilizan distintas unidades funcionales como: registros, ALU, etc
- La interpretación de distintas instrucciones máquina dará lugar a diferentes secuencias de estados en la Unidad de Control.

7.7.3. Estructura de la CPU

- Tres recursos básicos: Unidad de Control, **Unidad de Ejecución** y Registros.
- Dos Bloques básicos de la CPU
 - Unidad de Control (UC) y la Ruta de Datos (DataPath).
- La unidad de control esta formada por
 - generador de las microoperaciones que implementan el ciclo de instrucción
 - registros: registro de instrucción IR, registro contador de programa PC
- La Ruta de Datos esta formada por
 - **Unidad de Ejecución UE :**
 - Unidad Aritmetico Lógica ALU: cálculos números enteros
 - Unidad de Punto Flotante FPU: cálculos números reales
 - Unidad Load/Store LSU: cálculos de la dirección efectiva y acceso a la memoria principal
 - Memory Management Unit (MMU): cálculo de la dirección efectiva FISICA de la MP. Traduce las direcciones virtuales de memoria utilizadas por la cpu en direcciones físicas de la memoria principal.
 - los Registros
 - Registros de propósito general GPR accesibles por el programador
 - Registros de estado SR

7.7.4. Fase de Captación

- Ejemplo: Microoperaciones de la Fase de captación del ciclo de instrucción.
 - Se realiza la lectura de una instrucción mediante las siguientes acciones que son activadas por la Unidad de control:
 - El Contador de Programa (PC) o Instruction Pointer (IP) contiene la dirección de referencia de la instrucción a captar
 - El Memory Address Register (MAR) se carga con el contenido del (PC)
 - El bus de direcciones del sistema se carga con el contenido de MAR
 - Se vuelca el contenido de la dirección apuntada al Buffer i/o de memoria, de ahí al bus de datos transfiriéndose así al Memory Buffer Register (MBR)



MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Figure 12.6 Data Flow, Fetch Cycle

Figure 37. Flujo de Datos. Ciclo de Captación

- Secuencia de las microordenas en el ejemplo:
 - a. MAR → address bus
 - b. UC → control bus
 - c. data bus → MBR
 - d. MBR → IR y UC → PC
 - e. al finalizar la ejecución: PC → MAR

7.7.5. Perspectiva de la CPU

- Divimos la CPU en 5 unidades:
 - Unidad de Control (UC)
 - Unidad de Ejecución (UE)
 - Registros : de Propósito General (rax,mmx,sse,xmm,...), control (usuario,superusuario,paginación,interrupción,...) y status (rflags, ..).
 - Los registros de control no son accesibles por el usuario, son accesibles por el sistema operativo.
 - Memoria Cache L0
 - Memory Management Unit (MMU)
 - Reloj para sincronizar las tareas: facilita el diseño del Hardware.

Unidad de Control

- The control unit (sometimes called the fetch / decode unit) is responsible for retrieving individual instructions from their location in memory, then translating them into commands that the CPU can understand. These commands are commonly referred to as machine-language instructions, but are sometimes called **micro-operations**, or UOPs. When the translation is complete, the control unit sends the UOPs to the execution unit for processing.
- Señales de control de la UC
 - Señales digitales binarias

Unidad de Ejecucion (EU)

- The execution unit is responsible for performing the third step of the instruction cycle, namely, executing, or performing the operation that was specified by the instruction.
- Incluye: ALU+FPU+LSU+RPG
 - Operaciones: Aritméticas, Lógicas, Transferencia,

Ruta de Datos

- Es la ruta que realizan los datos (instrucciones, campos del formato de instrucciones, operando, dirección, etc ...) a través del procesador, internamente al procesador, dirigidos por la Unidad de Control.
- Es necesario interconectar las distintas unidades y subunidades de la CPU para poder transferir y procesar los bits y conjuntos de bits entre ellas.
- Los microcomandos de la UC en forma de señal transportan y procesan dichos datos.
 - Ejemplos de microcomandos: abrir puerta, conectar bus, multiplexar datos, etc ...microordenes de control del hardware
 - Dicho transporte y procesamiento dependerá de la interpretación de la instrucción en ejecución y del diseño de la microarquitectura.
- Los componentes básicos de la Ruta de Datos son :
 - Unidades de transporte: BUS, conmutador, multiplexor, etc
 - Unidad de memoria: cálculo de la dirección efectiva, interfaz con la memoria externa
 - Unidades de procesamiento: ALU
 - Unidades de almacenamiento: registros
- RTL: Register Transfer Language
 - Lenguaje para indicar las acciones de transporte, procesamiento y almacenamiento.
 - $AC \leftarrow [PC] + M[CS:SP]$
- Esquema de la Ruta de Datos

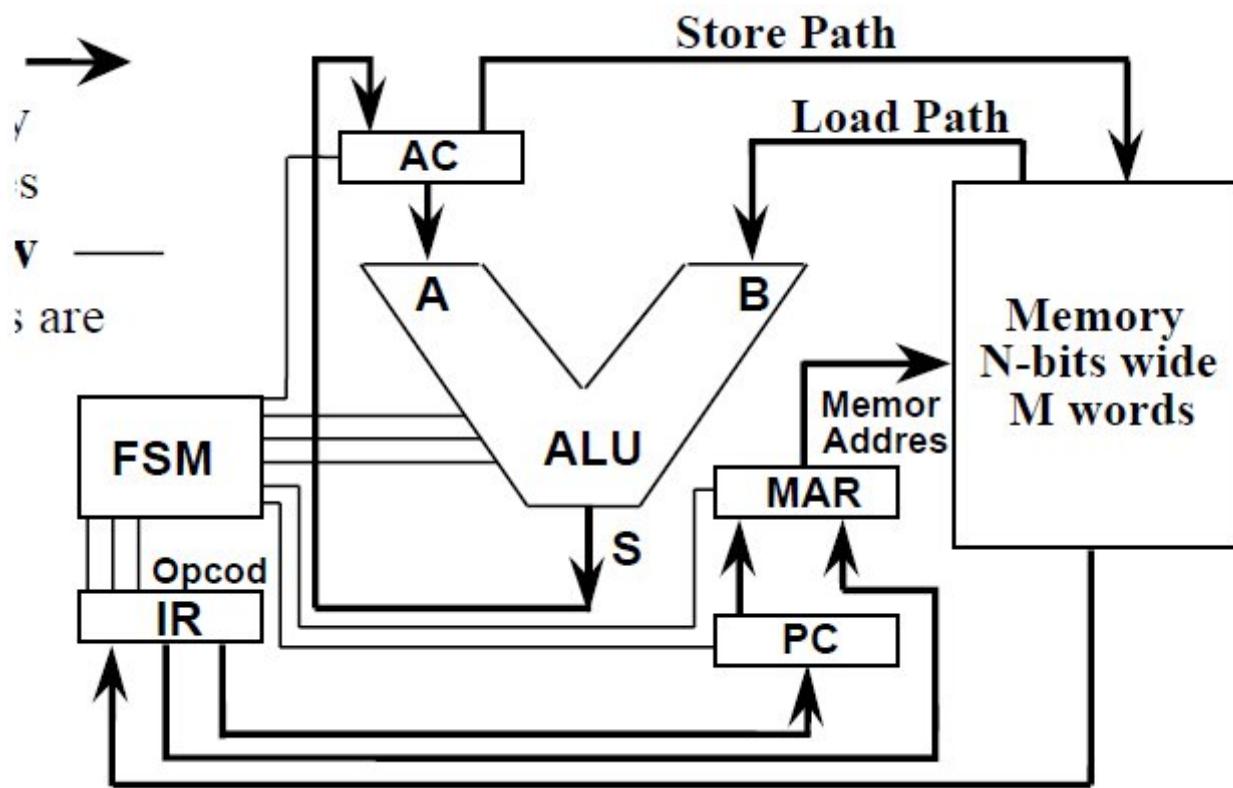


Figure 38. Datapath

- Líneas gruesas: bus de datos
- Líneas finas: bus de control → chip select, microorden sumar, cargar registro, etc ..
- Ver applet de la ruta de datos del apartado Imágenes
- Diseño del datapath
 - determinar que microunitades son necesarias
 - cómo conectarlas
 - Qué microseñales accionar y cuándo en cada microoperación. Paralelismo a nivel de microoperaciones
 - ubicación y temporización de los datos según la secuencia del diagrama de estados de la UC
 - $AC \leftarrow [PC] + M[CS:SP] \Rightarrow$ microoperaciones asociadas y diagrama de tiempos

7.7.6. Unidad de Control Microprogramada

- Unidad de Control Microprogramada vs Cableada
- Microcableada: El secuenciador o FSM de la unidad de control ejecuta *directamente* las instrucciones en código máquina almacenadas en la memoria principal
- Microprogramada:
 - Las instrucciones máquina (ISA) almacenadas en la memoria principal y cuya secuencia constituye el **código máquina** del programa del usuario no son ejecutadas directamente por la UC. En su lugar cada instrucción en código máquina es traducida en una secuencia de **microinstrucciones** y cada microinstrucción genera las microoperaciones o microseñales de la unidad de control que conforman el ciclo de instrucción.
 - La secuencia de microinstrucciones asociadas a una microinstrucción constituye el **microcódigo** que se encuentra almacenada en una memoria de sólo lectura (Read Only Memory ROM) interna de la Unidad de control.
 - Cambiando o añadiendo microcódigo a nuestra Unidad de Control conseguimos nuevas arquitecturas

ISA de una manera más flexible que con la unidad de control cableada.

- [microcode](#)

7.8. Arquitecturas CISC/RISC

7.8.1. Introducción

- CISC: Complex Instruction Set Computer
- RISC: Reduced Instruction Set Computer
- CISC y RISC son dos filosofías de diseño de un computador, dos arquitecturas.

CISC

- Ejemplos: Motorola 68k, Intel x86.
- Instrucciones de varios bytes y no uniformes.
- Necesita un HW complejo que ocupa mucho espacio y necesita muchos ciclos de reloj.
- La arquitectura del lenguaje ensamblador está próxima a un lenguaje de alto nivel como el lenguaje C por lo que facilita la tarea a los compiladores y a los programadores de lenguaje ensamblador.
- En cambio complica el diseño e implementación de elementos hardware como la CPU.

RISC

- Ejemplos: PowerPC, ARM, MIPS and SPARC
- Apuesta por un Hardware sencillo por lo que las instrucciones han de ser sencillas, regulares.
 - Un HW sencillo es rápido y ocupa poca área del chip.
- Inconveniente: Gran número de accesos a memoria para capturar las instrucciones, los operandos y el resultado.
 - Solución: incrementar la memoria interna: el número de Registros internos y la memoria caché. Para lo cual hay espacio debido al HW sencillo.

Cuestiones

- Qué arquitectura optimiza el tamaño de bytes del programa
- Qué arquitectura optimiza el tiempo de ejecución del cada instrucción
- Qué arquitectura optimiza el tamaño y coste de fabricación de la CPU
- Qué arquitectura optimiza el consumo
- Qué arquitectura optimiza el número de capturas a memoria. ¿Existe independencia entre captura y ejecución de instrucciones?

SW

- Un programa ensamblador de una arquitectura RISC tiene más instrucciones que un CISC
- Cada instrucción RISC se ejecuta en menor tiempo que una CISC.

7.8.2. Tabla Comparativa

- [RISC vs CISC](#)

7.9. Instruction Level Parallelism (ILP)

- [wikipedia](#)
 - Instruction-level parallelism (ILP) es la medida de cuantas instrucciones de un programa pueden ser ejecutadas simultáneamente. El solapamiento de la ejecución de las instrucciones recibe el nombre de instruction level parallelism (ILP)
 - Son dos los mecanismos para conseguir el ILP
 - Hardware
 - Software
- Técnicas de diseño de microarquitecturas que persiguen un solape ILP
 - VLIW
 - Superscalar
 - Pipelining (Segmentación)
 - Out-of-order execution
 - etc

7.9.1. VLIW vs Superscalar

VLIW

- Very Long Instruction Words
- La CPU contiene múltiples Unidades de Ejecución
- Una palabra contiene tantas instrucciones como unidades de ejecución.
 - A la palabra se le denomina Instruction Word, la cual contiene múltiples instrucciones máquina.
 - El *compilador* crea las Instrucciones Word con las múltiples instrucciones **asignando** a cada una de ellas una Unidad de Ejecución distinta.
 - Múltiples Instrucciones en Paralelo

Superscalar

- La arquitectura superescalar significa que la CPU tiene múltiples Unidades de Ejecución (UE), no confundir con múltiples núcleos (core), y es la *propia CPU* la que **asigna** en tiempo de ejecución los recursos de la máquina a las distintas instrucciones .
- Dicha arquitectura permite la ejecución simultánea de múltiples instrucciones.
- Una CPU superscalar n-way significa que puede ejecutar simultáneamente n instrucciones.
- Superscalar no significa multinúcleo. Un único núcleo es superscalar.

Comparativa Superscalar-VLIW

- One of the great debates in computer architecture is static vs. dynamic. **static** typically means "let's make our compiler take care of this", while **dynamic** typically means "let's build some hardware that takes care of this". Each side has its advantages and disadvantages. the compiler approach has the benefit of time: a compiler can spend all day analyzing the heck out of a piece of code. however, the conclusions that a compiler can reach are limited, because it doesn't know what the values of all the variables will be when the program is actually run. As you can imagine, if we go for the hardware approach, we get the other end of the stick. there is a limit on the amount of analysis we can do in hardware, because our resources are much more limited. on the other hand, we can analyze the program when it actually runs, so we have complete knowledge of all the program's variables.

- **VLIW** approaches typically fall under the "static" category, where the compiler does all the work.
- **Superscalar** approaches typically fall under the "dynamic" category, where special hardware on the processor does all the work. consider the following code sequence:

```
sw $7, 4($2)
lw $1, 8($5)
```

\$ significa direccionamiento directo
() direccionamiento indirecto indexado

- suppose we can run two memory operations in **parallel** [but only if they have **no dependencies**, of course]. are there dependencies between these two instructions? well, it depends on the values of \$5 and \$2. if \$5 is 0, and \$2 is 4, then they depend on each other: we must run the store before the load.
 - in a VLIW approach, our compiler decides which instructions are safe to run in parallel. there's no way our compiler can tell for sure if there is a dependence here. so we must stay on the safe side, and dictate that the store must always run before the load. if this were a bigger piece of code, we could analyze the code and try to build a proof that shows there is no dependence. [modern parallelizing compilers actually do this!]
 - if we decide on a SUPERSCALAR approach, we have a piece of hardware on our processor that decides whether we can run instructions in parallel. the problem is easier, because this dependence check will happen in a piece of hardware on our processor, as the code is run. so we will know what the values of \$2 and \$5 are. this means that we will always know if it is safe to run these two instructions in parallel.
 - Hopefully you see some of the tradeoffs involved. dynamic approaches have more program information available to them, but the amount of resources available for analysis are very limited. for example, if we want our superscalar processor to search the code for independent instructions, things start to get really hairy. static approaches have less program information available to them, but they can spend lots of resources on analysis. for example, it's relatively easy for a compiler to search the code for independent instructions.

7.9.2. Pipeline (Segmentacion)

- Pipeline: cauce o tubería.
- Ejemplo de Lavado de coches
 - Fases: Humedecer - Enjabonar - Cepillar - Aclarar - Secar - Abrillantar
- Máquina Secuencial
 - Cola de coches ante la máquina
 - Si un coche está en cualquiera de las fases no entra el siguiente coche.
 - El intervalo de tiempo de salida de coches será la suma de todas las fases. ¿Cada cuanto tiempo sale un coche del lavadero?
 - **Throughput (Producción):** Número de coches de salida por unidad de tiempo
- Segmentación frente a Secuencial.
 - En lugar de tener una máquina que realice todas las fases tenemos máquinas independientes que realizan cada fase.
 - El intervalo de tiempo de salida de coches será el de la duración de la fase de mayor duración.
 - El throughput, del número de coches atendidos por unidad de tiempo, aumenta.
- Flujo de Instrucciones con segmentación en 2 etapas

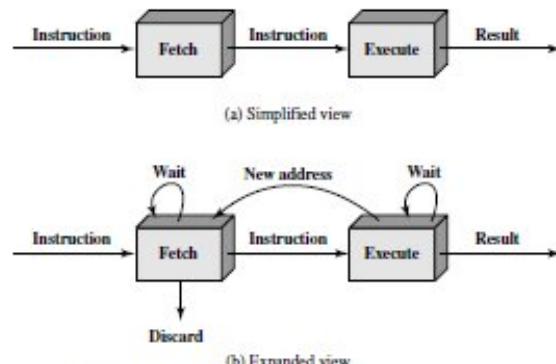


Figure 12.9 Two-Stage Instruction Pipeline

Figure 39. Segmentación en 2 etapas

- En caso de que los tiempos de cada etapa sean distintos o halla penalización por saltos en el flujo , se producirán tiempos de espera.

| | Time → | | | | | | | | | | | | | |
|---------------|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| Instruction 1 | FI | DI | CO | FO | EI | WO | | | | | | | | |
| Instruction 2 | | FI | DI | CO | FO | EI | WO | | | | | | | |
| Instruction 3 | | | FI | DI | CO | FO | EI | WO | | | | | | |
| Instruction 4 | | | | FI | DI | CO | FO | EI | WO | | | | | |
| Instruction 5 | | | | | FI | DI | CO | FO | EI | WO | | | | |
| Instruction 6 | | | | | | FI | DI | CO | FO | EI | WO | | | |
| Instruction 7 | | | | | | | FI | DI | CO | FO | EI | WO | | |
| Instruction 8 | | | | | | | | FI | DI | CO | FO | EI | WO | |
| Instruction 9 | | | | | | | | | FI | DI | CO | FO | EI | WO |

Figure 12.10 Timing Diagram for Instruction Pipeline Operation

Figure 40. Diagrama de tiempos con segmentación de 6 etapas

| | Time → | | | | | | | | | | | | Branch penalty | |
|----------------|--------|----|----|----|----|----|----|----|----|----|----|----|----------------|----|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| Instruction 1 | FI | DI | CO | FO | EI | WO | | | | | | | | |
| Instruction 2 | | FI | DI | CO | FO | EI | WO | | | | | | | |
| Instruction 3 | | | FI | DI | CO | FO | EI | WO | | | | | | |
| Instruction 4 | | | | FI | DI | CO | FO | | | | | | | |
| Instruction 5 | | | | | FI | DI | CO | | | | | | | |
| Instruction 6 | | | | | | FI | DI | | | | | | | |
| Instruction 7 | | | | | | | FI | | | | | | | |
| Instruction 15 | | | | | | | | FI | DI | CO | FO | EI | WO | |
| Instruction 16 | | | | | | | | FI | DI | CO | FO | EI | WO | |

Figure 12.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

Figure 41. Diagrama de tiempos. Salto incondicional

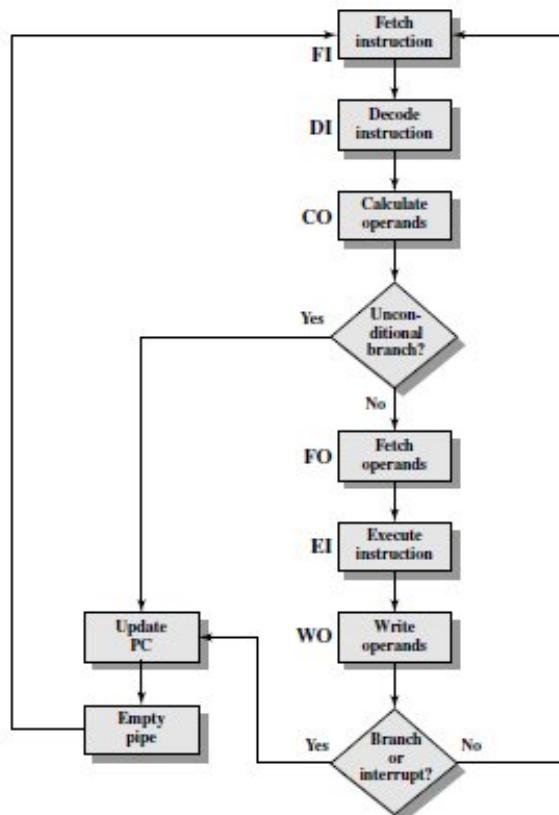


Figure 42. Flujo de instrucciones con segmentación de 6 etapas

7.10. Ejercicios

- Capítulo 12 del libro de texto William Stallings.
- Capítulo 13 del libro de texto William Stallings

7.11. Imágenes

- [Imagenes](#)

Chapter 8. Mecanismos de Entrada/Salida

8.1. Temario

8. Sistema de entrada / salida
 - a. Sincronización por encuesta
 - b. Sincronización por interrupción
 - c. Vector de interrupciones
 - d. Acceso directo a memoria DMA
 - e. Programación en lenguaje ensamblador de rutinas de entrada/salida

8.2. Bibliografía

- Libro de Texto:
 - Estructura y Organización de Computadores. William Stalling: Capítulo 7

8.3. Periféricos

8.3.1. Ejemplos

- Teclado
- Monitor
- Disco Duro
- Red
 - LAN
 - Wifi
- Periférico externo
 - Pen-Drive
- De cada ejemplo info de:
 - modelo y link a características
 - interfaz: bus eléctrico, protocolo de comunicaciones
 - ancho de banda

8.3.2. Modelo

- Media : Magnético (HD), Mecánico (Robot), Óptico (CD), Eléctrico (pen drive), etc
 - Electrónica analógica.
- Driver HW
 - Interfaz con el media:
 - las señales que actúan sobre el media son de distinto tipo: óptica (luz), mecánica (pneumático), acústica, etc. Estas señales se obtienen normalmente de la transformación de una señal eléctrica: interfaz eléctrico/óptico, eléctrico/mecánico, eléctrico/acústico
 - Ej: Un altavoz
 - Ej: Un Láser con el disco óptico.

- Ej: El modulador electrónico del láser
- Controlador del Periférico (**MCU: MicroController Unit**)
 - [Imagen de un controlador de disco](#)
 - El Controlador da órdenes al Driver HW
 - Es un secuenciador que interpreta **Comandos** (lenguaje específico para tareas del periférico) cuya ejecución realizará funciones propias del periférico.
 - Lenguaje de comandos. Command Set Architecture (CSA) ¿ISA?
 - [Lenguaje SCSI](#)
 - [Lenguaje ATA / ATAPI](#)
 - ATA Command Set (ACS): ejemplo de comandos IDENTIFY, READ DMA, WRITE DMA and FLUSH CACHE commands
 - comandos de transferencia de datos, de control (operaciones mecánicas como girar), de test (estado del periférico: conectado, desconectado)
 - Ej: en el caso de un disco el comando "girar a determinadas revoluciones". El disco integra un secuenciador propio, un MCU.
- Firmware
 - El set de comandos del periférico son interpretados por el software (firmware) cargado en la memoria del controlador MCU. Dicho software ha sido grabado por el fabricante del periférico. El usuario únicamente podrá escribir en el periférico algunos parámetros de configuración del periférico siendo accesible el Firmware únicamente por el fabricante.

8.4. Teclado

- Estructura



- Códigos



- Driver

8.5. Arquitectura Computadora

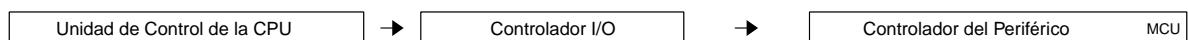
8.5.1. Von Neumann

- 3 unidades básicas
- Controlador i/o
 - Controlador auxiliar, no central, con dedicación específica a las operaciones E/S.
- Subsistema de Entrada/Salida : es uno de los 3 componentes del modelo Von Neumann
- Es necesario acceder a la máquina para:

- Introducir el programa : Desde un soporte de almacenamiento (papel, disco, etc) ha de cargarse el programa en la memoria
- Extraer el resultado generado por la máquina: Desde la memoria los resultados han de almacenarse en un soporte de almacenamiento (disco, impresora, etc), visualizarse (pantalla, etc), transferirse (red, etc).
- Dispositivos Periféricos
 - Son recursos hardware que complementan y extienden los servicios del tandem CPU-MEMORIA facilitando las tareas del programador y del usuario.
 - Gran diversidad: teclado, monitor, ratón, discos, tarjeta video, tarjeta red, ...
 - diferencia de complejidad entre un teclado y un disco duro
- La CPU normalmente es un recurso único compartido por todos los programas y por todos los periféricos.

8.5.2. Conexión CPU-E/S

- La arquitectura está formada, por lo tanto, por dos controladores: CPU y MCU. La CPU tiene un controlador generalista (CPU) mientras que el periférico tiene un controlador (MCU) muy específico. El lenguaje máquina de la CPU es generalista mientras que el lenguaje máquina del periférico es muy específico.
 - El periférico se comporta como una máquina **servidor** con su propio procesador. Podemos hablar de la máquina **host** (anfitrión) y de la máquina **server**. El controlador host es la CPU de la computadora y el controlador server es la MCU del periférico.
 - La CPU no se comunica directamente con el MCU sino que delega la tarea de los periféricos a procesadores no centrales, es decir, a los controladores I/O. La arquitectura típica de la computadora es la de un Procesador Central y un Set de controladores i/o que Intel denomina **Chipset**



- Ejemplo: Disco Duro
 - [Seagate Momentus 7200.4 500GB 7.2K 2.5-inch SATA Hard Drive ST9500420AS](#)
 - [video del movimiento del brazo](#)
 - [Disk buffer](#): memoria interfaz entre la transferencia del drive y la transferencia i/o del puerto.
 - [Controlador Atmel casero](#): Disco Sata con controladora Atmel e interfaz ethernet.

8.5.3. Controlador I/O

Introducción

- Periférico remoto:
 - Ej:PC---->SATA---->Compact Disc
- Controlador I/O del PC :
 - NO es la Unidad de Control de la CPU
 - Es uno de los 3 elementos básicos de la arquitectura Von Neumann
 - La CPU delega en otro controlador denominado controlador I/O la ejecución de las instrucciones máquina de entrada/salida.
 - Es necesario una INTERFAZ entre la CPU y el PERIFERICO
 - ISA: Instrucciones máquina de entrada salida de la cpu: de lectura (IN) y de escritura (OUT).

Comunicación en ambos sentidos.

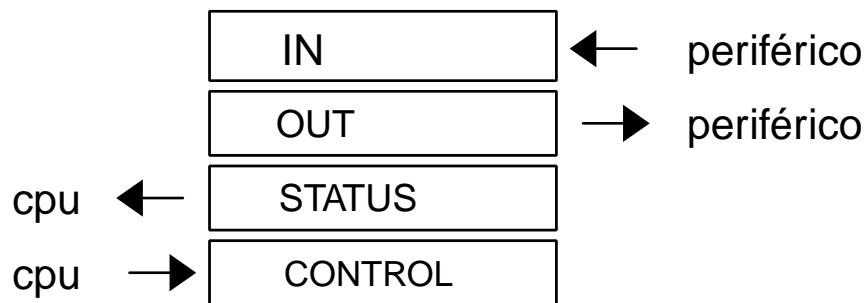
- Se transfieren tanto los DATOS como los COMANDOS del periférico.
- William denomina al controlador I/O con el nombre Módulo de E/S
- Es el controlador I/O el que transfiere los comandos y los datos al controlador del periférico.
 - El controlador del periférico interpretará los COMANDOS recibidos del controlador I/O y escribirá o/leerá los DATOS.
 - Ejemplo: The Advanced Host Controller Interface (AHCI) is a technical standard defined by Intel that specifies the operation of Serial ATA (SATA) io controller (host bus adapters).
- Estructura



Puertos

- Los puertos son registros de memoria implementados en el controlador i/o.
- Un puerto está formado por distintos tipos de registros: entrada de datos, salida de datos, estado del periférico, control del periférico

controlador i/o puerto



- El **controlador I/O** controla y ejecuta las comunicaciones a través de sus puertos.
 - Ej: Controlador I/O con puerto SATA
- **Puerto** de comunicaciones: Acceso al otro interlocutor (el periférico en este caso)
- Un controlador I/O puede tener varios puertos y controlar las comunicaciones con varios periféricos.
- Un puerto puede ser compartido por varios periféricos
- Linux
 - `cat /proc/ioports`

8.5.4. Espacio de direcciones

- Las direcciones i/o del puerto del controlador i/o se puede implementar de dos formas:
 - puertos mapeados en la memoria principal
 - direcciones de los puertos en un espacio diferente de la memoria principal: espacio i/o.

Memory-Mapped I/O (MMIO)

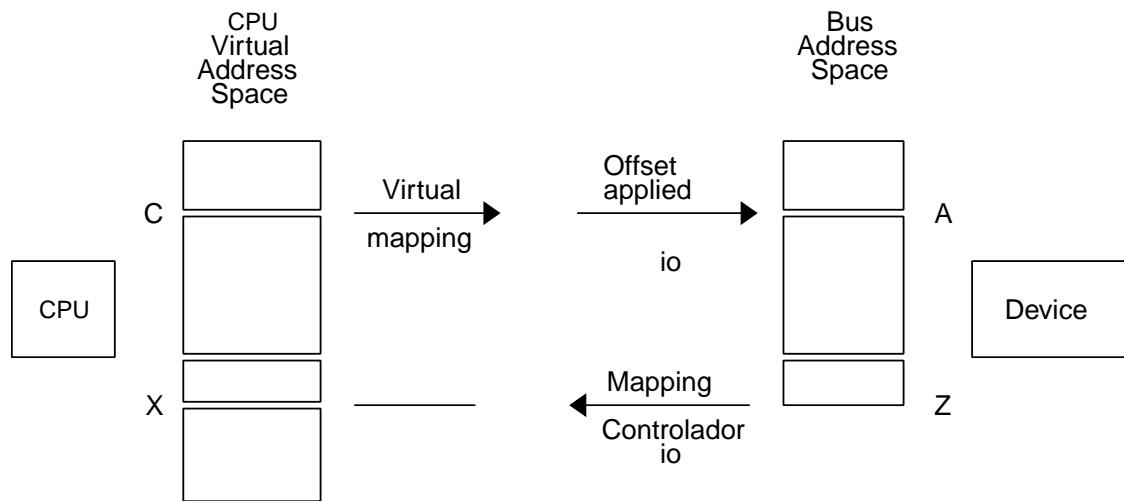
- Main Memory Address Space
- El bus del sistema shares memory address space between I/O devices and program memory
- Interface to the I/O is treated as a set of primary memory locations (Memoria principal)
- Software drivers determine meaning of data stored or retrieved.
- Loss of some memory space (8086 - 300K) because it is reserved for I/O interfaces. Less important with 4GiB address space.
- All instruction modes available → Todo el repertorio de instrucciones, no solo IN,OUT.
- May slow overall memory bus access down.
- Can limit or complicate contiguous memory range.
- Original x86 architecture had a 1MiB boundary because I/O was mapped above 640K.

Port mapped I/O (PMIO)

- I/O Address Space
- CPU has separate set of instructions that access specific pin on CPU that act as ports or that cause a demux to connect the address and data pin-outs to a different set of lines tied to i/o.
- Separate bus tied to I/O devices. CPU can go back to using memory bus while I/O devices responds.
- Adds complexity to CPU design.
- Often limited set of instructions. May need to write to memory before other actions can be taken.

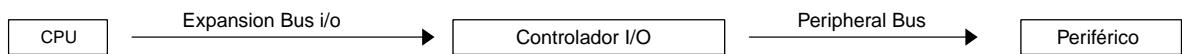
Direcciones de los periféricos

- Driver
 - El programa driver es un proceso i/o que utiliza el mecanismo de memoria virtual igual que el resto de procesos.
 - Mapa en el fichero `/proc/iomem`
- Periférico
- Al espacio de direcciones utilizado por los periféricos se le denomina **bus address**
- Hará falta mapear, traducir, direcciones bus a direcciones virtuales de la cpu
 - Mapa en el fichero `/proc/ioports`



8.5.5. Buses

- Tipos de buses
 - del procesador: bus interno a la cpu
 - de memoria: bus entre el controlador de memoria y la memoria principal
 - del sistema: bus externo a la cpu para el interconexión de dispositivos externos como la memoria principal y los controladores i/o de los periféricos.
 - local: bus i/o corto que permite elevados anchos de banda
 - de expansión: bus i/o largo que permite la conexión de múltiples *tarjetas*
 - periférico: bus i/o que permite conectar dispositivos externos a la computadora



- Bus i/o local; vlb, PCI, AGP
- Bus i/o de expansión: ISA, EISA
 - Conexión directa de la tarjeta i/o al bus de expansión de la placa base a través de *slots*:
- Bus periférico: SCSI, SATA, USB, RS232
 - Conexión externa a través de un *cableado*
- La arquitectura del bus i/o la componen
 - Interfaz (cable y conector)
 - Protocolo de comunicaciones: set of standardized rules for consistent interaction between system and i/o devices, including physical properties, access methods, data formats, etc. El Bus da nombre al protocolo.
 - Lenguaje de comandos
- Ejemplos prácticos
 - ISA
 - Industry Standard Architecture
 - PC/XT 8086 (1983) 8 bits

- 4 canales DMA
 - PC/AT i286 (1984) 16 bits
 - 16 MB/s
 - 7 canales DMA
 - 11 líneas IRQ
- EISA
 - Extended Industry Standard Architecture
 - PC Clon: i386-i486 (1988)
 - 32 Bits
 - Alternativa de los clónicos al propietario MCA de IBM en su PS/2
 - 33 MB/s de velocidad de transferencia para buses maestros y dispositivos DMA
 - 7 canales DMA
 - 15 líneas IRQ
- MCA
 - Micro Channel Architecture
 - IBM PS/2 (1987)
 - 32 bits
- PCI: Peripheral Component Interconnect
- PCI Express
- [listado de anchos de banda](#)
- <http://www.karbosguide.com/hardware/module2b2.htm>
 - El controlador i/o se conecta indirectamente al bus del sistema (CPU-MP) a través de los puentes (bridges)
- [Intel](#)
 - Intel ha evolucionado de los puentes ICH con el puente Sur y Norte a un Concentrador Central PCH
 - Observar que la CPU integra el controlador de memoria integrado (IMC) y controladores i/o de video (PCI-E Graphics)

8.5.6. Análisis: Portátil Lenovo - Disco Duro

- Ruta de la transferencia de datos entre el disco duro y la memoria principal en la computadora Lenovo T520
- Disco (ATA disk, ST9500420AS Seagate) → Driver Mecánico/Electrónico/Magnético → Micro del Disco (SATA Interface, Seagate) → Bus i/o serie (SATA 6Gb/s) → Host Adapter (Platform_Controller_Hub PCH, ChipSet 200C/6 Series, SATA AHCI Controller) → Flexible Display Interface (FDI) → CPU (Intel Core i5)
- SATA: Serial Advanced Technology Attachment is a computer bus interface that connects host bus adapters (controladora de disco) to mass storage devices (MCU, MicroControllerUnit) such as hard disk

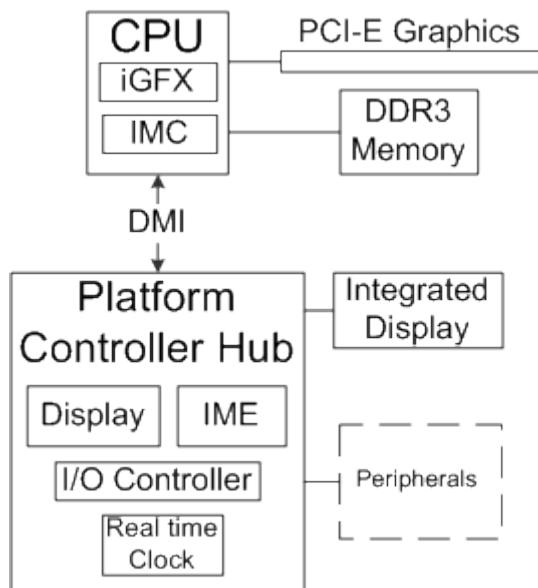


Figure 43. Lenovo T520

8.6. Programa E/S

- Programmed i/o (PIO)
 - Las transferencias de datos mediante mecanismos de E/S por consulta la realiza un programa i/o (PIO) que ejecuta la CPU. La CPU en cada transferencia de datos entre la memoria y el periférico debe de ESPERAR a que dicha transferencia termine.

8.6.1. Módulo fuente

- Transferencia de 512 bytes entre el puerto 0x380 y un buffer.

```

mov %bx,buf ; destination address. BX es un puntero a un buffer
mov $512,%si ; count. Número de bytes a transferir
mov $0x380,%dx ; source port. DX es un puntero al puerto
loop:
    in %dx,%al ; get byte from i/o port. AL<-DX
    mov %al,(%bx) ; store in buffer      M[bx]<-AL
    inc %bx ; next memory location in buf
    dec %si ; decrement bytes left
    jnz loop
    
```

ISA

- IN: leer un dato del puerto
- OUT: escribir un dato en el puerto

8.7. Driver: Sistema Operativo

8.7.1. Gestor E/S: jerarquía

- La gestión de las operaciones E/S las realiza el Sistema Operativo
- La estructura del programa gestor E/S del sistema operativo se basa en una estructura jerárquica por

niveles:

- Nivel más bajo: controlador sw (módulo driver) del controlador hw i/o del periférico.
- Nivel más alto: Sistema virtual de ficheros. Las aplicaciones acceden a los periféricos mediante la abstracción de estos en ficheros virtuales.

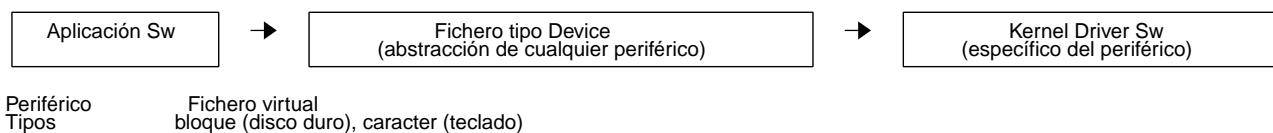
8.7.2. Código Fuente

- Pseudo-código

```
While (STATUS == BUSY)
    ; // wait until device is not busy . Puerto ocupado.
Write data to DATA register // dato a transmitir el puerto out
Write command to COMMAND register // registro de control
    // Doing so starts the device and executes the command
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

8.7.3. Concepto

- El programa que implementa las funciones del periférico es un *módulo del kernel* denominado DRIVER
 - Driver del teclado, monitor, disco duro,
- Capas SW:



- Ejemplo: Escritura de un fichero en el disco duro
 - write → syscall → OUT → comando propio del HD



- Espacio de usuario: write (función de escritura de datos), syscall (llamada al módulo gestor de E/S del Sistema Operativo)
- Espacio kernel del S.O.: Driver: Orden interpretada por la CPU y ejecutada por el controlador I/O para transferir datos (comandos y datos) entre Memoria y el Controlador Periférico
- Espacio periférico: Comandos a Interpretar por el Periférico (Firmware) y transferencia de Datos.

8.7.4. Utilización del Driver

- El driver está protegido por el Sistema Operativo. Hay funciones como ioctl que permite al usuario interactuar con el driver.
 - La interfaz entre el usuario y el driver son las llamadas al sistema operativo.
 - Mediante la instrucción máquina SYSCALL (x86-64) o int 0x80 (x86-32) llamamos indirectamente a las funciones del driver a través del sistema operativo.

- Ejemplo
 - Imprimir en la pantalla: open, write, close → open y close interactúan con el sistema de ficheros virtual.

8.8. Mecanismos de Implementación de la Interfaz E/S

8.8.1. Introducción

- All data manipulation not directly performed in the CPU or between CPU and primary memory is I/O.
- PIO: Polling
- Interruption
- DMA: Direct Memory Access

8.8.2. Sincronización por Encuesta

- Polling: encuesta
- Query: encuesta
- Mecanismo
 - Comprobación del estado o *encuesta-polling*
 - La CPU consulta el registro de estado de cada puerto al que están conectados los periféricos. Comprueba si algún periférico requiere el servicio de la CPU. Reserves a register for each I/O device. Each register is continually polled to detect data arrival.
 - Es necesario ejecutar programas de atención al periférico cuando este lo requiera: sincronización
 - El anfitrión consulta el bit de estado del controlador i/o
 - Identificación
 - Una vez aceptada la petición del cliente (controlador i/o)
 - El controlador identifica el periférico que solicita el servicio
 - Comunica al S.O. qué periférico
 - Estructura
 - CPU:
 - Ejecuta el programa i/o: un programa que controla DIRECTAMENTE la operación E/S: Programmed I/O → PIO
 - realiza las transferencias entre la memoria principal y el controlador i/o
 - espera al periférico hasta que termine. La CPU espera hasta que concluya la operación E/S.
 - Memoria principal: almacena el programa i/o
 - Controlador i/o
 - Puerto: Un puerto está compuesto por REGISTROS del tipo datos, control, test
 - transfiere los datos al periférico

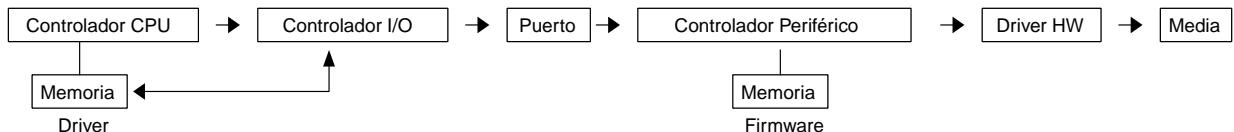
8.8.3. Sincronización por Interrupción

- Interrupt-Driven I/O (Mecanismo de E/S por Interrupción)
- Estructura
 - CPU:

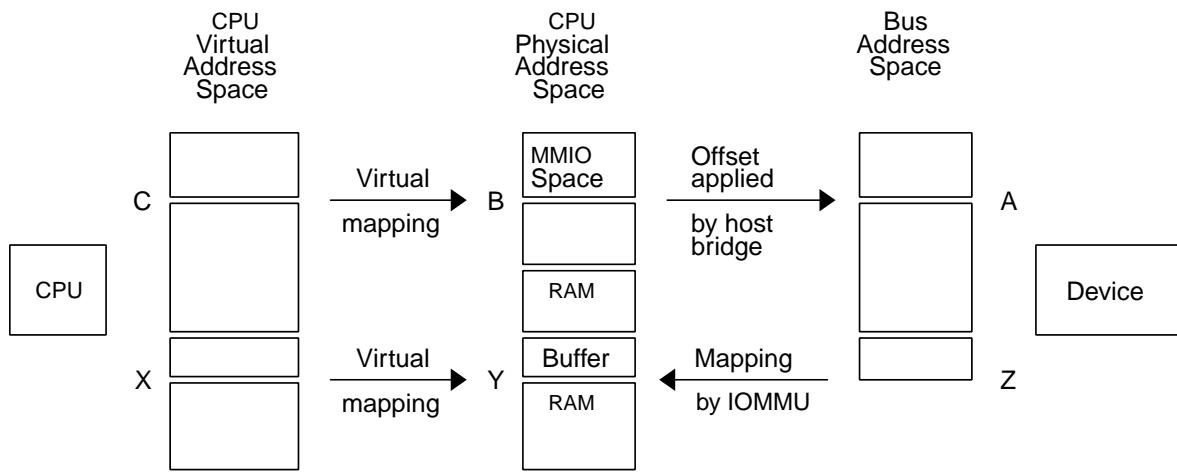
- Ejecuta el programa de atención a la interrupción. Un programa que controla DIRECTAMENTE la operación E/S.
- realiza las transferencias entre la memoria principal y el controlador i/o
- **no espera** al periférico hasta que termine. Es *interrumpido* cada vez que es necesario realizar una transferencia
 - Al finalizar el ciclo de instrucción de cada instrucción que ejecuta la CPU, se comprueba si la señal de petición de interrupción está activada.
- Memoria principal: almacena el programa i/o
- Controlador i/o
 - Puertos: datos, control, test
 - transfiere los datos al periférico
- allows the CPU to do other things until I/O is requested
 - Interrupt request - Driven I/O (Still PIO - CPU has to move data)
 - I/O devices can request the attention of CPU with an interrupt at any time, but only when needed.
 - CPU can dedicate extended time for particular device.
 - CPU does not have to check in on I/O that does not need attention.
 - CPU can delay processing of I/O request.
 - Newer systems - cpu hands off transfer of data to secondary controller, which only interrupts cpu on completion of task or problem.
 - Requires external circuitry.
 - e.g 8259A programmable interrupt controller (PIC). CPU may have to communicate with the PIC to identify requesting device.
- Programmed input/output (PIO) is a method of transferring data between the CPU and a peripheral such as a network adapter or an ATA storage device. In general, programmed I/O happens when software running on the CPU uses i/o instructions that access I/O address space to perform data transfers to or from an I/O device. This is in contrast to Direct Memory Access (DMA) transfers.
- The best known example of a PC device that uses programmed I/O is the ATA interface;

8.8.4. Direct Memory Access (DMA)

- Estructura



- Mapeo de direcciones
- Hará falta una unidad hardware de traducción de direcciones bus a direcciones físicas : iommu



- Estructura

- CPU:

- Ejecuta el programa i/o. El programa no controla la transferencia pero sí la inicializa (número de bytes a transferir, localización en la memoria principal, localización en el periférico, control errores, etc)
 - Cede el control de las transferencias al controlador DMA (DMAC), offloads I/O processing to a special-purpose chip that takes care of the details. La transferencia la controla y realiza el DMAC por Hardware → No es por programa como el PIO.

- Memoria principal: almacena el programa i/o

- Controlador i/o

- es el controlador DMA
 - Puertos: los puertos ahora no son para los datos de transferencia, únicamente para el control CPU-DMA
 - transfiere los datos entre la memoria principal y el periférico
 - el controlador no espera al periférico
 - Direct Memory Access controller.

Handles I/O interaction without the intervention of the CPU after initial CPU interaction. Uses interrupts to report status back to CPU. Requires separate arbitration protocol - shares buses with CPU. Predefined standardized tasks. CPU NOT occupied but may have to compete for resources.

8.8.5. Channel I/O

- uses dedicated I/O processors
 - Channel I/O (Mainframe or Supercomputer)
 - Estructura: integra la unidad DMA más un procesador específico.
 - Programable: ejecuta el *channel program* almacenado en la memoria principal. (Diferencia con DMA).

- Transfiere datos (Memoria principal <→ Periférico) independientemente de la CPU

Memory Shared

- Estructura
 - CPU
 - cede el control de las transferencias al procesador o canal i/o
 - Memoria principal: almacena el programa i/o
 - Procesador i/o
 - Es el canal i/o
 - Ejecuta el programa i/o almacenado en la memoria principal
 - Puertos: los puertos ahora no son para los datos de transferencia, únicamente para la control CPU-DMA
 - transfiere los datos entre la memoria principal y el periférico
 - Memoria Principal
 - Compartida entre la CPU y el Canal_IO

Memory Independent

- Estructura
 - CPU
 - cede el control de las transferencias al procesador o canal i/o
 - Memoria principal: almacena el programa i/o
 - Procesador i/o
 - Es el canal i/o
 - Ejecuta el programa i/o almacenado en la memoria principal
 - Puertos: los puertos ahora no son para los datos de transferencia, únicamente para el control CPU-DMA
 - transfiere los datos entre la memoria principal y el periférico
 - Memoria Principal
 - Accesible sólo por la CPU
 - Memoria IO
 - Accesible sólo por el canal_IO

8.9. Sincronizacion por Interrupcion

- Extensión del apartado anterior sobre implementación de la interfaz i/o driven-interruption.

8.9.1. Concepto

- El inconveniente del Polling es que la CPU realiza la consulta aunque el periférico no requiera sus servicios.
- El periférico toma la iniciativa y solicita la INTERRUPCION del programa que este ejecutando para ejecutar el programa requerido por el periférico

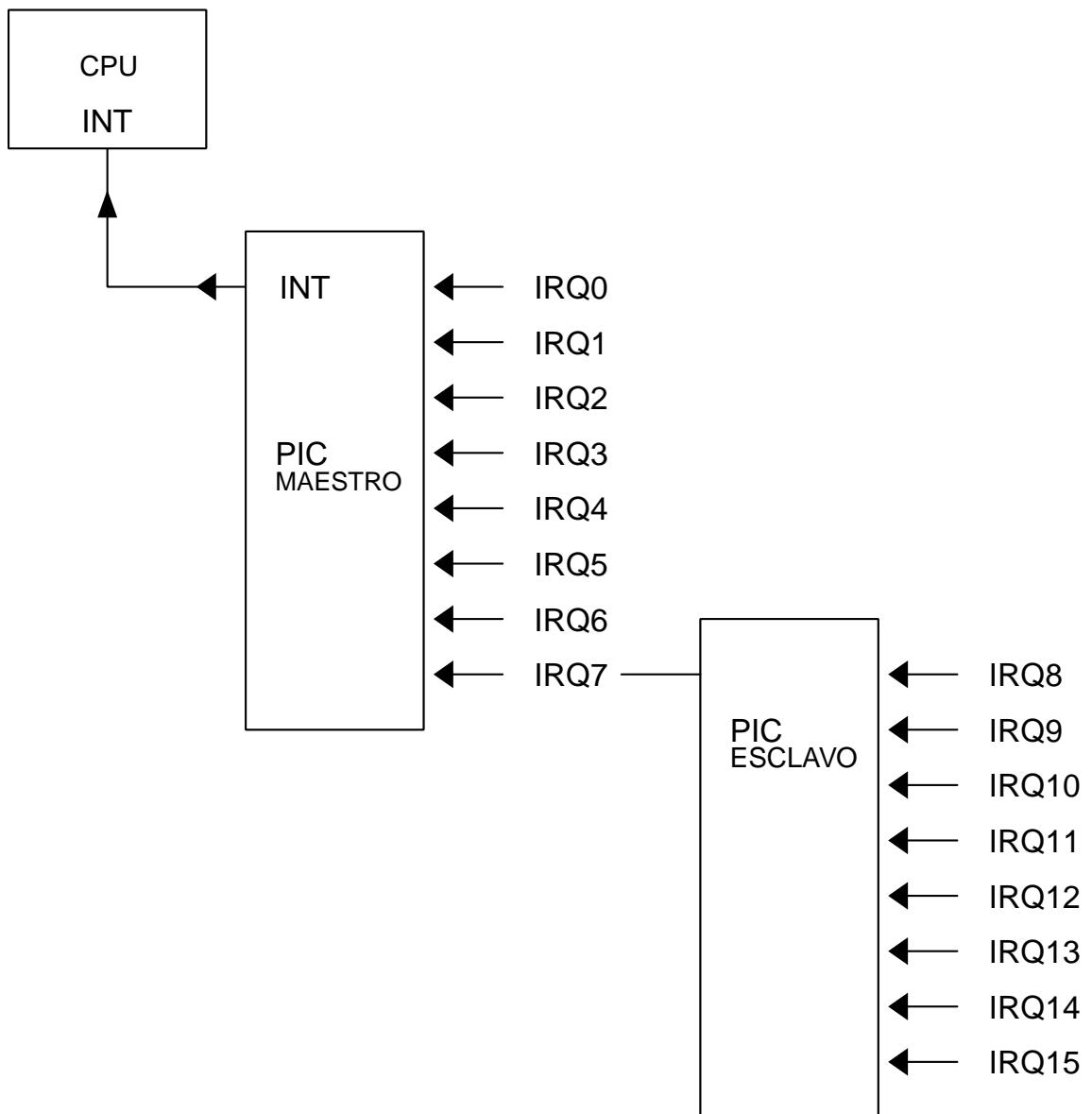
8.9.2. Mecanismo de Interrupcion

- El periférico a través de una línea eléctrica de entrada de la CPU, solicita al controlador i/o los servicios del kernel
 - El kernel va a ser **INTERRUMPIDO**
- La CPU tiene dos líneas de interrupción:
 - Línea Interrupt ReQuest (*IRQ*) : Maskable
 - Línea Non Maskable Interrupt (*NMI*)
 - La CPU en el ciclo de instrucción tiene en su última fase la fase de Chequeo de Interrupción
 - Si se está solicitando un servicio al kernel, la CPU entra en modo atención a la interrupción y pasa el control al módulo de **Gestión de Interrupciones** del Kernel
- Las líneas de los periféricos para solicitar la interrupción se denominan (*IRQ*).

8.9.3. Controlador de Interrupciones

PIC

- Controlador de Interrupciones Programable
 - **PIC** : Programmable Interrupt Controller
 - Se utiliza en arquitecturas cuya CPU tiene un único núcleo.
 - Tiene como entrada todas las líneas de interrupción de los periféricos
 - Salidas: INT (petición de interrupción) y D0-D7 (Control, Status and Interrupt-Vector)
- Ejemplo : [PIC 8259](#)
- Conexión daisy-chain
 - Maestro - Esclavo



- Acciones que realiza el controlador PIC
 - Chequea si se activa alguna señal (Monitorización). En caso de activarse más de una se dar prioridad a la de menor nivel y procede:
 - **Convierte** la línea activada IRQn en un vector (0x00-0xFF)
 - Escribe en el puerto i/o del PIC el vector. El puerto es accesible por parte de la CPU. El vector se apunta a la entrada de una tabla que contiene un puntero a la rutina de atención a la interrupción (ISR)
 - Activa la señal INTR de la CPU
 - Si la CPU lee el valor del vector se desactiva la señal INTR
- A la línea IRQn le corresponde, por defecto según Intel, el vector $n+32$. Este mapeo se puede alterar programando el PIC.
 - A la línea IRQ0 le corresponde el vector 0x20
- Cada línea IRQ se puede desautorizar por programa pero dicha interrupción no se pierde.
- Mediante la instrucción **cli** se hace un clear del flag IF del registro EFLAGS, ignorando la CPU todas las

interrupciones hardware.

- Mediante la instrucción `sli` hacemos un set de IF.

NMI

- Es una entrada de la cpu.
- The NMI (is a hardware driven interrupt much like the PIC interrupts but the NMI goes directly to the cpu, and not via the PIC controller).
- Aplicaciones
 - Temporizador watchdog
 - Es un temporizador que hay que poner a cero regularmente. Si la cpu está bloqueada no podrá resetear el temporizador y este generará un interrupción nmi con lo que el contador de programa se cargará con una dirección que apunta a la rutina de atención a la interrupción NMI cuya ejecución desbloqueará el estado de la cpu.

Intel

- Intel
 - 1º generation of interrupts (XT-PIC): only supported 15 interrupts.
 - 2º generation (IO-APIC): number of supported interrupts to 24.
 - APIC : Advanced Programmable Interrupt Controller: Utilizado en modernas arquitecturas multinúcleo.
 - 3º generation, MSI: number of available interrupts to 224.

8.9.4. Gestor de Interrupciones

- El Gestor de Interrupciones está implementado por el S.O.: `entry.S` en la versión linux 2.x
- Identifica al solicitador de la interrupción para poder ejecutar la rutina específica de atención a dicha interrupción.
- Anula la posibilidad de ser interrumpido por otros dispositivos no prioritarios, a través del flag IF , bit 9 del registro de control `rflags` de la CPU.
- Consulta el Vector de interrupciones (Tabla de punteros a las rutinas de atención a la interrupción).
- Salva el entorno del programa en ejecución que va a ser interrumpido.
- Activa la rutina Interrupt Service Routine (*ISR*).
 - Autoriza nuevamente las interrupciones
 - Dicha rutina estará implementada en el módulo driver del kernel.

8.9.5. Tipos de Interrupciones

- Intel define dos tipos de Interrupt Signals
 - **Síncronas**
 - Son originadas por la propia CPU al final del ciclo de instrucción en la fase de interrupción
 - Se denominan *exceptions*
 - *Interrupt Software* : originadas por la instrucción `syscall`: Llamadas al Sistema
 - Originadas por un *Error*: Fault, Trap, Abort.
 - **Asíncronas**

- Originadas por los periféricos o hardware que no es la CPU
- Se denominan *interruptions* o *hardware interruptions*
 - Maskable: IRQ
 - Non-Maskable: NMI

8.9.6. Tabla de los Vectores de interrupciones

Modo Real: Tabla IVT

- En plataformas con S.O. al encender la computadora (arranque con bootloader) la CPU está operando inicialmente en *Modo Real* y en plataformas sin S.O. (arranque con BIOS) la cpu opera permanentemente en Modo Real. En plataformas con S.O. el arranque se inicia en modo real y se configura la computadora para pasar al modo protegido antes de cargar el S.O. en la memoria principal.
 - Real Mode :
 - Is a simplistic 16-bit mode that is present on all x86 processors: Equivale a comportarse como la antigua cpu 8086.
 - La cpu 8086 tiene 20 bits de direcciones y 16 bits de datos.
 - A real mode pointer is defined as a 16-bit segment address and a 16-bit offset into that segment
 - El segmento se expande a 20 bits multiplicando x4.
 - 2^{20} : El Código tiene que estar en el primer Mega de la memoria RAM
- Permite el acceso a funciones de la BIOS.
 - [Tabla de interrupciones BIOS](#)
 - [Tecnología del PC](#)

```
MOV AH, 0Eh    ; Imprime carácter en la pantalla
MOV AL, '!'    ; carácter a imprimir
INT 10h        ; Llamada a las funciones de video del BIOS
```

- Tabla IVT.

| Interrupt Address | Type | Description |
|-------------------|------------|---|
| 00h | 0000:0000h | Processor Divide by zero |
| 01h | 0000:0004h | Processor Single step |
| 02h | 0000:0008h | Processor Non maskable interrupt (NMI) |
| 03h | 0000:000Ch | Processor Breakpoint |
| 04h | 0000:0010h | Processor Arithmetic overflow |
| 05h | 0000:0014h | Software Print screen |
| 06h | 0000:0018h | Processor Invalid op code |
| 07h | 0000:001Ch | Processor Coprocessor not available |
| 08h | 0000:0020h | Hardware System timer service routine |
| 09h | 0000:0024h | Hardware Keyboard device service routine |
| 0Ah | 0000:0028h | Hardware Cascade from 2nd programmable interrupt controller |
| 0Bh | 0000:002Ch | Hardware Serial port service - COM post 2 |
| 0Ch | 0000:0030h | Hardware Serial port service - COM port 1 |
| 0Dh | 0000:0034h | Hardware Parallel printer service - LPT 2 |
| 0Eh | 0000:0038h | Hardware Floppy disk service |

| | | | |
|---------|-------------------------|----------|---|
| 0Fh | 0000:003Ch | Hardware | Parallel printer service - LPT 1 |
| 10h | 0000:0040h | Software | Video service routine |
| 11h | 0000:0044h | Software | Equipment list service routine |
| 12h | 0000:0048H | Software | Memory size service routine |
| 13h | 0000:004Ch | Software | Hard disk drive service |
| 14h | 0000:0050h | Software | Serial communications service routines |
| 15h | 0000:0054h | Software | System services support routines |
| 16h | 0000:0058h | Software | Keyboard support service routines |
| 17h | 0000:005Ch | Software | Parallel printer support services |
| 18h | 0000:0060h | Software | Load and run ROM BASIC |
| 19h | 0000:0064h | Software | DOS loading routine |
| 1Ah | 0000:0068h | Software | Real time clock service routines |
| 1Bh | 0000:006Ch | Software | CRTL - BREAK service routines |
| 1Ch | 0000:0070h | Software | User timer service routine |
| 1Dh | 00000074h | Software | Video control parameter table |
| 1Eh | 0000:0078h | Software | Floppy disk parameter routine |
| 1Fh | 0000:007Ch | Software | Video graphics character routine |
| 20h-3Fh | 0000:0080f-0000:00FCCh | SW | DOS interrupt points |
| 40h | 0000:0100h | Software | Floppy disk revector routine |
| 41h | 0000:0104h | Software | hard disk drive C: parameter table |
| 42h | 0000:0108h | Software | EGA default video driver |
| 43h | 0000:010Ch | Software | Video graphics characters |
| 44h | 0000:0110h | Software | Novel Netware API |
| 45h | 0000:0114h | Software | Not used |
| 46h | 0000:0118h | Software | Hard disk drive D: parameter table |
| 47h | 0000:011Ch | - | Software Not used |
| 48h | | Software | Not used |
| 49h | 0000:0124h | Software | Not used |
| 4Ah | 0000:0128h | Software | User alarm |
| 4Bh-63h | 0000:012Ch | - | Software Not used |
| 64h | | Software | Novel Netware IPX |
| 65h-66h | | Software | Not used |
| 67h | | Software | EMS support routines |
| 68h-6Fh | 0000:01BCh | Software | Not used |
| 70h | 0000:01C0h | Hardware | Real time clock |
| 71h | 0000:01C4h | Hardware | Redirect interrupt cascade |
| 72h-74h | 0000:01C8h - 0000:01D0h | Hardware | Reserved - Do not use |
| 75h | 0000:01D4h | Hardware | Math coprocessor exception |
| 76h | 0000:01D8h | Hardware | Hard disk support |
| 77h | 0000:01DCCh | Hardware | Suspend request |
| 78h-79h | 0000:01E0h | - | Hardware Not used |
| 7Ah | | Software | Novell Netware API |
| 78h-FFh | 0000:03FCh | Software | Not used |

- El contenido de la tabla depende de la generación de la cpu de intel
- Primera columna: Número del vector de interrupción. Número de la entrada a la tabla de vectores.
- Segunda columna: el offset en la tabla del número de vecto de interrupción
- Columna X: Falta en la tabla.
 - El vector de 4 bytes: **Es un puntero a la rutina de atención a la interrupción ISR**

- La dirección es segmentada. Segmento:Offset. Dos bytes para el segmento y otros dos para el offset
- Direccionamiento:
 - El Registro IDTR apunta a la primera entrada de la tabla.
 - The IVT table is typically located at 0000:0000H, and is 400H bytes in size (**4 bytes for each interrupt of 265 interruptions**).
 - Observamos que podemos obtener la dirección relativa multiplicando el número de interrupción x4.
 - Al vector 9 le corresponde el offset IVT 36, es decir, 0x24 → en forma segmentada 0000:0024h
 - El offset de la última entrada será = $4 \times 0xFF = 0x400-4 = 0x3FC$
- Tipos de interrupciones
 - The first 32 vectors are reserved for the processor's internal *exceptions* (0x00-0x1F)
 - Las interrupciones 0x20-0xFF son interrupciones *hardware IRQ*.
 - PIC
 - El controlador PIC es el encargado de mapear la señal IRQ a un vector de entrada a la tabla.
 - Periférico IRQ0 → PIC vector 0x20 → Tabla IVT puntero 0000:0080f (RAM) → llamada a la función ISR de atención al periférico IRQ0 (RAM)

Modo Protegido: Tabla IDT

- En las plataformas con S.O. una vez finalizadas las operaciones en modo real el bootloader finaliza la carga del sistema operativo y la cpu se configura en modo protegido no pudiendo el usuario: ejecutar módulos del S.O como los drivers, acceder a cualquier región de la memoria física, registros privilegiados, instrucciones privilegiadas,...
- El S.O. configura la tabla de descripción de interrupciones IDT con la misma función que la IVT pero distinto contenido.
- [Interrupt Descriptor Table IDT](#)

| IDT Offset | INT # | Description |
|------------|-----------|-----------------------------------|
| 0x0000 | 0x00 | Divide by 0 |
| 0x0004 | 0x01 | Reserved |
| 0x0008 | 0x02 | NMI Interrupt |
| 0x000C | 0x03 | Breakpoint (INT3) |
| 0x0010 | 0x04 | Overflow (INTO) |
| 0x0014 | 0x05 | Bounds range exceeded (BOUND) |
| 0x0018 | 0x06 | Invalid opcode (UD2) |
| 0x001C | 0x07 | Device not available (WAIT/FWAIT) |
| 0x0020 | 0x08 | Double fault |
| 0x0024 | 0x09 | Coprocessor segment overrun |
| 0x0028 | 0x0A | Invalid TSS |
| 0x002C | 0x0B | Segment not present |
| 0x0030 | 0x0C | Stack segment fault |
| 0x0034 | 0x0D | General protection fault |
| 0x0038 | 0x0E | Page fault |
| 0x003C | 0x0F | Reserved |
| 0x0040 | 0x10 | x87 FPU error |
| 0x0044 | 0x11 | Alignment check |
| 0x0048 | 0x12 | Machine check |
| 0x004C | 0x13 | SIMD Floating Point Exception |
| 0x00xx | 0x14 0x1F | Reserved |
| 0xxx | 0x20 0xFF | User definable → IRQ |

- El contenido depende del kernel del S.O.
- Primera columna: offset a la rutina de atención a interrupción ISR
- Segunda columna: número del vector de interrupción.
- tipos de interrupción
 - 0-0x1F: exceptions *ERROR* y NMI
 - 0x20-0x2F: INT maskable: IRQ0-----IRQ15
 - 0x30-0xFF: exceptions *SW*
 - 0x80
 - isa x86-64: *syscall*
 - isa x86: *int 0x80*
- ¿A que rutina apunta el vector 0xE? → Page Fault
- Descripción de las Entradas
 - IDTR: registro que apunta a la primera entrada de la tabla
 - Cada entrada son 8 bytes que intel llama gates.
 - Contiene un selector de segmento que identifica un descriptor de segmento de la tabla de descriptores de segmentos (ver segmentación intel)

IRQ

- XT-PIC interrupts use a pair of Intel 8259 programmable interrupt controllers (PIC)
 - PIC configurado por el kernel **Linux** : [Understanding the Linux Kernel, Second Edition by Marco Cesati, Daniel P. Bovet](#)
 - Example XT-PIC IRQ Assignment , [intel interrupts paper](#): Esta configuración es un ejemplo, es decir, el SO puede **reprogramarla** y variar su configuración.

| IRQ | Interrupt Hardware Device (vector de la tabla) |
|-----|--|
| 0 | 32 Timer |
| 1 | 33 Keyboard |
| 2 | 34 PIC Cascade |
| 3 | 35 Second Serial Port (COM2) |
| 4 | 36 First Serial Port (COM 1) |
| 5 | 37 <Free> |
| 6 | 38 Floppy Disk |
| 7 | 39 <Free> |
| 8 | 40 System Clock |
| 9 | 41 <Free> |
| 10 | 42 Network Interface Card(NIC) |
| 11 | 43 USB Port, and Sound Card |
| 12 | 44 Mouse (PS2) |
| 13 | 45 Math Co-Processor |
| 14 | 46 IDE Channel 1 |
| 15 | 47 IDE Channel 2 |

Note: Linux* requires IRQ 0, 2, and 13 to be as shown.

- Master 8259 (PC compatible)

| IVT Offset | INT # | IRQ # | Description |
|------------|-------|-------|------------------------|
| 0x0020 | 0x08 | 0 | PIT |
| 0x0024 | 0x09 | 1 | Keyboard |
| 0x0028 | 0x0A | 2 | 8259A slave controller |
| 0x002C | 0x0B | 3 | COM2 / COM4 |
| 0x0030 | 0x0C | 4 | COM1 / COM3 |
| 0x0034 | 0x0D | 5 | LPT2 |
| 0x0038 | 0x0E | 6 | Floppy controller |
| 0x003C | 0x0F | 7 | LPT1 |

- Segunda columna: número de interrupción en el PIC
- Tercera columna: número de interrupción IRQ
- Primera columna: offset de esa entrada respecto de la primera entrada. Número de Vector.

- Slave 8259

| IVT Offset | INT # | IRQ # | Description |
|------------|-------|-------|------------------------|
| 0x01C0 | 0x70 | 8 | RTC |
| 0x01C4 | 0x71 | 9 | Unassigned |
| 0x01C8 | 0x72 | 10 | Unassigned |
| 0x01CC | 0x73 | 11 | Unassigned |
| 0x01D0 | 0x74 | 12 | Mouse controller |
| 0x01D4 | 0x75 | 13 | Math coprocessor |
| 0x01D8 | 0x76 | 14 | Hard disk controller 1 |
| 0x01DC | 0x77 | 15 | Hard disk controller 2 |

- Segunda columna: número de interrupción en el PIC

- Tercera columna: número de interrupción IRQ
- Primera columna: offset de esa entrada respecto de la primera entrada. Número de Vector.

Linux

- Interrupciones configuradas por el kernel : `cat /proc/interrupts`

Lineas compartidas

- https://nptel.ac.in/courses/Webcourse-contents/IIT-%20Guwahati/comp_org_arc/web/module06_io/lect_03_intr/lect_03.htm
- Si una línea de interrupción está compartida por varios dispositivos, cuando uno de los dispositivos envía la señal de interrupción por la línea común, la CPU puede identificar el dispositivo que solicita la interrupción de varias maneras:
 - sondeo por software a cada miembro de la línea
 - la línea de concesión de la CPU hacia los dispositivos colocados en modo daisy-chain (se pasan la concesión entre ellos, de uno en uno), cuando la concesión llegue al miembro que solicita, este devuelve su identificación.

8.10. Acceso Directo a Memoria DMA

8.10.1. Funcionalidad

- Realizar las transferencias de datos liberando así a la CPU
- Aplicación: Transferencias de datos entre el disco duro y la memoria principal
- Unidad: DMAC (DMA Controller)
 - Puede tener varios canales DMA: cada canal se ocupa de la transferencia de un periférico.

8.10.2. Transferencias

- Modo ráfaga
 - Una vez que el DMAC toma el control del bus del sistema no lo cede hasta que la transferencia de todo el bloque es completada
 - Mientras el bus del sistema está ocupado por el DMAC la CPU puede operar con la memoria caché.
- Modo robo de ciclo
 - El DMAC devuelve el control del bus del sistema a la CPU cada vez que transfiere una palabra.
 - El bus es compartido en el tiempo: útil en sistemas críticos en tiempo real
- Modo transparente
 - El DMAC únicamente se adueña del bus cuando está libre y no lo necesita la CPU.

8.10.3. Sincronización

- La CPU puede iniciar una operación DMA en los límites del ciclo de bus de lectura o escritura. Por lo tanto se puede iniciar una operación DMA durante el ciclo de instrucción .

8.10.4. Operación del controlador DMA

Secuencia de pasos a nivel alto

- Cuando un proceso realiza una llamada *read*, el driver le asigna una región de memoria principal (DMA buffer) y genera la señales hw para solicitar la transferencia de datos al DMA buffer. El proceso queda en estado *sleep*.
- El DMAC transfiere los datos al buffer DMA y activa una señal de interrupción cuando finaliza
- El gestor de interrupciones ubica los datos del buffer al lugar definitivo, avisa de interrupción atendida y despierta al proceso, el cual ya puede leer los datos de la memoria principal.

Secuencia de pasos a nivel bajo

- Tres parámetros a programar:
 - dirección inicial de MPrincipal del bloque de datos a transferir: AR
 - Número de datos a transferir: WC
 - Modo de transferencia
- Pasos
 - a. La *CPU* durante el arranque de la computadora inicializa el DMAC programando los parámetros.
 - b. El *controlador del periférico* solicita su servicios.
 - c. El *periférico* realiza una petición de DMA al DMAC (DMA Controller): *DMA Request*.
 - d. El DMAC le responde con una señal de aceptación
 - e. El DMAC activa la línea de petición de DMA a la *CPU*: *Bus Request*
 - f. Al final del *ciclo del bus* en curso, el procesador pone las líneas del bus del sistema en alta impedancia y activa la sesión de DMA: *Bus Grant*
 - g. El DMAC asume el *control del bus del sistema*
 - h. El dispositivo de E/S transmite una nueva palabra de datos al registro intermedio de datos del DMAC (un pequeño *buffer* en el DMAC)
 - i. El DMAC ejecuta un ciclo de escritura en memoria para transferir el contenido del registro intermedio a la posición M[AR].
 - j. El DMAC decrementa WC e incrementa AR.
 - k. El DMAC libera el bus y desactiva la línea de petición de DMA.
 - l. El DMAC compara WC con 0:
 - m. Si WC > 0, se repite desde el paso 2.
 - n. Si WC = 0, el DMAC se detiene y envía una petición de interrupción al procesador.

8.10.5. Problemas de coherencia en la memoria cache

- El controlador DMA al transferir datos entre el periférico y la memoria Principal provoca que las líneas de la memoria caché no sean copia de los bloques de la memoria principal. Será necesario que la controladora de la caché actualice la memoria caché después de una operación DMA.

8.11. Buses

- La arquitectura i/o ha ido evolucionando en dos direcciones
 - incremento del ancho de banda de los buses
 - integración de los controladores i/o en un único chip

8.11.1. ISA

IBM PC/XT Architecture ('82, '83) (*XT is “extended” PC – 4.77 MHz Bus*)

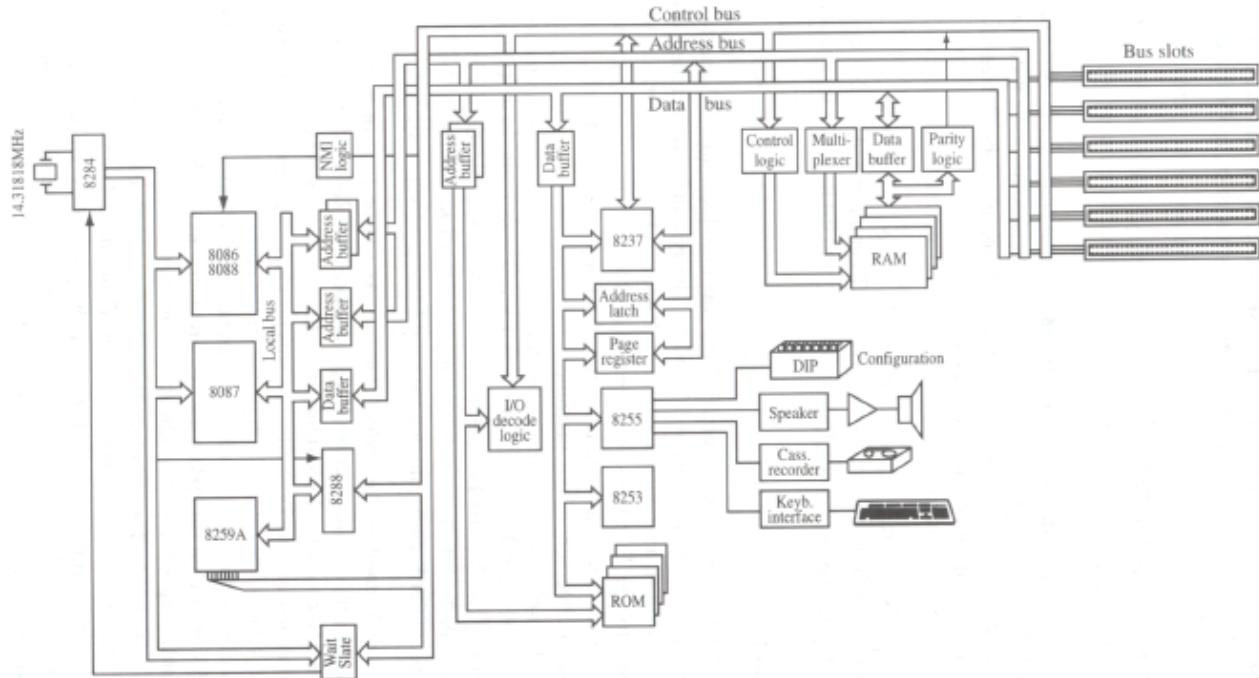


Figure 44. Arquitectura Bus ISA

ISA (8-bit Version)

(Typical 4.77 MHz Bus – IBM PC, IBM XT)

| | B1 | A1 | I/O CH CK | Signal | Name | Type* | Description |
|------------|-----|-----|-----------|-------------|----------------------|--------------|--|
| GND | | | D7 | A0–A19 | Address Lines | Output | 20-bit address bus |
| PRESET DRV | | | D6 | AEN | Address Enable | Output | High when DMA controller is controlling the buses. |
| +5V | | | D5 | | | | |
| IRQ2 | | | D4 | | | | |
| -5V | | | D3 | ALE | Address Latch Enable | Output | High when valid address signals are on the bus. |
| DRQ2 | | | D2 | | | | |
| -12V | | | D1 | CLK | System Clock | Output | In the original PC this was 4.77 MHz |
| res | | | D0 | D0–D7 | Data Lines | Input/Output | 8-bit data bus |
| +12V | | | | DACK0–DACK3 | DMA Acknowledge | Output | When low these signals acknowledge a peripheral's DMA request. |
| GND | B10 | A10 | | | | | |
| MEMW | | | | AEN | | | |
| MEMR | | | | A19 | | | |
| IOW | | | | A18 | DRQ1–DRQ3 | DMA Request | High to request a DMA transfer. DRQ0 is dedicated to memory refresh and is therefore not available on the bus. |
| TOR | | | | A17 | | | |
| DACK3 | | | | A16 | I/O CH CK | Input | Low to indicate an error condition and generate an NMI. |
| DRQ3 | | | | A15 | | | |
| DACK1 | | | | A14 | | | |
| DRQ1 | | | | A13 | I/O CH RDY | Input | High when the peripheral is ready. If low, a wait state is inserted into the current bus cycle. |
| DACK0 | | | | A12 | | | |
| CLK | B20 | A20 | | A11 | | | |
| IRQ7 | | | | A10 | IOR | Output | Low when inputting data from an I/O device. |
| IRQ6 | | | | A9 | | | |
| IRQ5 | | | | A8 | IOW | Output | Low when writing data to an I/O device. |
| IRQ4 | | | | A7 | IRQ2–IRQ7 | Input | High to request an interrupt from the processor. |
| IRQ3 | | | | A6 | | | |
| DACK2 | | | | A5 | MEMR | Output | Low when reading data from memory. |
| TC | | | | A4 | MEMW | Output | Low when writing data to memory. |
| ALE | | | | A3 | OSC | Output | Oscillator clock frequency. Normally 14.318180 MHz |
| +5V | | | | A2 | | | |
| OSC | | | | A1 | RESET DRV | Output | High during the power-on cycle. |
| GND | B31 | A31 | | A0 | T/C | Output | High to indicate the end of the DMA cycle. |

*Input to or output from the processor/bus controller.

Figure 45. Interfaz Bus ISA

8.11.2. PCI

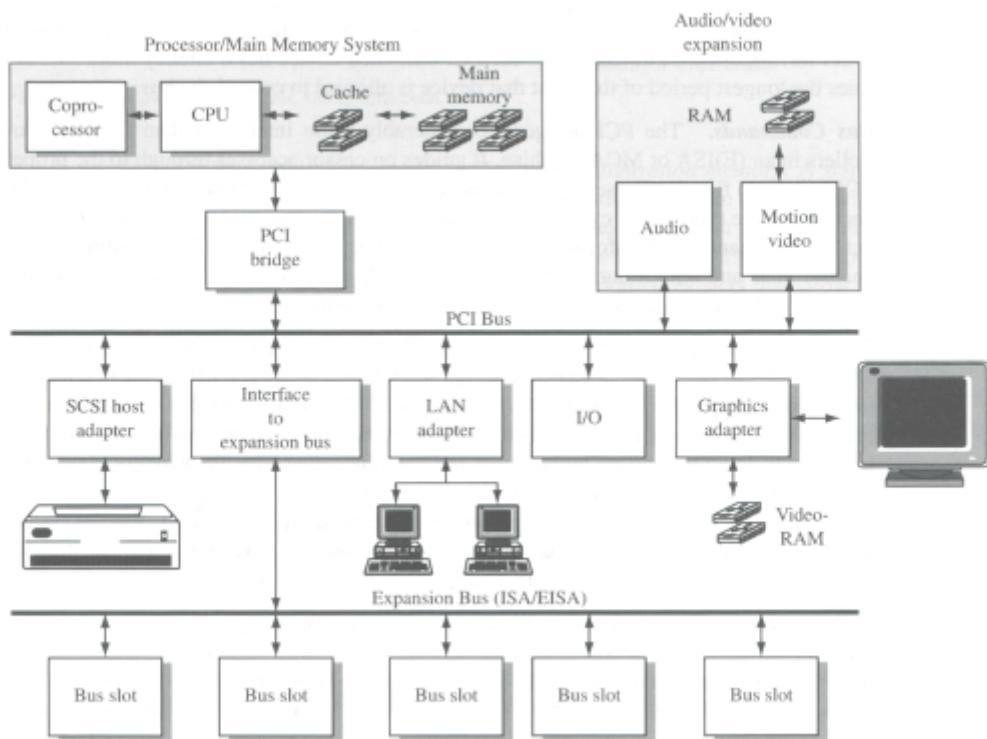


Figure 46. Arquitectura Bus PCI

8.11.3. North-South Bridge

Typical PCI Based x86 Computer Architecture

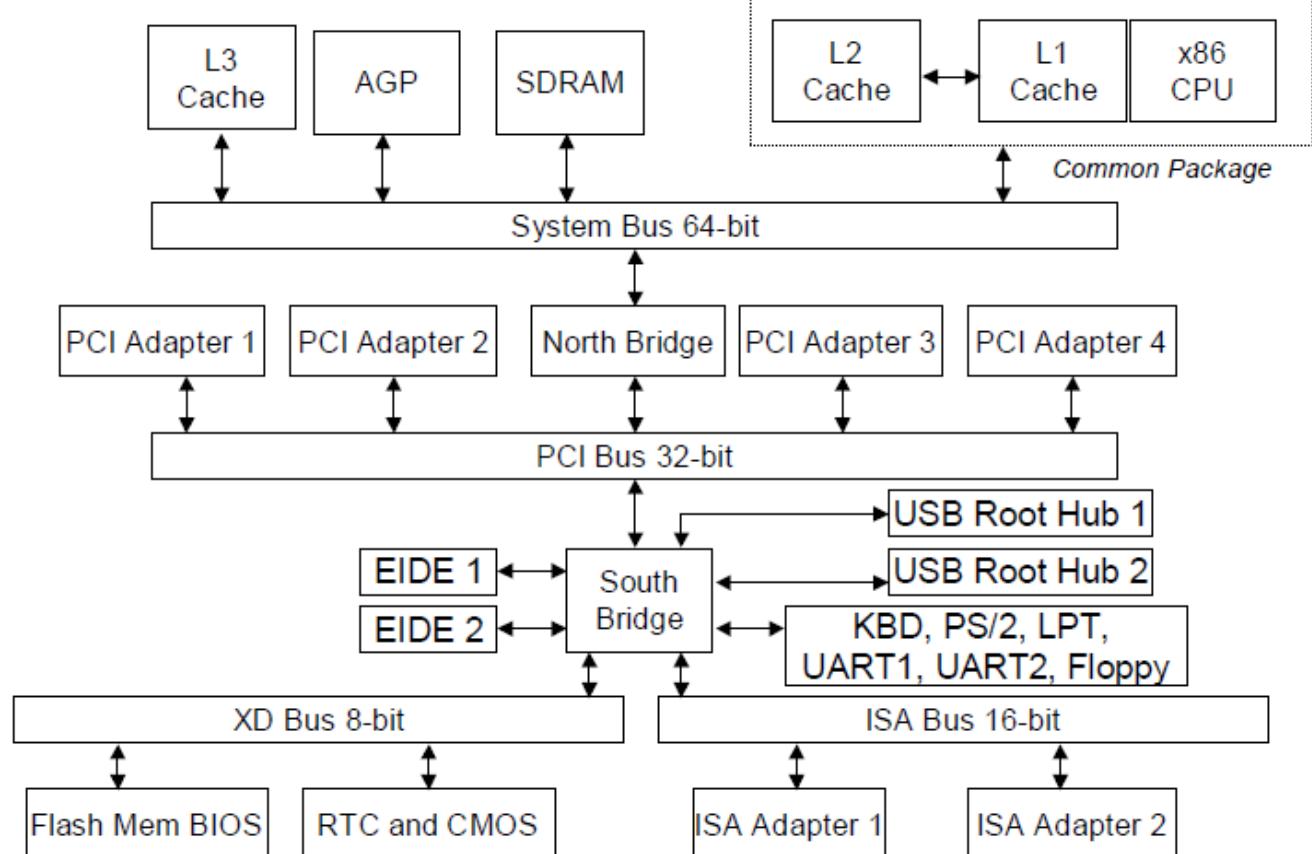


Figure 47. North and South Bridges

8.11.4. Chipset x58

- https://en.wikipedia.org/wiki/Intel_X58
- PCH: Platform Controller Hub
- FSB: Front Side Bus
- BSB: Back Side Bus
- FDI: Flexible Display Interface (para CPU que integran la controladora gráfica)
- DMI: Direct Media Interface
- ICH: i/o Controller Hub
- IOH: i/o Hub

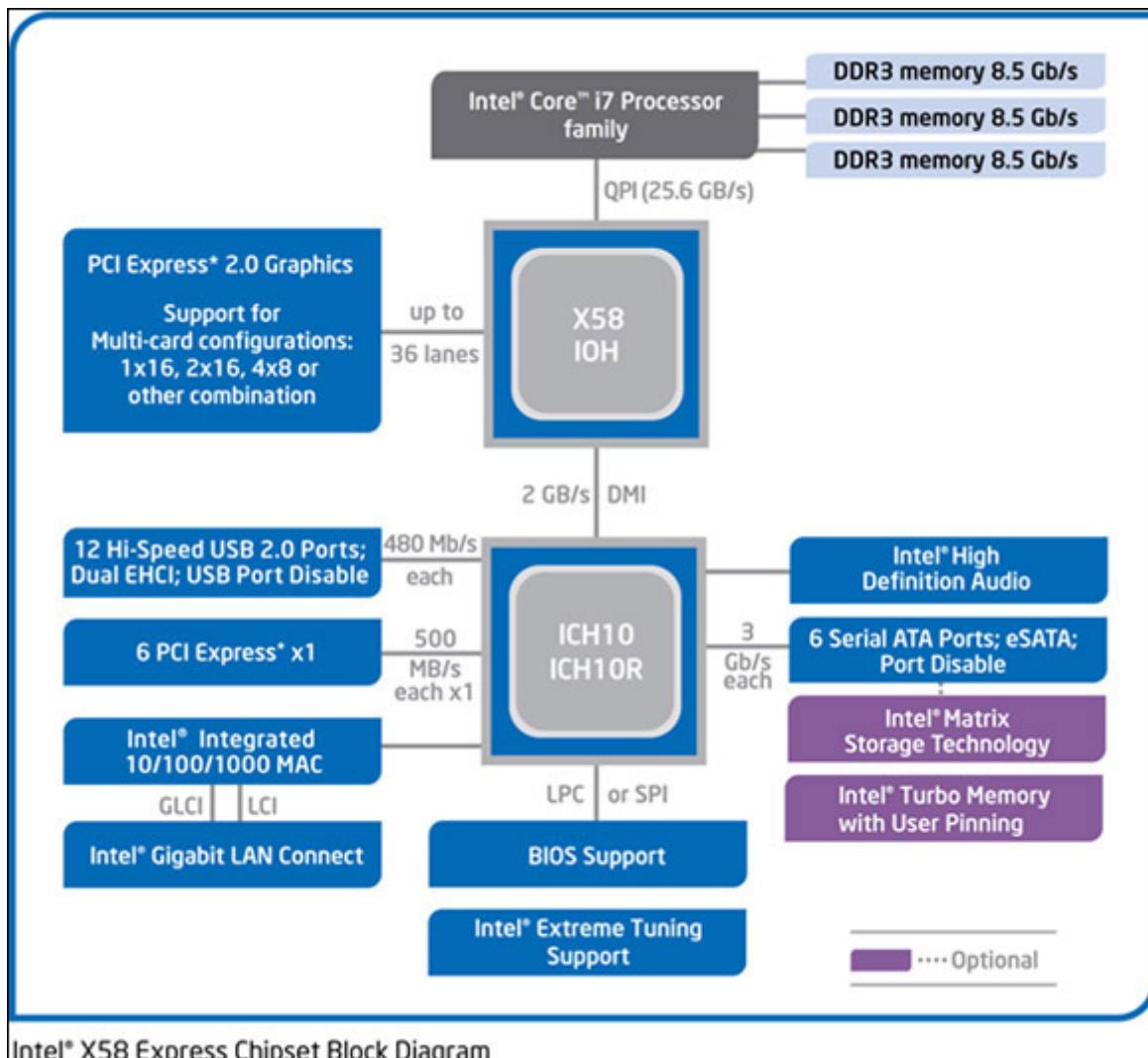


Figure 48. corei7 x58 chipset: año 2008

Chipsets supporting LGA 1366, LGA 2011, and LGA 2011-v3 CPUs.: X58 (2008), X79 (2011), X99 (2014).

8.12. Programacion de rutinas de entrada/salida

8.12.1. Software jerarquico del sistema operativo

- El driver o controlador sw es el nivel más bajo de la estructura sw: depende fuertemente del hardware de la computadora: programación en lenguaje C o ensamblador.

8.12.2. Instruction Set Architecture

- I/O access
 - OUTx : Sends a byte (or word or dword) on a I/O location. Traditional names are *outb*, *outw* and *outl* respectively. The "a" modifier enforces *val* to be placed in the eax register before the asm command is issued and "Nd" allows for one-byte constant values to be assembled as constants, freeing the edx register for other cases.

```
static inline
void outb( unsigned short port, unsigned char val )
{
    asm volatile( "outb %0, %1"
                  : : "a"(val), "Nd"(port) );
}
```

- El programa fuente en C incluye lenguaje ASM: Programa fuente en C con inline-asm.
- %0 hace referencia a la primera variable "a", %i hace referencia a la i-nésima variable.
- INx : Receives a byte (or word or dword) from an I/O location. Traditional names are *inb*, *inw* and *inl* respectively.

```
static inline
unsigned char inb( unsigned short port )
{
    unsigned char ret;
    asm volatile( "inb %1, %0"
                  : "=a"(ret) : "Nd"(port) );
    return ret;
}
```

- The register I/O instructions IN (input from I/O port) and OUT (output to I/O port) move data between I/O ports and the EAX register (32-bit I/O), the AX register (16-bit I/O), or the AL (8-bit I/O) register. The address of the I/O port can be given with an immediate value or a value in the DX register.

Intel Manual

- This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 16 or 14, "Input/Output," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.
- I/O ports can be mapped so that they appear in the I/O address space or the physical-memory address space (memory mapped I/O) or both.
- **memory-mapped:**
 - mediante una línea del bus de control se especifica si la dirección es de memoria principal o port i/o, en algún procesador mediante el M/IO# pin.
 - When using memory-mapped I/O, caching of the address space mapped for I/O operations must be prevented
- **I/O mapped**
 - i/o devices don't collide with memory, as they use a different *address space*, with different instructions to read and write values to addresses (ports). CPU decode the memory-I/O bus transaction instructions to select I/O ports These devices cannot be addressed using machine code instructions

that targets memory. What is happening is that there are two different signals: *MREQ* and *IOREQ*. The first one is asserted on every memory instruction, the second one, on every I/O instruction. So this code...

```
MOV DX, 1234h
MOV AL,[DX]      ; reads memory address 1234h (memory address space)
IN AL,DX        ; reads I/O port 1234h (I/O address space)
```

- The I/O device at port 1234h is connected to the system bus so that it is enabled only if the address is 1234h, RD (Read Data) is asserted and IOREQ is asserted.
- (64K) individually addressable 8-bit I/O ports

- **Protection**

- Port Mapped
 - Here, kernel and the device drivers are allowed to perform I/O, while less privileged device drivers and application programs are denied access to the I/O address space. Application programs must then make *calls* to the operating system to perform I/O.
- Memory mapped
 - the normal segmentation and paging protection affect the i/o port access.

8.12.3. Programación del Controlador de Interrupciones Programable

- [programación del pic](#)

- [Mapeo del PIC](#)

```
/* remap the PIC controller interrupts to our vectors
   rather than the 8 + 70 as mapped by default */

#define PIC1          0x20
#define PIC2          0xA0
#define PIC1_COMMAND    PIC1
#define PIC1_DATA     (PIC1+1)
#define PIC2_COMMAND    PIC2
#define PIC2_DATA     (PIC2+1)
#define PIC_EOI        0x20

#define ICW1_ICW4    0x01      /* ICW4 (not) needed */
#define ICW1_SINGLE   0x02      /* Single (cascade) mode */
#define ICW1_INTERVAL4 0x04    /* Call address interval 4 (8) */
#define ICW1_LEVEL    0x08      /* Level triggered (edge) mode */
#define ICW1_INIT     0x10      /* Initialization - required! */

#define ICW4_8086   0x01      /* 8086/88 (MCS-80/85) mode */
#define ICW4_AUTO    0x02      /* Auto (normal) EOI */
#define ICW4_BUF_SLAVE 0x08    /* Buffered mode/slave */
#define ICW4_BUF_MASTER 0x0C   /* Buffered mode/master */
#define ICW4_SFNM    0x10      /* Special fully nested (not) */

void remap_pics(int pic1, int pic2)
```

```

{
    UCHAR    a1, a2;

    a1=inb(PIC1_DATA);
    a2=inb(PIC2_DATA);

    outb(PIC1_COMMAND, ICW1_INIT+ICW1_ICW4);
    io_wait();
    outb(PIC2_COMMAND, ICW1_INIT+ICW1_ICW4);
    io_wait();
    outb(PIC1_DATA, pic1);
    io_wait();
    outb(PIC2_DATA, pic2);
    io_wait();
    outb(PIC1_DATA, 4);
    io_wait();
    outb(PIC2_DATA, 2);
    io_wait();

    outb(PIC1_DATA, ICW4_8086);
    io_wait();
    outb(PIC2_DATA, ICW4_8086);
    io_wait();

    outb(PIC1_DATA, a1);
    outb(PIC2_DATA, a2);
}

```

8.12.4. Driver del Teclado

- Fijarse cómo se programa teniendo en cuenta el mecanismo de atención a las interrupciones.
- Buscar el código fuente de un kernel sencillo.

8.12.5. parallel port

Desde Espacio de Usuario

- Es necesario realizar llamadas al sistema ya que no podemos acceder desde el espacio de usuario directamente al HW
- [Direcciones base de los puertos](#)
- Acceso a un puerto en linux desde el espacio de usuario

```

/* led_bloq_mayus.c: very simple example of port I/O
 *
 * This code active LED keyboard CAP, just a port write, a pause,
 * and a port read. Compile with `gcc -O2 -o led_bloq_mayus led_bloq_mayus.c',
 * and run as root with `sudo ./led_bloq_mayus'.
 */

```

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

```

#include <stdlib.h>
#include <unistd.h>
#include <sys/io.h>

#define BASEPORT 0x0060 /* keyboard */

int main()
{
    /* Get access to the ports */
    if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}
    printf("\n\t\t Port -> registro status: %d\n", inb(BASEPORT + 1));

    /* Set the data signals (D0-7) of the port to all low (0) */
    outb(0xED, BASEPORT);

    /* Sleep for a while (100 ms) */
    usleep(1000);
    printf("\n \t\tActiva el LED de la tecla BLOQ MAYUS \n ");
    outb(0x07, BASEPORT);

    usleep(1000);

    /* We don't need the ports anymore */
    if (ioperm(BASEPORT, 3, 0)) {perror("ioperm"); exit(1);}
    exit(0);
}

```

- <http://tldp.org/HOWTO/IO-Port-Programming-2.html>
- man ioperm
- man inb
- `cat /proc/ioports`
- <http://opensourceforu.efytimes.com/2011/07/accessing-x86-specific-io-mapped-hardware-in-linux/>

8.12.6. Serial communication RS-232

- [tutorial](#)
 - Tarjeta Avr Atmega 8bits → Pej Arduino One
 - Dos casos: Polling i/o e interrupt-driven i/o
- Puerto UART (RS-232)
 - Conector físico
 - Comunicación semiduplex entre dos terminales: DTE (PC) y DCE (Arduino)
 - Señales Tx Rx
 - Registros del puerto
 - Control
 - Estado
 - Datos Rx y Tx: UDR

- Programación
 - Librería
 - Cross toolchain
 - Algoritmo: Diagrama de flujo
 - Casos Polling y Interruption
 - Estructura modular: dos módulos
 - Símbolos
 - Buffer de datos i/o
 - Nombre del vector de interrupción

8.13. Ejercicios

- Capítulo 7 del libro de texto William Stallings.

Chapter 9. Unidad de Memoria

9.1. Introducción

9.1.1. Temario

9. Organización de la memoria
 - a. Jerarquía de memoria
 - b. Latencia y ancho de banda
 - c. Memoria cache
 - d. Memoria virtual

9.1.2. Libro: William Stalling

1. Introducir conceptos de William.
 - William tiene un capítulo para la memoria principal y otro para memoria cache
 - Capítulo 4 : Caché
 - La introducción de la memoria cache tiene Conceptos Generales
 - Capítulo 5 : Memoria Interna (DRAM)
 - Capítulo 6 : Memoria Externa (Almacenamiento Periférico)
 - Capítulo 8 : Sistemas Operativos: Gestión de Memoria

9.1.3. Historia

- El gran avance del Ingeniero John Von Neumann fue desarrollar la computadora IAS en la cual los programas no eran cableados sino almacenados electrónicamente en una unidad de memoria denominada Selectron. Los programas eran "editados" mediante la escritura de tarjetas de cartón perforadas que posteriormente eran convertidas en secuencias de dígitos binarios para poder ser almacenadas en código binario en la unidad de memoria Selectrón. De esta manera surgió el concepto de "programa almacenado" o software y el desarrollo de las **unidades de memoria**.
- La memoria no consistía de una única unidad sino que se estructuraba en distintos niveles:
 - Nivel cpu: registros PC,MAR,MBR, IR, IBR, Acumuladores AC y AR : registros con capacidad para almacenar una instrucción y un dato.
 - Memoria principal: memoria Selectron con capacidad para almacenar programas agrupando en una sección las instrucciones y en otra sección los datos. Capacidad para direccionar 4K palabras de 40 bits cada palabra
 - Memoria secundaria: Tambores magnéticos "drum" con capacidad para almacenar una colección de programas.

9.1.4. Interés

- Programación en un lenguaje de alto nivel
 - ¿Tenemos en cuenta el concepto memoria? → Abstracción de los mecanismos de gestión de memoria por parte del S.O, hardware, etc
 - Tener conocimientos de la estructura, organización y gestión de la memoria ayuda a la hora de programar → Fase de depuración, diseño, etc
 - Ingeniería de programación y de sistemas

- Conocimientos previos sobre memoria: temas previos de la asignatura Estructura de Computadores.
 - variable, puntero, registros, secciones, direccionamiento, violación de segmento, linker, ...
- Conocimiento de S.O.
 - Gestión de la memoria de los procesos, paginación, memoria virtual, TLB, etc
- Objetivo
 - Qué: Almacenar: datos e instrucciones → programas → ficheros → procesos
 - Para qué:
 - Arquitectura von-Neumann: programa almacenado.
 - Ciclo de Instrucción: Captura (datos, instrucciones) de la CPU. Esquema de bloques CPU-RAM.
 - Cómo: Cómo se almacenan ,cómo se capturan?
- Físicamente la memoria es
 - chip de semiconductor conectado a la CPU.
 - memoria semiconductor interna a la CPU
 - memoria magnética de almacenamiento masivo
- El tema Memoria está aislado del resto de:
 - la arquitectura de la computadora?:
 - de la CPU? del kernel?
 - la programación?
 - instrucciones y datos?
 - qué es un array?
 - un goto?
 - en ensamblador que es la directiva .text?
 - qué es la pila?

9.1.5. Perspectivas

- El concepto de memoria puede ser estudiado en función de diferentes perspectivas.
- Gestión de la Memoria de los procesos en ejecución por parte del kernel del sistema operativo vs Organización de Memoria (ficheros,secciones,jerarquía de memoria ,..)
- Software:
 - Programación:
 - variables (reserva e inicialización de memoria), punteros, asignación dinámica de la memoria malloc(), secciones de memoria (text,data,rodata,bss,etc..), pila, ..
 - Herramientas:
 - Compilador, Linker (direcciones reubicables, resolución direcciones, segmentos de memoria,..), Cargador (memoria física, mapa de memoria, etc ..), volcado de memoria (objdump, ..)
 - Sistema Operativo:
 - Gestor de memoria virtual
 - Gestión del sistema de ficheros virtual
- Hardware:
 - Unidad de gestión de memoria MMU (Memory Management Unit): Convertidor del espacio de direcciones virtual en físico.

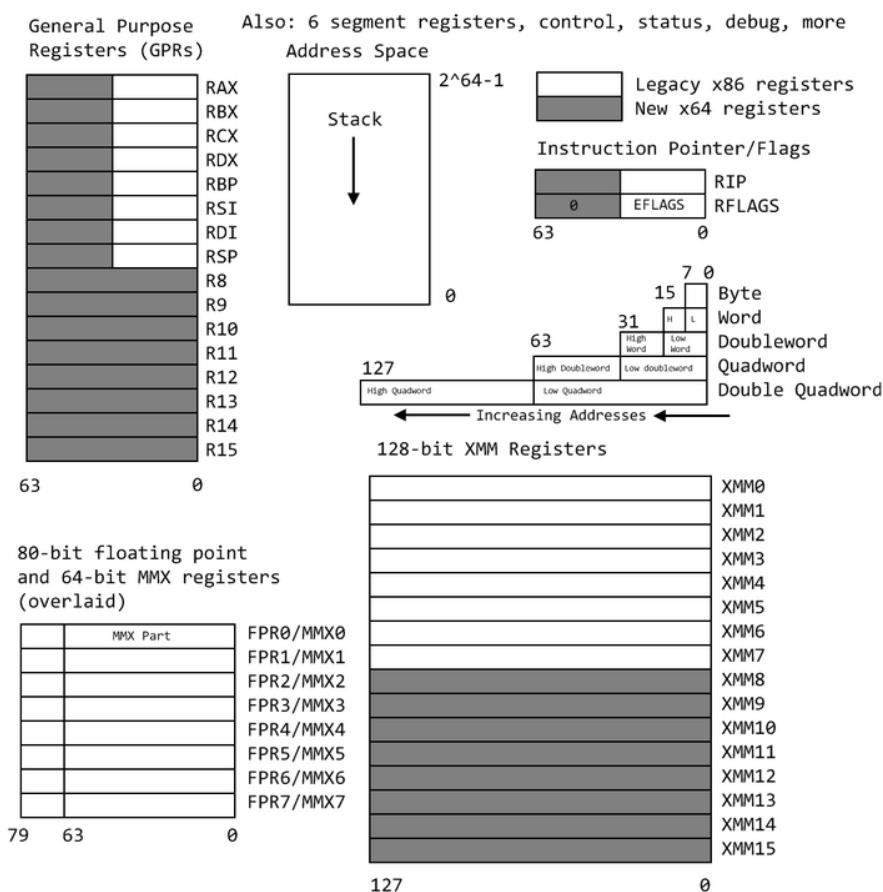
- Módulos de Memoria
 - tarjetas, chips, conexión buses
 - características: capacidad, velocidad, consumo, tecnología

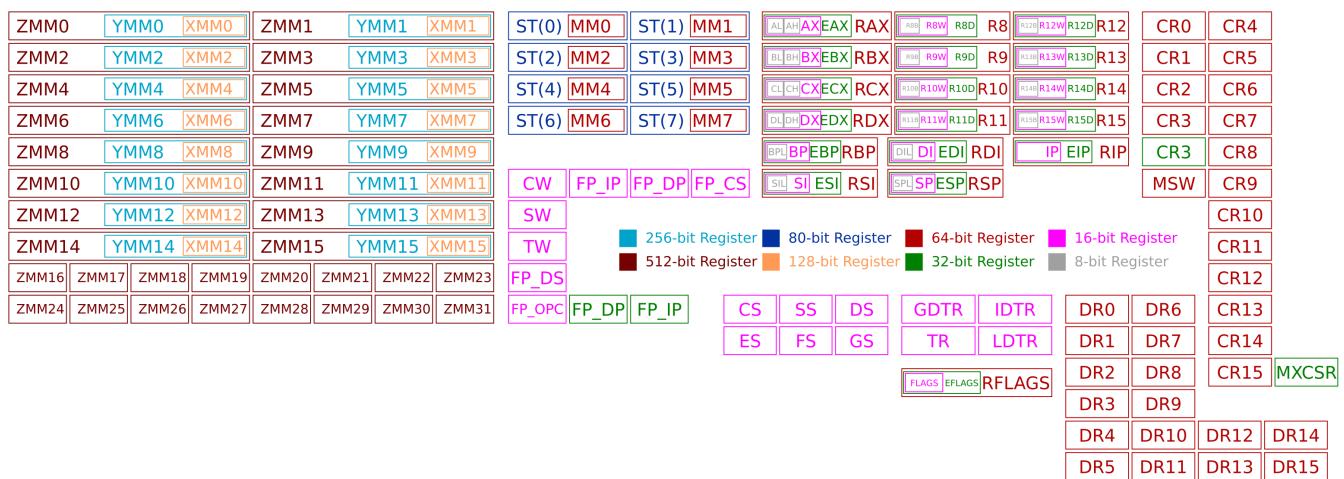
9.1.6. Jerarquía de Memoria

- Hoy en día la tecnología de semiconductor ha conseguido avances en capacidad y memoria manteniendo la estructura por niveles en la siguiente jerarquía:
 1. Jerarquía memoria
- Niveles: L0, L1, L2
- Memoria Registros CPU
- Memoria Cache
- Memoria Principal
- Memoria Secundaria: disco, pen-drive
- Características
 - Capacidad ascendente top-down
 - Tiempo del ciclo de memoria ascendente top-down

9.2. Registros

9.2.1. Arquitectura amd64





- Observar que al igual que rax incluye a eax y eax incluye ax y ax incluye al, también zmm incluye a ymm e ymm incluye a xmm
- [wiki x86](#)
- General Purpose Registers x86
 - RAX,etc
 - RFLAGS
 - CS-DS-SS-
 - ST0-ST7
 - Float Point Registers
 - alias de los registros FPU
 - eight 80-bit wide registers: 32-, 64-, or 80-bit floating point, 16-, 32-, or 64-bit (binary) integer, and 80-bit packed decimal integer.
 - MMX
 - ya en desuso y superados por XMM.
 - MMX instructions: *integer SIMD* (Single Instruction Multiple Data). MMX is a instruction set designed by Intel, introduced in 1997 with its P5-based Pentium line of microprocessors. a single instruction can then be applied to two 32-bit integers, four 16-bit integers, or eight 8-bit integers at once.
 - MM0-MM7 (64 bits). Each register is 64 bits wide and can be used to hold either 64-bit integers, or multiple smaller integers in a "packed" format.
 - packed data types: two 32-bit integers, four 16-bit integers, or eight 8-bit integers concurrently → solo enteros.
 - Trabaja sólo con enteros pero por causas de compatibilidad en los cambios de contexto de los S.O. se creo un alias entre los MMX y los FPU generando el problema de no poder utilizar en una misma aplicación los FP y los MMX ya que las operaciones de uno afecta al otro.
 - Los registros FP del FPU x87 tiene acceso modo pila mientras que los MMX tienen acceso aleatorio.
 - XMM
 - Ya en desuso y superados por YMM.
 - XMM0-XMM15 (128 bits)
 - SSE (Streaming SIMD Extensions) instruction. Is an *floating point SIMD* instruction set extension to the x86 architecture introduced on 1999 with Pentium III.
 - Evolución de los MMX. Equivalente a MMX pero con datos de tipo coma flotante.

- Se puede operar simultáneamente con los FP, MMX y XMM.
- YMM
 - YMM0-YMM15 (256 bits)
 - La extensión de los XMM a 256 bits
 - **AVX**: Advanced Vector Extensions instructions
 - Únicamente datos en coma flotante: *floating point SIMD*
 - AVX introduces a three-operand SIMD instruction format, where the destination register is distinct from the two source operands
 - Compatibilidad: The AVX instructions support both 128-bit and 256-bit SIMD
 - Intel comienza con este set en el 2011: Sandy Bridge processor, Q1 2011.
 - **AVX2**
 - Haswell microarchitecture año 2013: Haswell processor, Q2 2013
 - AVX2 - *Integer* data types expanded to 256-bit SIMD
- ZMM
 - (ZMM0-ZMM31) : 512 bits
 - **Intel AVX-512**: Julio 2013
 - Programs can pack eight double precision or sixteen single precision floating-point numbers, or eight 64-bit integers, or sixteen 32-bit integers within the 512-bit vectors. This enables processing of twice the number of data elements that AVX/AVX2 can process with a single instruction and four times that of SSE.
 - **AVX-512 instructions**
- Control Registers x86:CRx
 - Controlar por ejemplo la paginación de memoria.
- Debug Registers x86:DRx
 - Se utilizan para implementar por ejemplo las direcciones de los puntos de ruptura: DR0-DR3
- Check
 - En linux con la instrucción **cpuid** podemos chequear la compatibilidad de la cpu con la extensiones ISA: mmx,sse,avx,etc
- Conceptos
 - SIMD: Single instruction multiple data.
 - Vectorizing code: instrucción que operan con vectores → una operación sobre múltiples datos simultáneamente.
 - DSP: Digital Signal Processor
- Tipos de registros
 - https://en.wikibooks.org/wiki/Microprocessor_Design/Register_File
 - Register File: memoria estática formada por una secuencia de registros con un bus de direcciones que mediante un decodificador selecciona uno de los registros.
 - Register Banking: dos posibles interpretaciones.
 - Banked Registers for Interrupt Handling: En lugar de utilizar la memoria principal (pila) para salvar y recuperar los registros cuando es interrumpido la ejecución de un proceso debido a una interrupción externa, utilizamos registros internos de la CPU para tal propósito: Se incrementa la velocidad de respuesta a una interrupción. La forma de implementar esta técnica es renombrando los registros utilizados por la rutina que interrumpe respecto de la rutina interrumpida.

- Agrupamiento por bancos: El conjunto de registros se agrupa por bancos que pueden ser accedidos simultáneamente.

9.3. Memoria Principal (RAM Dinámica DRAM)

9.3.1. Tipos de memoria de semiconductores

| Memory Type | Category | Erasure | Write Mechanism | Volatility |
|-------------------------------------|--------------------|---------------------------|-----------------|-------------|
| Random-access memory (RAM) | Read-write memory | Electrically, byte-level | Electrically | Volatile |
| Read-only memory (ROM) | Read-only memory | Not possible | Masks | Nonvolatile |
| Programmable ROM (PROM) | | | Electrically | |
| Erasable PROM (EPROM) | Read-mostly memory | UV light, chip-level | | |
| Electrically Erasable PROM (EEPROM) | | Electrically, byte-level | | |
| Flash memory | | Electrically, block-level | | |

- SSD: Solide State Drive : Memoria flash de alta capacidad
 - SLC, TLC y MLC (Single-, Triple- y Multi- level cell)

9.3.2. Memoria principal semiconductora

- Ver enlace imágenes al final de los apuntes
- RAM:
 - Semiconductor: transistores.
 - Random Acces Memory
 - Operaciones de lectura y escritura
 - Volátil
 - Tipos
 - SRAM: Static RAM .
 - Mientras está alimentada la información no se pierde.
 - Estructura de la Celda 6T: seis transistores. Tamaño y consumo elevados. Latencia y capacidad reducidas.
 - Memoria caché.
 - DRAM: Dynamic RAM .
 - Estructura de la celda: 1C1T: un condensador y un transistor. Tamaño y consumo reducidos. Latencia y capacidad elevadas.
 - Memoria principal.
 - Tiene fugas por lo que necesita periódicamente una reescritura (DINAMISMO).

- Asíncrona DRAM:
- Síncrona SDRAM:

Organización

- Celda de memoria:
 - es la unidad básica de almacenamiento de un bit (Binary digit). El bit es un valor lógico *High* o *Low*, 1 o 0
 - acceso a la celda:
 - la línea de direcciones selecciona la celda a leer o escribir
 - la línea de bit es la línea de entrada/salida del bit a leer o escribir.
- Matriz: las celdas de memoria se organizan en una estructura 2D matricial formadas por filas y columnas
- Bus del Sistema: bus de interconexión entre el controlador MC y la CPU.
- Bus de Memoria: bus de interconexión entre el controlador MC y la MP
 - Bus Direcciones:
 - El bus de direcciones transfiere el código de la palabra a seleccionar
 - La dirección se almacena temporalmente en el buffer de direcciones de la memoria
 - El bus de direcciones se conecta al buffer de direcciones de la memoria
 - El buffer de direcciones se conecta a la entrada del decodificador de direcciones de la memoria
 - La dirección se decodifica. La salida del decodificador activa la dirección de memoria del dato/instrucción a leer o escribir
 - Bus Datos:
 - El dato de salida o entrada se almacena temporalmente en el buffer de datos i/o de la memoria
 - Las celdas no se conectan directamente al buffer de datos i/o de la memoria
 - las salidas de las celdas seleccionadas son amplificadas para detectar si almacenan 0 o 1
 - El bus de datos esta conectado al buffer de datos de la memoria
 - Bus Control:
 - Es necesario alimentar la memoria con una tensión continua de unos pocos voltios (1v)
 - Señal de lectura y escritura que activa la CPU o el controlador E/S
 - Señal de reloj de sincronismo. Sincroniza las tareas a realizar entre la MP y el controlador de memoria (MC)
 - Bus chip select:
 - Señal *Chip Select* (CS) de selección del módulo de memoria que lo conecta a los buses de direcciones y de datos . Si la señal CS no está activa el módulo de memoria está desconectado de los buses.
- Controlador de Memoria (MC)
 - La MP no se conecta directamente a la CPU. El controlador MC hace de intermediario.
 - El controlador MC se conecta por un lado a la CPU y por otro lado a la memoria MP.
 - La CPU envía comandos al controlador MC para que actue sobre la MP.
 - El controlador MC es un secuenciador que sabe cómo actuar sobre la estructura interna de la memoria para:
 - qué módulo seleccionar, qué chip seleccionar, qué palabra seleccionar.
 - leer y escribir un dato

- otras acciones sobre la memoria como mantenimiento, chequeo, detección de errores, etc
- Memory Management Unit (MMU)
 - Las direcciones con que opera la Unidad de Control de la CPU en sus registros de propósito general, contador de programa, etc, no son físicas → son virtuales
 - Cuando programamos, el programador, el compilador, el linker, el desensamblador, el depurador, etc trabajan en el espacio virtual. El módulo ejecutable ELF y los procesos hacen referencia al espacio virtual.
 - Los procesos (programas que están siendo ejecutados por la CPU) operan con direcciones del espacio virtual → memoria virtual del proceso
 - MMU: circuito electrónico HW que convierte direcciones del espacio virtual (CPU) en direcciones físicas de la MP y que serán las que se transfieran al bus del controlador de la caché y al controlador de memoria MC para poder acceder a la memoria física.

DRAM (Dynamic Random Access Memory)

- Celda
 - Estructura física:
 - Es un condensador Metal-Dielectrico-Metal(Polysilicio) fabricado en un substrato de Silicio.
 - Su capacidad es del orden de femto-faradios: $C=10\sim(-15)F$
 - Si le aplicamos una tensión de 1mv la carga almacenada $Q=CV= 1mv \cdot 1fF = 1 \cdot 10^{-18}$ culombios que equivale a una decena de electrones.
 - Su forma es la de un cilindro empotrado en el substrato.
 - La sección transversal del condensador es del orden de 30 nm en el año 2010
 - [evolución proceso tecnológico](#)
 - La densidad de condensadores es del orden del giga → 10~9 condensadores.
 - Es necesario conectar el condensador a las líneas de direcciones y de bit para acceder a él. Se conecta a través de UN transistor CMOS que hace de interruptor.

DRAM (Operaciones de lectura-escritura-refresco)

- Almacenamiento:
 - el condensador inicialmente no está conectado a ninguna línea ya que su interruptor está abierto
 - en circuito abierto el condensador almacena la carga mientras está alimentado → volátil
 - el condensador tiene FUGAS y se descarga a través del substrato. Es necesario reescribir el bit cada 64 ms: DYNAMIC (la información que almacena no puede ser estática, hay que REFRESCARLA PERIODICAMENTE)
- Escritura:
 - Cerramos el interruptor (línea de dirección) para conectar el condensador a la línea de bit (línea de dato)
 - A través de la línea de bit cargamos (H) o descargamos (L) el condensador
- Lectura:
 - Una vez seleccionada la celda a leer, está se conecta al Sensor de Carga (amplificador) que detecta su estado y lo escribe en el buffer i/o
 - Esta lectura es DESTRUCTIVA, dejando el condensador descargado. Es necesario que el amplificador realice el condensador a su estado original. La escritura del buffer i/o y la RE-escritura del condensador se dan simultáneamente.

- Refresco
 - Es necesario leer y reescribir todos los condensadores. Esta operación la realiza el sensor de carga.
 - Es necesario reescribir todas las celdas en un tiempo inferior a los 64ms.

Ejemplo de Estructura

- <https://www.anandtech.com/print/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask>

DRAM Matriz(Array 2D)

- Ejemplo: un bus de direcciones de 30 líneas son 2^{30} CELDAS, a seleccionar UNA de ellas.
- Mediante un DEMULTIPLEXOR de 30 entradas y 2^{30} salidas podemos seleccionar 1 de las celdas
- Un demultiplexor de 2^{30} salidas es muy complejo y costoso
- Solución:
 - organizar las CELDAS en un array 2D : Filas y columnas: 1 DEMUX o DECODIFICADOR para filas y 1 DEMUX/MUX o DECODIFICADOR para las columnas
 - $2^{30} = 2^{15} \times 2^{15}$ = Ahora el número de salidas de cada demux se ha reducido de 2^{30} a 2^{15} , es decir, un factor raíz cuadrada.
 - word line selecciona todas las columnas de una fila (ROW) de celdas
 - bit line selecciona una de las columnas (COL) de la fila seleccionada
 - el resultado es seleccionar una CELDA del ARRAY y cargar (LECTURA) el BUFFER I/O
- Bus de direcciones muy denso: ejemplo de 30 líneas
 - Podemos diseñar un bus con la mitad de líneas y multiplexar en dos tiempos el código de direcciones(parte que selecciona la fila y parte que selecciona la columna).
 - Multiplexación temporal de la dirección de filas y la dirección de columnas: REDUCIMOS EL NUMERO DE LINEAS EN LA PLACA BASE
 - RAS: Row Address Strobe : Señal que válida el bus de direcciones indicando que es el código que selecciona la FILA del array.
 - CAS: Row Address Strobe : Señal que válida el bus de direcciones indicando que es el código que selecciona la COLUMNA del array.
- Burst (ráfaga)
 - Una vez seleccionada una fila de celdas (OPEN ROW) si queremos celdas consecutivas de la misma columna podemos leer o escribirlas consecutivamente en cada ciclo de reloj . Bloque de palabras a transferir a/desde la memoria Caché. El controlador de memoria ha tenido que enviar un comando a la memoria para configurar el número de palabras de la ráfaga.

Logica del Chip (Figura 5.3 del libro)

- Componentes:
 - buffers: direcciones y columnas
 - decodificadores: decodifican el código de dirección de fila y de columna y seleccionan (fila,columna) una celda.
 - sensor de carga (amplificador): detecta si la celda está cargada o descargada y da como salida un *H* o *L* en el buffer i/o.
 - 4 señales de control: RAS, CAS, WE, OE

- la combinación de señales de control (2^4) se utiliza también para codificar los **comandos** del controlador de memoria.
- COMANDOS: son ordenes a los módulos de memoria donde las características de la memoria como el timing (tiempos de latencia, ciclo, etc) y el burst length (número de palabras por bloque, longitud de la ráfaga) son programables y por lo tanto la CPU puede configurar estos parámetros.
- circuitería de refresco:
 - contador de direcciones y temporizador
 - la asociación JEDEC recomienda un refresco completo cada 64 ms.

Encapsulado

- La memoria de semiconductor ocupa unos pocos mm² que debe de ser protegido (térmica y mecánica) y permitir que las conexiones sean robustas para permitir su soldadura a las líneas externas por lo que requiere un encapsulado de plástico.
- Los terminales del encapsulado se denominan PIN y son soldados a la tarjeta de memorias.
 - pines o terminales:
 - address bus (A0-A29)
 - data bus (DQ0-DQ7) : los chips no tienen 64 pines de datos : 1,2,4,8.
 - alimentación Vcc
 - masa Vss
 - chip select /CS
 - write enable (/WE): L(escritura) H(lectura)
 - output enable (/OE): L(los pines de datos se conectan al bus de datos)

Temporización de la operación de lectura/escritura

Table 14. Asignación de terminales de la SDRAM

| Señales | Descripción |
|---------|--------------------------------------|
| A0-A29 | Entrada de dirección de celda |
| CLK | Entrada del reloj del bus de memoria |
| /CS | Selección del chip |
| /RAS | Selección de dirección de fila |
| /CAS | Selección de dirección de columna |
| /WE | Habilitación de escritura |
| DQ0-DQ7 | Entrada/Salida de datos |

La barra inclinada / significa señal negada: lógica negativa : se activa a nivel Low (L).

Ejemplo extraido de [wikipedia](#):

Table 15. DRAM Asíncrona: Temporización

| | "50 ns" | "60 ns" | Description |
|-----|---------|---------|---|
| tRC | 84 ns | 104 ns | Random read or write cycle time (from one full /RAS cycle to another) |

| | "50 ns" | "60 ns" | Description |
|------|---------|---------|---|
| tRAC | 50 ns | 60 ns | Access time: /RAS low to valid data out |
| tRCD | 11 ns | 14 ns | /RAS low to /CAS low time |
| tRAS | 50 ns | 60 ns | /RAS pulse width (minimum /RAS low time) |
| tRP | 30 ns | 40 ns | /RAS precharge time (minimum /RAS high time) |
| tPC | 20 ns | 25 ns | Page-mode read or write cycle time (/CAS to /CAS) |
| tAA | 25 ns | 30 ns | Access time: Column address valid to valid data out (includes address setup time before /CAS low) |
| tCAC | 13 ns | 15 ns | Access time: /CAS low to valid data out |
| tCAS | 8 ns | 10 ns | /CAS low pulse width minimum |

- Sincronismo
 - DRAM : asíncrona: responds as quickly as possible to changes
 - SDRAM significantly revises the asynchronous memory interface, adding a **clock** (and a clock enable) line. All other signals are received on the *rising edge* of the clock. No responde tan rápido como es posible, sino que espera al flanco de subida.
- NO vemos la memoria DRAM asíncrona, únicamente el concepto.

Table 16. DRAM Síncrona: Temporización

| | PC-3200 (DDR-400) | | | | PC2-6400 (DDR2-800) | | | | PC3-12800 (DDR3-1600) | | | | Description |
|------|-------------------|------|--------|------|---------------------|---------|--------|------|-----------------------|----------|--------|-------|--|
| | Typical | | Fast | | Typical | | Fast | | Typical | | Fast | | |
| | cycles | time | cycles | time | cycles | time | cycles | time | cycles | time | cycles | time | |
| tCL | 3 | 15ns | 2 | 10ns | 5 | 12.5 ns | 4 | 10ns | 9 | 11.2 5ns | 8 | 10 ns | /CAS low to valid data out (equivalent to tCAC) |
| tRCD | 4 | 20ns | 2 | 10ns | 5 | 12.5 ns | 4 | 10ns | 9 | 11.2 5ns | 8 | 10ns | /RAS low to /CAS low time |
| tRP | 4 | 20ns | 2 | 10ns | 5 | 12.5 ns | 4 | 10ns | 9 | 11.2 5ns | 8 | 10ns | /RAS precharge time (minimum precharge to active time) |
| tRAS | 8 | 40ns | 5 | 25ns | 16 | 40ns | 12 | 30ns | 27 | 33.7 5ns | 24 | 30ns | Row active time (minimum active to precharge time) |

- When describing synchronous memory, timing is described by **memory bus clock cycle counts** separated by hyphens. These numbers represent tCL-tRCD-tRP-tRAS in multiples of the DRAM *clock cycle time*

latency times

- En este contexto latency es sinónimo de **retardo**. Distinto concepto de Memory Latency que es el tiempo de acceso.
- tCL :Cas Latency . Retardo desde la señal CAS hasta la obtención del dato en el buffer i/o
- tRCD :Ras Cas Delay. Retardo de la señas RAS hasta la señal CAS

- tRP :Ras Precharge. Mínimo retardo entre la precarga y la activación
- tRAS :Row Active Time. Mínimo tiempo que tiene que transcurrir la activación de la fila y el inicio de la precarga.
- **Tacceso:** tCL+tRCD : desde que se valida la dirección del bus hasta la obtención en el buffer i/o del dato referenciado.
- **Tciclo**(lectura o escritura) del bus: tCL+tRCD+tRP+tBURST ó tCL+tRCD+tRAS(si hemos transferido un comando a la MP):
 - T acceso más el retardo en ser transferido a la CPU. Tiempo entre dos lecturas o dos escrituras consecutivas,
- tBURST: tiempo necesario para transferir un bloque de palabras:RAFAGAS: no se realizan transferencias de 1 byte: 2,4,8,16, ..
- El módulo MP es programable por lo que podemos alterar los tiempos tCL-tRCD-tRP-tRAS y también la longitud de la ráfaga(burst o bloque)
- El módulo MP suele indicar la secuencia tCL-tRCD-tRP-tRAS con valores típicos de ciclos reloj

Ejemplo PC2-6400 (DDR2-800) 5-5-5-16

- Módulo PC2-6400 (DDR2-800) 5-5-5-16
- PC2 : SDRAM de segunda generación → Double_Data_Rate x2
- 6400 MB/s de ancho de banda
- 800MHz de ciclo efectivo de reloj del bus del sistema
 - Cada palabra se transfiere en un ciclo de 800MHz.
 - Ciclo de Reloj del Bus de memoria 400MHz
 - Clock cycle time = 1/400Mhz = 2.5ns
- 5-5-5-16 son los ciclos de reloj (400MHz↔2.5ns) de los tiempos tCL-tRCD-tRP-tRAS → 12.5ns-12.5ns-12.5ns-40ns

Table 17. Glosario

| tiempo | Descripción | tiempo | Descripción |
|--------|------------------------------|--------|--------------------------|
| tCL | CAS latency | tRRD | RAS to RAS delay |
| tCR | Command rate | tRTP | Read to precharge delay |
| tPTP | precharge to precharge delay | tRTR | Read to read delay |
| tRAS | RAS active time | tRTW | Read to write delay |
| tRCD | RAS to CAS delay | tWR | Write recovery time |
| tREF | Refresh period | tWTP | Write to precharge delay |
| tRFC | Row refresh cycle time | tWTR | Write to read delay |

- memory timing
- Fig 5.13 del libro de texto: Lectura de SDRAM (longitud de ráfaga=4, CL=2)

Agrupamientos: Módulos-Rank-Chips-Bank

Fig 5.12 : Módulo SDRAM.

- Jerarquía: Estructura de la memoria DRAM en agrupamientos de direcciones.

Channel

- Channel :
 - interfaces del controlador de memoria con el bus del sistema.
 - Cada canal tiene su propio bus de memoria físico.
 - El controlador tiene acceso al bus del sistema y a más de un bus de memoria.
 - Todos los canales de un mismo controlador de memoria conforman todo el espacio de memoria física, por lo tanto un controlador tiene asignado un canal lógico (todo el espacio de memoria) formado por varios físicos (distintos espacios de memoria)

Module DIMM

- Module:
 - Proporciona la conexión física al bus de datos (palabra de 64 bits), al bus de direcciones, al bus de control y al bus de chip-select (CS) del BUS DE MEMORIA.
 - Es la tarjeta de memoria encapsulada que se inserta en el socket de la placa base conectándose al bus de memoria del controlador de memoria (MC)
 - Para los PC la conexión de los módulos de memoria es **DIMM** y para los portátiles **SO-DIMM**. El encapsulamiento DIMM permite disponer de conectores y de chips en ambos lados de la tarjeta (front-side y back-side)
 - En el módulo están interconectados todos los chips de memoria de la tarjeta.
 - Un canal del controlador puede conectar a más de un módulo de memoria: P.ej dos módulos de 4GB cada uno. Si un canal tiene más de un modulo, todos los modulos comparten el mismo BUS DE MEMORIA. Cada módulo implementa direcciones de memoria diferentes.

Rank

- Rank :
 - Es un conjunto o *agrupamiento de chips* dentro de todo el sistema de memoria (todos los módulos DIMM ,no cada módulo DIMM) que tienen en común la señal chip select (CS), compartiendo así el mismo espacio de direcciones.
 - Así organizados, todos los chips del mismo rank pueden responder AL UNISONO al mismo bus de direcciones (filas,columnas). Al activar una señal CS y seleccionar una Fila , se consigue activar todas las columnas de todas las filas de todos los arrays de todos los chips del mismo rank. Este es el objetivo del agrupamiento.
 - Un rank es independiente del resto, con direcciones de memoria *diferentes*, compartiendo el mismo bus de direcciones y bus de chip select. Un rank al ser INDEPENDIENTE puede ser precargado, refrescado, activado, etc al mismo que el resto de ranks.

Chip

- Chip :
 - Es el circuito integrado que contiene el *die* de semiconductor donde están implementadas las celdas de memoria (condensadores) y los interruptores (transistores).
 - El número de pins del chip dependerá del tamaño del dato proporcionado y de la capacidad de almacenamiento de datos.
 - El número de bits "N" del dato proporcionado por el chip a través del buffer i/o, se indica diciendo que el chip es xN: x2,x4,x8,x16,x32
 - Esta formado por *MULTIPLES bancos*, un *buffer i/o*, un *demux* de filas, un *demux* de columnas y la lógica de control.

Bank

- Este término es confuso ya que depende del contexto e incluso hay diversas interpretaciones.
- Bank:
 - Un chip se estructura en bancos independientes.
 - Un banco es un conjunto, *agrupamiento de arrays 2D*.
 - Si cada array envía un bit al buffer i/o, entonces, el número de arrays del banco será el mismo que el número de bits del buffer i/o.
 - Así organizados, todos los arrays del mismo banco pueden responder AL UNISONO al mismo bus de direcciones (filas,columnas).Se selecciona la misma fila y la misma columna para todos los arrays del banco. Cada array del banco proporciona el bit de la celda seleccionada por lo que el número de bits proporcionados por el banco será el número de arrays del banco.
 - Todos los bancos del chip forman parte del mismo buffer i/o del chip.
 - El número de bits " n ", del dato proporcionado por el banco a través del buffer i/o, se indica diciendo que el chip es xn : x2,x4,x8,x16,x32
 - Un banco es independiente del resto, con direcciones de memoria *diferentes*, compartiendo el mismo bus de direcciones y bus de chip select. Un banco al ser INDEPENDIENTE puede ser precargado, refrescado, activado, etc al mismo tiempo que el resto de bancos.

Array

- Array:
 - Son agrupamientos o conjuntos de celdas organizados en filas y columnas.
 - Una dirección de memoria (fila,columna) selecciona una celda del array.

Celda

- Celdas:
 - Una celda de memoria almacena la información de 1 o más (2,4,8,16) bits. Inicialmente, mientras no se especifique lo contrario, almacenará un único bit.

Ejemplo

- Si un sistema tiene una capacidad de memoria principal de 16GB y la estructuramos en 4 módulos cuyos chipsx64 se organizan en 4 ranks con 16 chips/rank, 8 bancos/chip, 16 arrays/banco. Calcular el número de bits/array.
- $2^4 \times 2^{30} \times 2^3 \text{ bits/byte} = 2^2 \text{ (ranks/canal)} \times 2^4 \text{ (chips/rank)} \times 2^3 \text{ (bancos/chip)} \times 2^4 \text{ (array/banco)} \times N \text{ (bits/array)} \rightarrow N = 2^{(4+30+3-2-4-3-4)} = 2^{24} \text{ bits organizados en un array 2D}$



Los 4 módulos al completo se organizan en 4 ranks

- Una posible solución sería 2^{12} filas x 2^{12} columnas.
- El buffer i/o de transferencia de datos al bus de memoria tiene el tamaño x64, es decir, 64 bits.

LECTURA de una palabra de la memoria MP

- Fases:
 - la dirección de memoria proporcionada por la CPU es convertida en dirección física por el circuito MMU
 - El circuito MP debe de descomponer la dirección física de memoria en los códigos:

- RANK-BANK-ROW-COLUMN
- Los códigos están asociados adentro del módulo de memoria un rank específico. Dentro del rank un bank específico. Dentro del bank una fila específica. Dentro de la fila una columna específica.
- Una vez que han sido identificados el rank-bank-row, se PRECARGAN los bit_lines del banco (se polarizan con la tensión media que hay entre un cero lógico y un uno lógico).
- Una vez precargado el banco se ACTIVA (OPEN) la fila: la fila queda abierta cuando los miles de amplificadores sensores de carga detectan los contenidos de los miles de celdas de las filas seleccionadas de todos los arrays del banco. Al conjunto de la misma fila de todos los arrays del banco se denomina **página** (una página está formada por filas). La página esta abierta cuando las salidas de los amplificadores recuperan los valores sensados y activan las line_bit con los datos almacenados.
 - Esta acción comienza con la activación de la señal /RAS y la espera del tiempo tRCD
- Una vez transcurrido el tRCD se selecciona las columnas específicas de todos los arrays del banco (x4,x8,..) y se carga el buffer i/o con el dato seleccionado.
 - Esta acción comienza con la activación de la señal /CAS.

9.3.3. Organización avanzada de memorias DRAM

DRAM asincrona

- En la memoria asíncrona las acciones realizadas dependen del diálogo entre el controlador y la memoria.
- La memoria síncrona comienza y finaliza las acciones en el flanco de subida o bajada del reloj facilitando el diseño del circuito digital electrónico y permitiendo mayores velocidades en el bus.
- Although the RAM is asynchronous, the signals are typically generated by a clocked memory controller, which limits their timing to multiples of the controller's clock cycle.
- [DRAM](#)

SDRAM (Synchronous DRAM)

Referencias

- [SDRAM](#)
- [SDRAM](#)
- [DDR_SDRAM](#)
- [DDR2_SDRAM](#)
- [DDR3_SDRAM](#)
- [jedec standard](#)
- http://www.freescale.com/webapp/sps/site/overview.jsp?code=784_LPBB_DDR
- [memory timing](#)
- http://en.wikipedia.org/wiki/Memory_timings
- http://en.wikipedia.org/wiki/SDRAM_latency
- http://en.wikipedia.org/wiki/CAS_latency

Introducción

- El flanco del reloj es el patrón de comienzo y fin de las operaciones
- DDR (Double Data Rate)

- Permite transferir el bit tanto en el flanco de bajada como de subida del reloj (**doble bombeo**)
- frecuencia del buffer i/o
 - El buffer i/o de la memoria puede ir a frecuencias x2, x4 y x8 respecto de la frecuencia de acceso a la celda.
- **Supercelda:** Ahora una selección (fila,columna) de un array supone no la selección de 1 celda sino la de 2, 4 u 8 CELDAS del array.
- Ver las figuras que representan la transferencia de múltiples celdas al buffer i/o
- Fabricantes: Samsung, Hitachi, NEC, IBM, Siemens.

Table 18. Módulos DDR para PC : características

| | DDR1 | DDR2 | DDR3 |
|----------------------------------|-------------|-------------|-----------------|
| bit i/o: celdas/ciclo_bus | x2 | x4 | x8 |
| frecuencia bus | f | 2f | 4f |
| burst mínimo | 2 | 4 | 8 |
| pines DIMM | 184 | 240 | 240 |
| pines SO-DIMM | 200 | 200 | 144/200/ 204 |
| alimentación(v) | 2.5 | 1.8 | 1.5 |

- Dual In-line Memory Module (DIMM)
- Small Outline Dual In-line Memory Module (SO-DIMM)
- BW (bits/s) = BF(ciclos/s)*CW(bits/channel)*TC(transferencias/ciclo)
 - BF: Frecuencia del bus del sistema (próximo a 1GHz en el año 2000)
 - CW: número de bits del data bus del canal. Típicamente 64 bits (año 2000)
 - TC: en un ciclo del reloj del bus del sistema el número de transferencias. Típicamente 1 (flanco de subida) o 2 (flancos de subida y bajada).
 - BW (bits/s) = frecuencia efectiva*anchura bus datos= $400\text{MHz} \times 2 \times 64 = 51200 \times 10 \sim 6 \text{ bits/s} = 51.2\text{Gbps}$
= 6400 MBps ← sistema decimal (habitual)

Ejemplo DDR3-800 / PC3-6400 5-5-5

- módulo de memoria DDR3-800 ó PC3-6400
 - timing 5-5-5
 - 800MHz es la frecuencia efectiva del bus de datos → 800MT/s
 - 6400 MB/s es el ancho de banda
 - DDR → La frecuencia del bus de memoria es la mitad de la frecuencia efectiva = $800/2 = 400\text{MHz}$.
Equivale a un ciclo de reloj de $1/400\text{MHz} = 2.5\text{ns}$
 - 5-5-5: son los ciclos de reloj, a la frecuencia real del bus de 400MHz, de los parámetros timing tCL-tRCD-tRP

Table 19. DDR3

| Standard name | Memory clock(MHz) | Cycle time(ns) | I/O bus clock(MHz) | Data rate(MT/s) | Module name | Peak transfer rate(MB/s) | Timings(CL-tRCD-tRP) | CAS latency(ns) |
|---------------|-------------------|----------------|--------------------|-----------------|-------------|--------------------------|----------------------|-----------------|
| DDR3-800D | 100 | 10 | 400 | 800 | PC3-6400 | 6400 | 5-5-5 | 12½ |
| DDR3-800E | | | | | | | 6-6-6 | 15 |

El 4º dígito es tRAS (mínimo retardo entre la activación y la precarga) no ha sido proporcionado. La cuarta columna proporciona tCL en nanosegundos.

- Parámetros:

- Memory clock: 100MHz: frecuencia de acceso a las palabras. Transferencia celda → buffer i/o
- Cycle time: 10ns: en esta tabla se refiere al período del memory clock y no tiene el significado de la definición de ciclo de memoria
- I/O bus clock: 400MHz: reloj del bus de memoria cuyos flancos(positivo,negativo) sincronizan las transferencias de las palabras.
 - ciclo de bus = $1/400\text{Mhz} = 2.5\text{ns}$ = este es el factor de tiempo de los retardos o latencias tCL,tRCD, etc
- Data rate: 800MT/s → Las transferencias se realizan a la frecuencia efectiva.
- Peak transfer rate: ancho de banda BW:6400MB/s
- timings: número de ciclos del reloj i/o bus clock de duración de los eventos:5-5-5-12½
 - tCL = 5 ciclos de reloj = $5 \times 2.5 = 12.5\text{ns}$
 - tRCD = 5 ciclos de reloj = $5 \times 2.5 = 12.5\text{ns}$
 - tRP = 5 ciclos de reloj = $5 \times 2.5 = 12.5\text{ns}$
 - tRAS = no se ha proporcionado

Ejemplo PC3-22400 11-14-14-35

- Dominator® Platinum with Corsair Link Connector — 1.65V 16GB Dual Channel DDR3 Memory Kit ([CMD16GX3M4A2800C11](#)):
- Memory Type: DDR3
- Speed Rating: PC3-22400 (2800MHz)
- Tested Latency: 11-14-14-35
- Our Price: 80€
- 16GB Kit (4 x 4GB)
- Dual Channel
- Características deducidas:
 - Ancho de banda de pico = 22400MB/s
 - Data rate (1data=8Bytes) = 2800MT/s
 - I/O bus effective clock = 2800MHz. I/O hace referencia al bus del buffer i/o de la memoria.
 - I/O bus clock = $2800\text{MHz} / 2 = 1400\text{MHz}$
 - I/O bus cycle time = $1/1400\text{MHz} = 710\text{ps}$
 - Latencies
 - tCL = 11 ciclos = $11 \times 710\text{ps} = 7.8\text{ns}$

- tRCD = 14 ciclos = $14 \times 710\text{ps} = 10\text{ns}$
- tRP = 14 ciclos = $14 \times 710\text{ps} = 10\text{ns}$
- tRAS = 35 ciclos = $35 \times 710\text{ps} = 24.8\text{ns}$
- Mejora PC3-22400 vs PC3-6400
 - Mejora del I/O bus cycle time = 710ps frente a 2.5ns = una reducción de $1.79\text{ns} = 1.79/2.5 = 71\%$

Diferencia entre PC2-6400 y PC3-6400

- No ha diferencias en cuanto a latencias ya que un 5-5-5 en los dos casos se refiere a una frecuencia del bus de memoria de 400MHz.
- Hay diferencias en cuanto a pines, tensión de alimentación, etc

Anchos de banda standard

- Módulos DDR1 SDRAM: PC-3200/PC-2700/PC-2100/PC-1600
- Módulos DDR2 SDRAM: PC2-6400/PC2-5300/PC2-4200/PC2-3200
- Módulos DDR3 SDRAM:
 - PC3-22400/PC3-21300/PC3-19200/PC3-17066/PC3-15000/PC3-12800/PC3-10600/PC3-8500/PC3-6400
 - DDR3-2800/DDR3-2666/DDR3-2400/DDR3-2133/DDR3-1866/DDR3-1600/DDR3-1325/DDR3-1065/DDR3-800/

aniyosgi diquenvsvi gega → miltar

diquenvsvi gega → go home

Capacidad

Registered/Buffered Memory

- http://en.wikipedia.org/wiki/Registered_memory
- Registered: RDIMM: Entre el controlador de memoria y el módulo de memoria hay un registro que memoriza la info de las líneas de control. Se manda el comando de control previamente a la transferencia, añadiendo un ciclo extra de bus. De esta forma se eliminan las líneas de control para la transferencia de comandos al controlador y así se disminuye la carga del bus de memoria del controlador de memoria y se consigue conectar más módulos al canal del controlador aumentando la capacidad de memoria.
- Unbuffered: UDIMM: No se latchea la info de las líneas de control.
- fully buffered:
 - Se registra tanto la info de las señales de control como de las señales de datos y direcciones con una reducción considerable de la carga de todos los buses del canal del controlador de memoria.
 - Los datos se transfieren en serie en lugar de en paralelo reduciendo el número de líneas y por lo tanto aumentando el número de módulos de memoria conectados al canal.

Bank Switching

- [Bank Switching](#)
 - En arquitecturas limitadas de 8 y 16 bits se utiliza la técnica *memory banking* para aumentar la capacidad de memoria.
 - En lugar de incrementar anchura del bus de direcciones incrementando el tamaño de palabra de la CPU y el bus de la placa base, se añaden más dispositivos de memoria direccionables mediante el

el mismo bus y un nuevo registro que selecciona uno de los dispositivos de memoria (Bank). No confundir con los bancos de los chips de memoria ni con los bancos de registros.

- Bank switching significa cambiar de banco de memoria.

9.3.4. Imagenes

- [Imagenes](#)

9.4. Memoria Cache

9.4.1. Bibliografia

- Libro William Stalling
 - Capítulo 4.

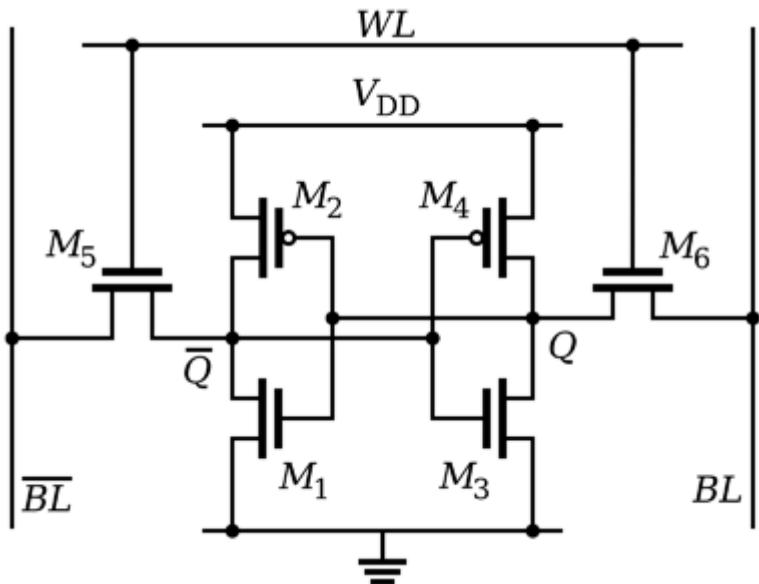
9.4.2. Introducción

- Objetivo
 - Principio de Localidad: Espacial y Temporal
- Ejercicio: Tiempo de acceso (probabilidad fallo ó exito)
- Tecnología: 6T
- Estructura
 - Controladora: función
- Espacios de direcciones
 - memoria principal
 - memoria cache
- Funciones de correspondencia entre espacios de direcciones:
 - Mapeo Directo
 - Asociación total
 - Asociación por conjuntos

9.4.3. Principios Basicos

Tecnología

- Cell: SRAM-6T



Funcionalidad

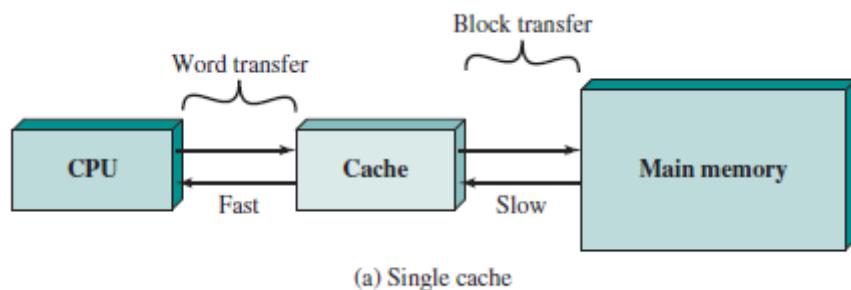
- Cuello de Botella von Neumann
- Memoria Cache:
 - Copia de una región o bloque de la memoria principal
- Principio de Localidad
 - Espacial: bucles, subrutinas, arrays
 - Temporal: histórico

Jerarquía

- Niveles de Cache
 - Level L1: Interna a la CPU : SRAM : memorias separadas para instrucciones y memoria para datos
 - Level L2: Externa/Interna a la CPU:
 - Level L3: Externa a la CPU

Interconexión

- En serie CPU → L1 → L2 → L3 → SDRAM
- CPU → L : transferencia de Palabras
- L → SDRAM: bloques



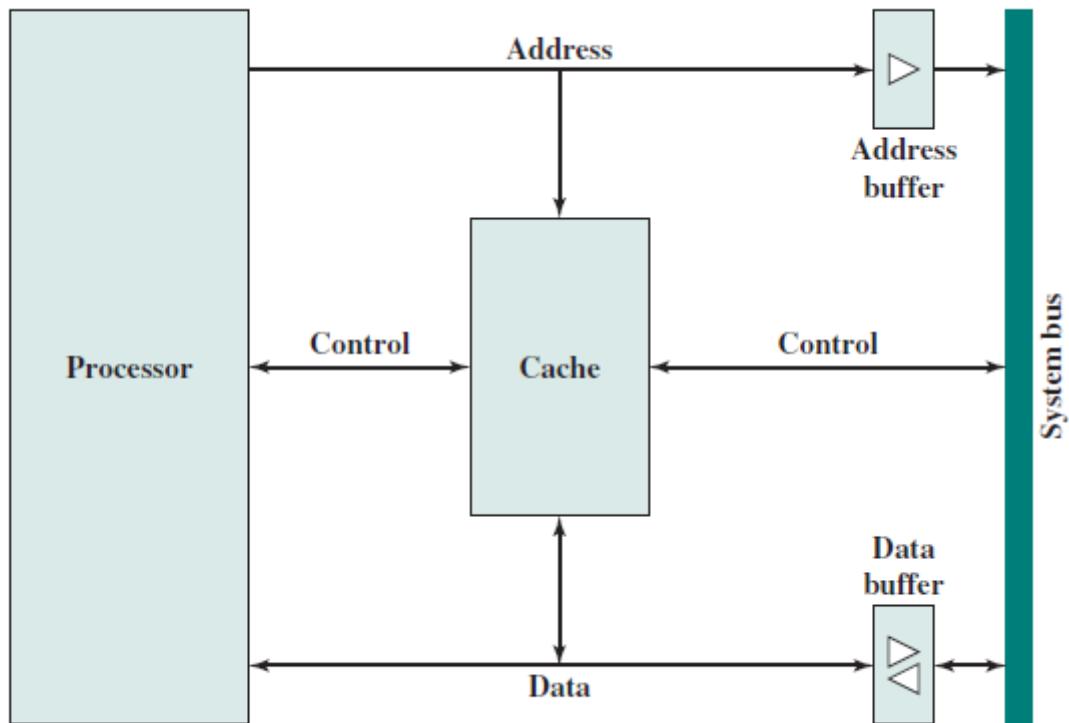


Figure 4.6 Typical Cache Organization

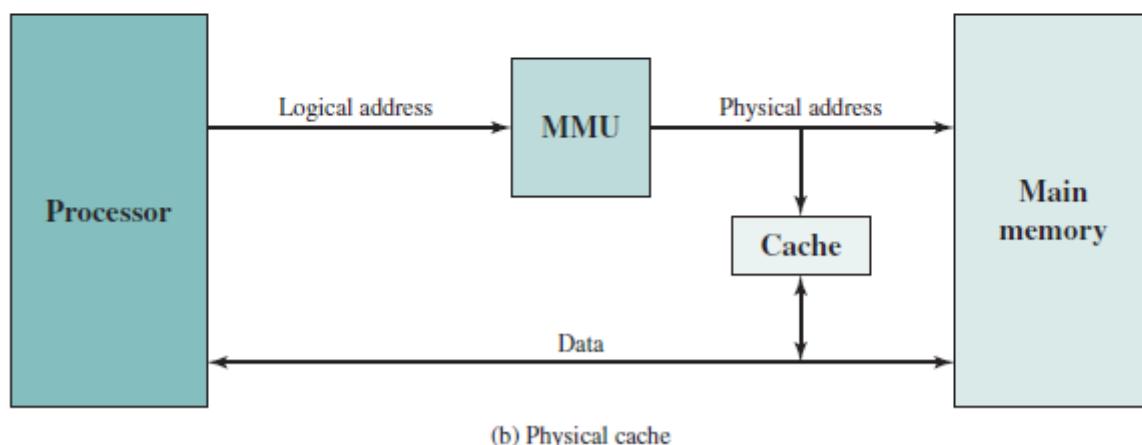
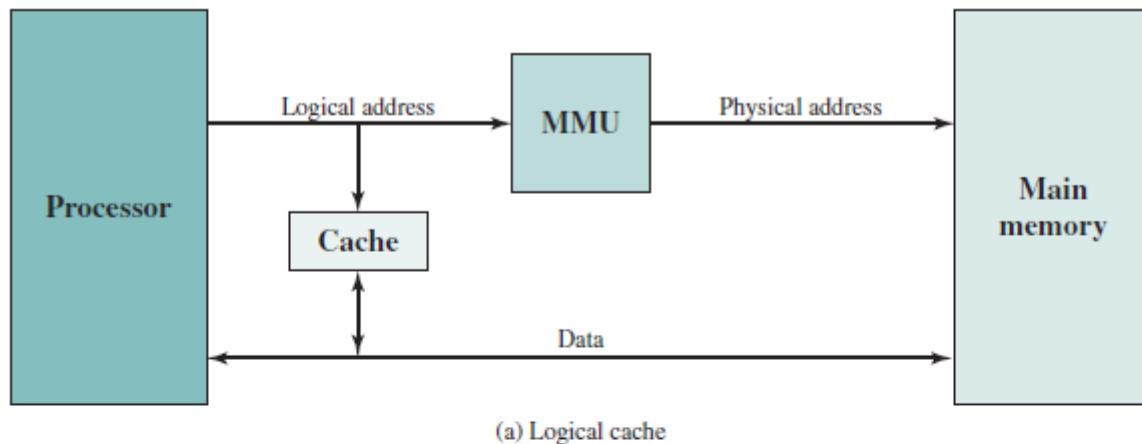


Figure 4.7 Logical and Physical Caches

Acierto-Fallo

- Ejemplo 4.1

Estructura Cache/Principal

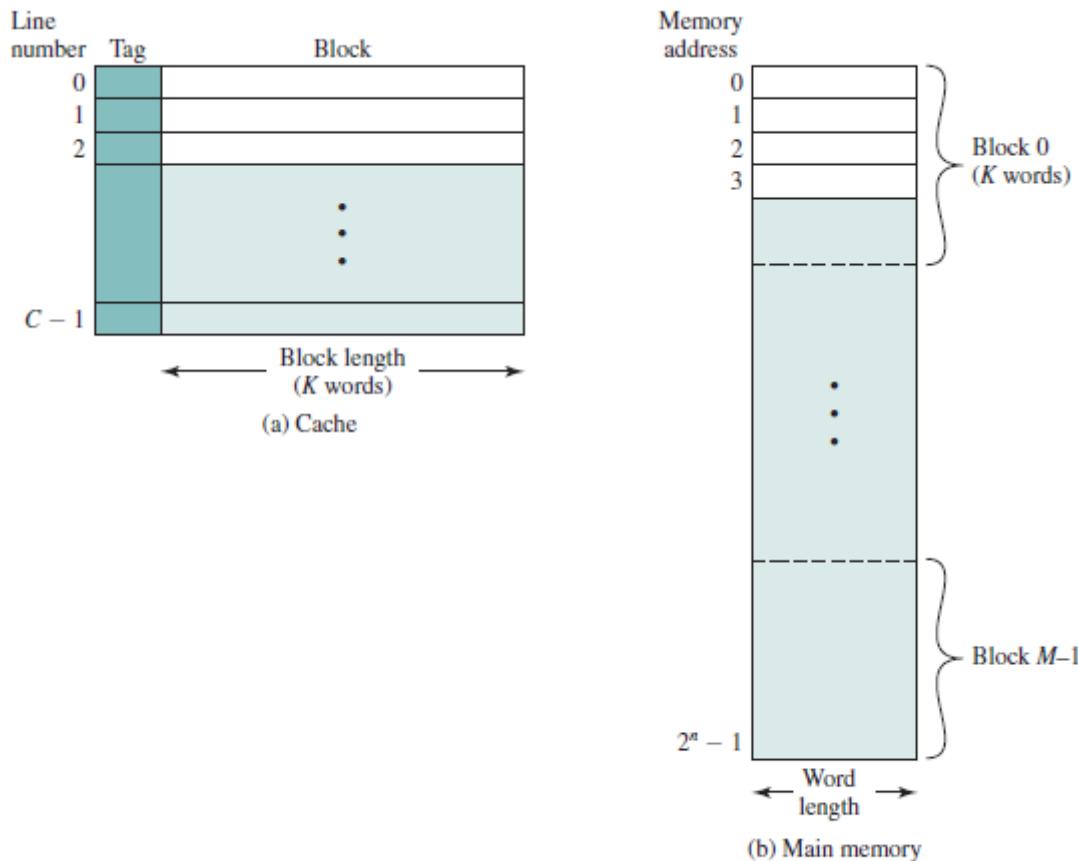


Figure 4.4 Cache/Main Memory Structure

- Cache: bytes → palabras → líneas
- MP: bytes → palabras → bloques

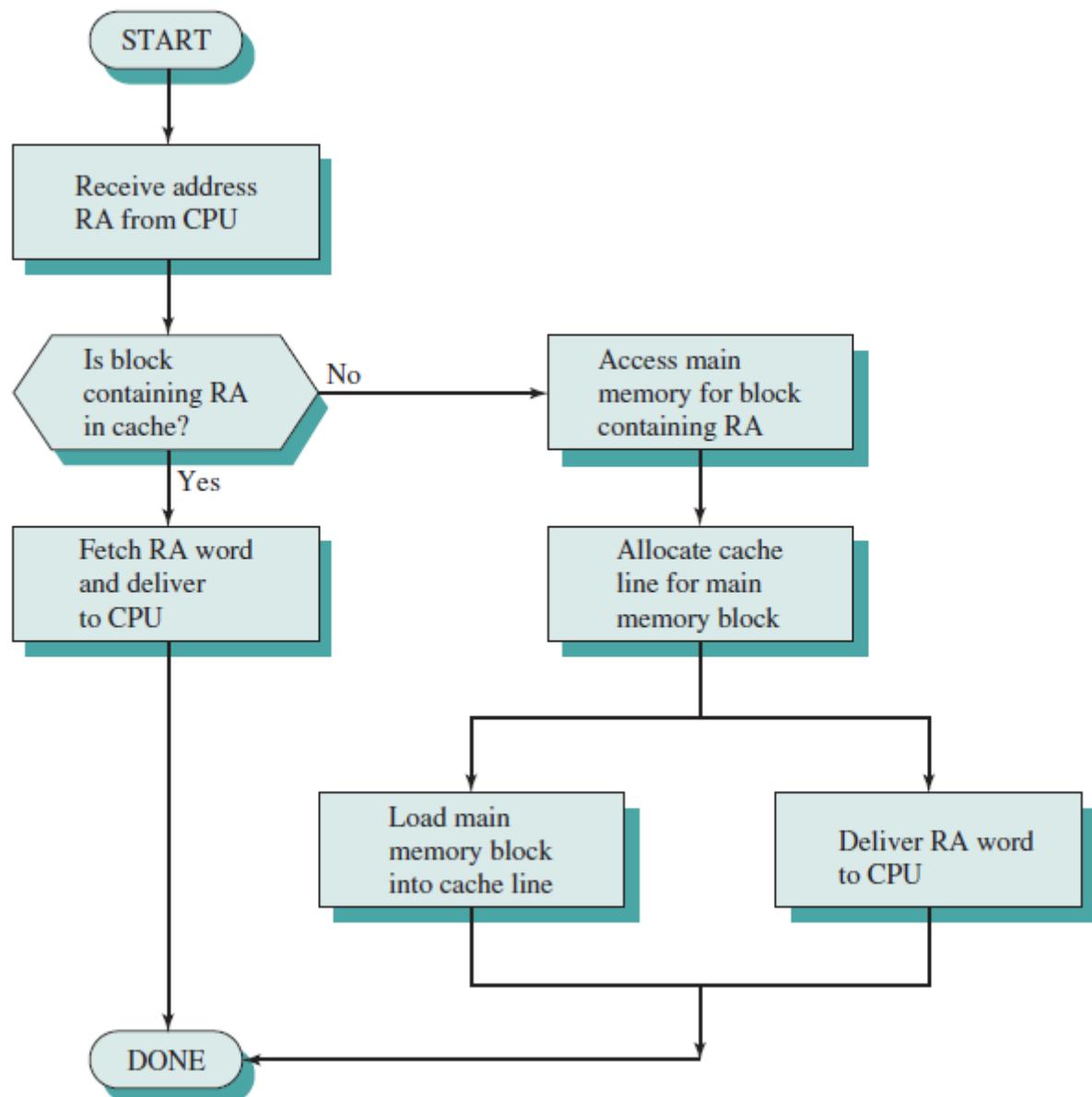
Principal

- Dirección: n bits : bus de direcciones del sistema → Capacidad= 2^n palabras → OJO: no bytes
- Direccionable: palabras : bus de datos del sistema
- Organizada en Bloques de K palabras
- Número de bloques : Capacidad en palabras / K

Cache

- Dirección: N° de Línea y Palabra
- Direccionable: Palabra
- Capacidad: C Líneas
- Organizada en Líneas de K palabras y 1 etiqueta
- Etiqueta: Asociación Línea con Bloque de la Memoria Principal

Operacion de Lectura



- Organigrama de operaciones
- Controladora de la Caché

9.4.4. Elementos de Diseño de la Cache

- Elementos:
 - Tamaño de la Caché, Función de Correspondencia, Algoritmos de Sustitución, Política de Escritura, Tamaño de Línea, Número de Cachés

Tamaño

- Contradicción: Cuanto más grande más lenta y más probabilidades de acierto.
- L1: KB
- L2: MB
- L3: MB

Función de Correspondencia

- Directa, Totalmente Asociativa, Asociativa en Grupo

Ejemplo

- Libro W.Stalling. Capítulo 4. Ejemplo 4.2
- Ejemplo para los 3 casos:
 - m : caché de capacidad 64 KB = $4 * 2^{14}$ bytes
 - MP :
 - word size : 1 byte
 - palabras/bloque = 4.
 - capacidad = 16MB = 2^{24} bytes = $4 * 2^{22}$ = 4M bloques
- La capacidad de la caché m son 2^{14} líneas = 16K líneas
- La relación de capacidad caché/MP es $16K / 4M = 1 / (2^8)$

Directa

- Estructura de direcciones
 - Memoria principal : bloques de palabras
 - Memoria cache : líneas de palabras
- Función de correspondencia
 - determinista - ningún grado de libertad en la elección de la línea correspondiente a un bloque determinado.

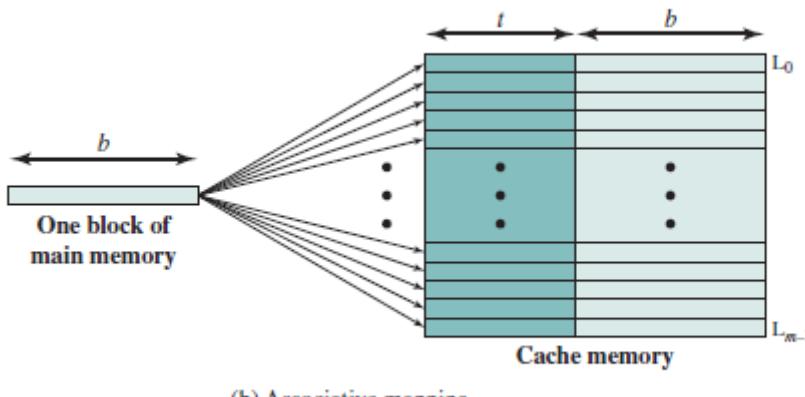
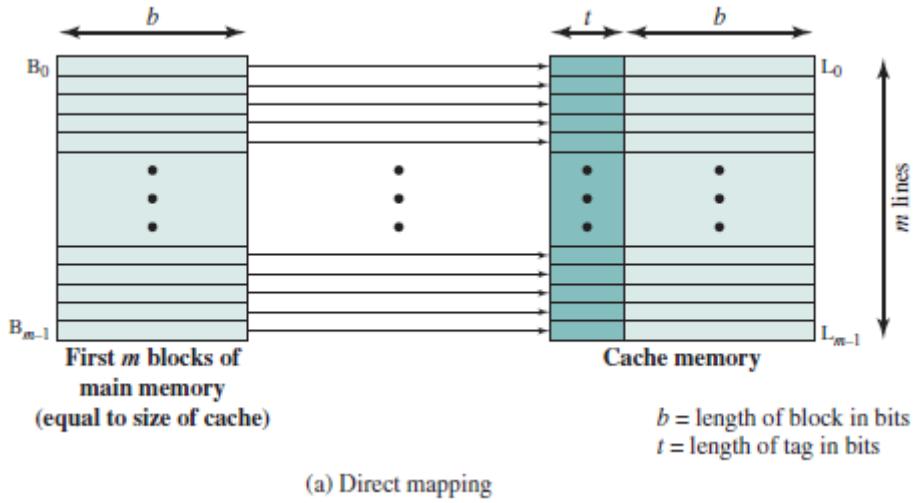


Figure 4.8 Mapping from Main Memory to Cache: Direct and Associative

- i : número de línea de caché
- j : número de bloque de la memoria principal
- m : número de líneas en la caché
- Función de correspondencia
 - $i = j \text{ módulo } m$
- Organización de la caché
 - Caché + Controladora

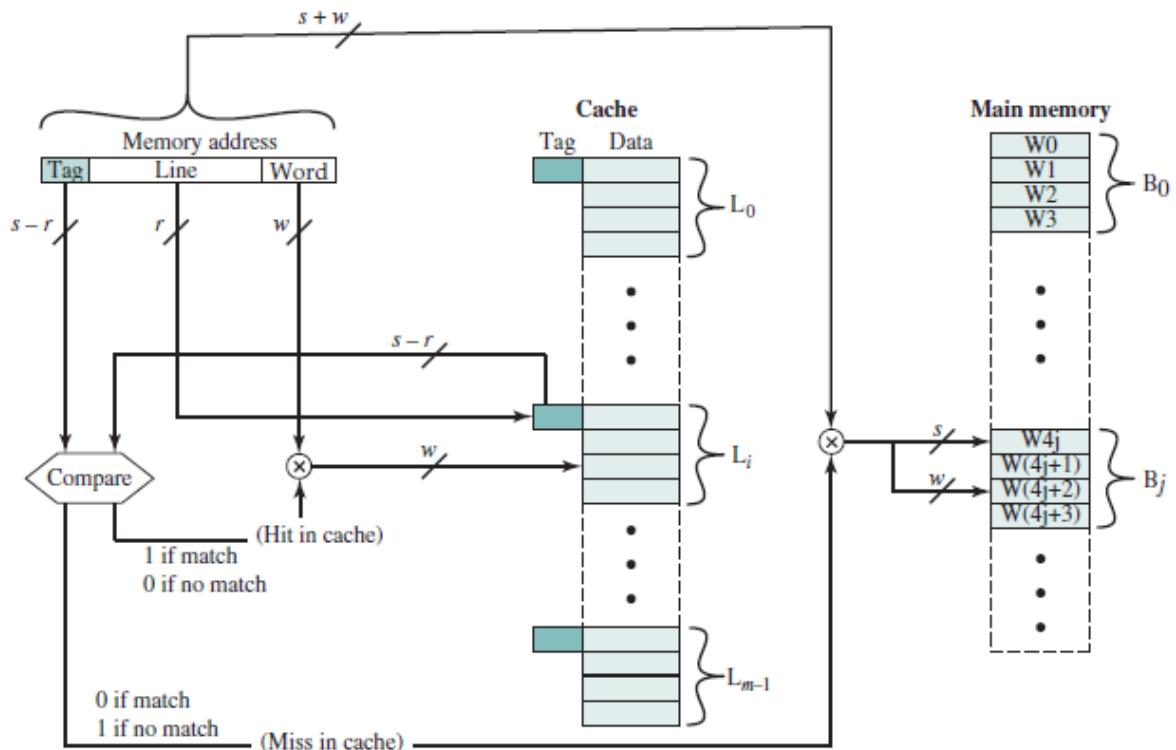


Figure 4.9 Direct-Mapping Cache Organization

- Formato de direcciones
 - Dirección física de la memoria principal: bloque-palabra
 - Dirección física de la memoria cache: tag-línea-palabra
- Operación de búsqueda de una palabra en la memoria caché.
 - Determinar los campos de etiqueta, línea y palabra del formato de direcciones de la memoria caché.
 - La palabra pudiera estar en únicamente en la línea asignada, por lo que es necesario comparar únicamente la etiqueta de dicha línea con la etiqueta del formato de direcciones.
- Ejemplo 4.2
 - Ejemplo 4.2a

Totalmente Asociativa

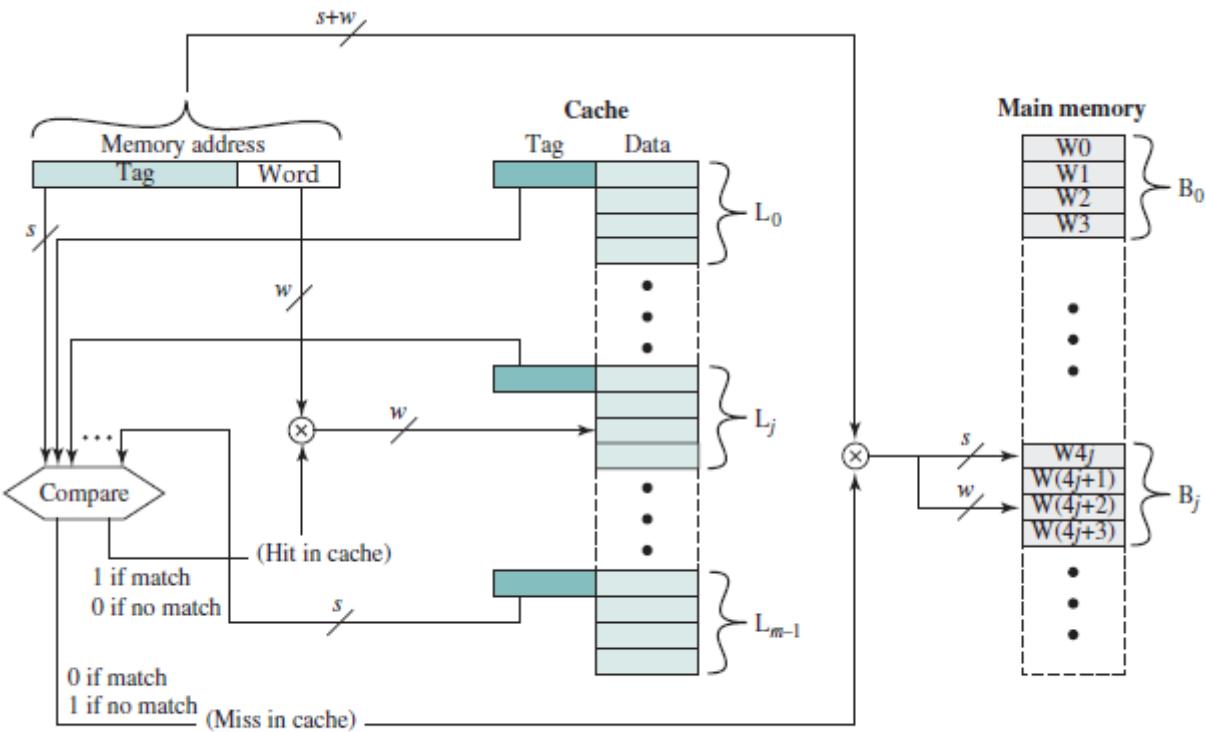


Figure 4.11 Fully Associative Cache Organization

- La dirección de memoria tiene únicamente dos campos
 - s bits= etiqueta: orden del bloque de MP : desde 1 hasta 4M.
 - w bits= orden de la palabra dentro del bloque: desde 1 hasta 4.
- controlador de caché
 - todas las etiquetas de las líneas de caché son SIMULTANEAMENTE comparadas con la etiqueta de la palabra referenciada.
 - si éxito, s apunta a la línea que contiene la palabra referenciada y w apunta a la palabra referenciada.
 - si fracaso, s apunta al bloque de la MP que contiene la palabra referenciada y w apunta a la palabra referenciada.
- función de correspondencia
 - NO HAY NORMA → CADA BLOQUE DE MP PUEDE SER ASIGNADA A CUALQUIER LINEA DE LA CACHE
 - LIBRE: un bloque de MP no tiene asignada ninguna línea específica y el controlador cache puede seleccionar qué línea será asignada a dicho bloque.
 - Formato de direcciones
- Dirección física de la memoria principal: bloque-palabra
- Dirección física de la memoria cache: tag-palabra
 - Operación de búsqueda de una palabra en la memoria caché.
- Determinar los campos de etiqueta y palabra del formato de direcciones de la memoria caché.
- La palabra puede estar en cualquier línea, por lo que es necesario comparar las etiquetas de todas las líneas
 - Diferencia con el mapeo directo:
- el campo de etiqueta tiene s bits $>>$ $s-r$ bits

- la comparación es entre todas las etiquetas → hardware complejo → coste

Asociativa por conjuntos

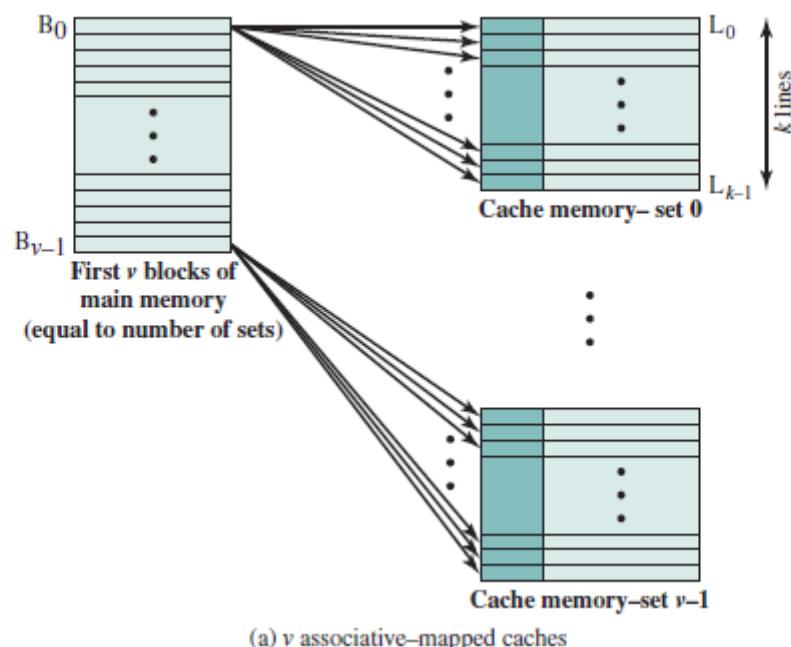
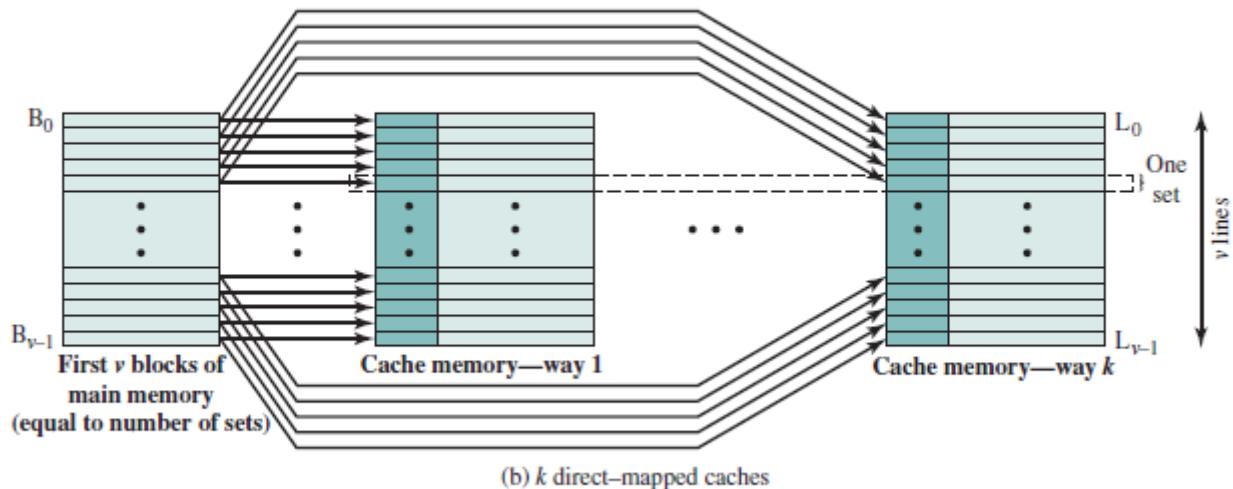
(a) v associative-mapped caches(b) k direct-mapped caches

Figure 4.13 Mapping from Main Memory to Cache: k -Way Set Associative

- compromiso entre el rigor de la correspondencia directa y la flexibilidad de la correspondencia totalmente asociativa.
 - La dirección de memoria tiene 3 campos: TAG-SET-WORD → (s-d)/d/w
 - w bits : orden de la palabra. Con 2^w palabras formo un bloque
 - d bits :
 - CONJUNTO de bloques o SET de bloques o SUPERbloque o SUPERlínea.
 - $d < l$: dividimos la cache en v superbloques.
 - Con $v=2^d$ superbloques de k bloques formo la memoria cache.
 - Al número k de líneas de cada superbloque se le denomina VIA (WAY)
 - estructura de la MP: queda dividida en bloques y superbloques.
 - 2^s es el número de bloques que si los agrupo en sets de k bloques tendré en MP 2^t agrupamientos de 2^d sets cada uno → $2^s * 2^w = 2^t * 2^d * 2^w \rightarrow 2^s = 2^t * 2^d \rightarrow N^o$ de bloques = N^o de etiquetas * N^o sets

- s-d bits: nº de bits de la etiqueta : ¿qué representa $2^{(s-d)}$?
- función de correspondencia
 - $i = j$ modulo $v \rightarrow$ NORMA SEMIRIGIDA: CADA BLOQUE TIENE ASIGNADO UN CONJUNTO ESPECIFICO DE LINEAS PERO NO TIENE ASIGNADA LA LINEA DENTRO DEL CONJUNTO
 - donde v es el número de superbloques, j es el número de bloque en MP e i el número de superbloque en la caché.
 - dentro del superbloque i hay flexibilidad para asignarle una de las líneas o vías dentro del superbloque.
- número de vías
 - si el número de vías fuese 1 no habría ninguna libertad de asignación y estaríamos en el caso de correspondencia directa
 - si el número de vías fuese 2 habría poco de libertad ya que habría que elegir una línea a sustituir de dos líneas posibles.
 - si el número de vías fuese la capacidad de la caché el grado de libertad sería máximo, a sustituir una línea de m posibles.

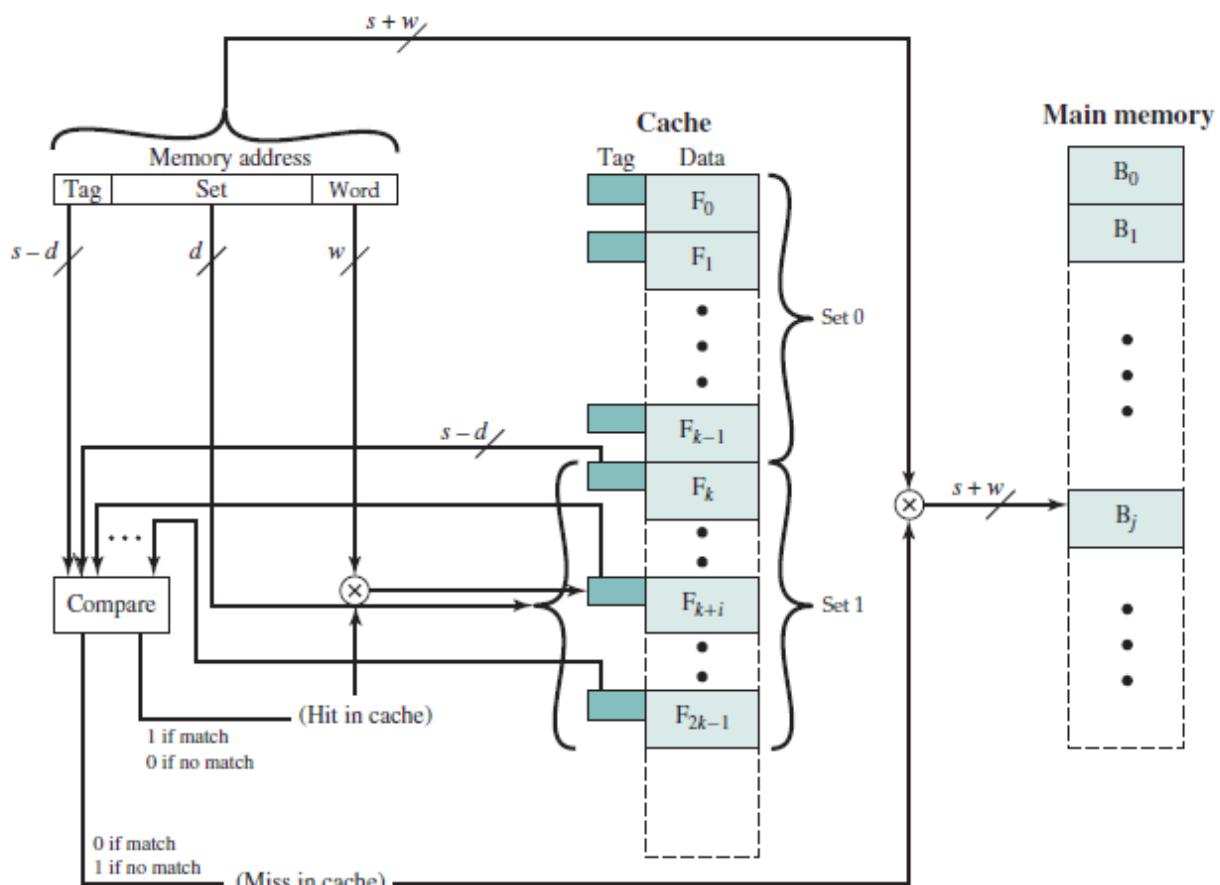


Figure 4.14 K-Way Set Associative Cache Organization

- Formato de direcciones
 - Dirección física de la memoria principal: bloque-palabra
 - Dirección física de la memoria cache: tag-set-palabra
- Operación de búsqueda de una palabra en la memoria caché.
 - Determinar los campos de etiqueta y set del formato de direcciones de la memoria caché.

- La palabra puede estar en cualquier línea pero únicamente del set asignado, por lo que es necesario comparar las etiquetas únicamente de las líneas de dicho set.

Comparativa de los 3 tipos de funciones

- En la función de correspondencia de mapeo directo no hay ninguna libertad a la hora de seleccionar la línea de la cache, esta viene determinada por la función de correspondencia. En el caso de función asociativa total la libertad es total pudiendo elegir la línea a ocupar siguiendo criterios estadísticos, etc. En el caso de función asociativa por conjuntos no hay ninguna libertad en la asignación del conjunto de líneas determinado por la función pero sí en la elección de la línea dentro del conjunto asignado por la función.
- Los dos primeros casos, mapeo directo y asociativa total, son los casos extremos de la asociativa por conjuntos:
 - Asociativa por conjuntos con un set de una línea sería el caso de mapeo directo
 - Asociativa por conjuntos con un set de todas las líneas de la caché sería el caso de asociativa total

9.5. Memoria Virtual

9.5.1. Bibliografía

- Computer Organization and Architecture: Designing for Performance. William Stallings, Cap8 Sistemas Operativos: 8.3 Gestión de la Memoria
 - Sistema Operativo: Gestión de la Memoria
- Computer Systems A Programmer's Perspective, Randal E. Bryant. Capítulo 9. Virtual Memory

9.5.2. Sistemas Operativos: Gestión de la Memoria

Sistemas Multiproceso

- En un sistema multitarea hay más de un proceso ejecutándose y residente en la memoria principal.
- La memoria principal es un recurso compartido por todos los procesos. En un entorno multiproceso, es necesario gestionar el recurso compartido para asignar regiones de memoria física a cada proceso, para proteger espacios de memoria entre los distintos procesos, etc
- Históricamente la memoria principal era muy limitada frente al tamaño de los programas.
- Técnicas de gestión de la memoria en sistemas operativos: swapping, particionamiento, memoria virtual, segmentación, paginación. .

Gestión de la Memoria Física

Memoria Principal

- La memoria principal es la memoria física externa a la CPU e implementada en tecnología SDRAM con una capacidad típica en el año 2010 de 4GB.

Swapping

- Significa intercambio.
- Los programas se almacenan en el disco duro como módulos ejecutables.
- Los módulos ejecutables deben cargarse en memoria para ser ejecutados, convirtiéndose en procesos.
- La memoria es *limitada* por lo que no puede almacenar todos los procesos requeridos por el usuario.
- Una solución es que los procesos utilicen tanto la memoria principal como el disco duro. En memoria residen los procesos que son ejecutados en un momento dado y cuando uno de dichos procesos no

requiere de la CPU (espera a un evento i/o)(no está en estado *ready*) se intercambia con el disco duro por un proceso que si requiere de la CPU (está en estado *ready*). La transferencia del proceso hacia la memoria principal se denomina swap-in y la transferencia hacia el disco duro swap-out.

- Se **intercambia todo** el proceso.
- Hay un intercambio de procesos completos entre la memoria y el disco duro. Este es el concepto de swap para algunos sistemas operativos como Solaris y el que se toma por definición. En Linux tiene otro significado.
- Al trasladar un proceso de memoria al disco duro se genera un **hueco en la memoria**. La existencia de múltiples huecos dispersados por la memoria se le denomina *fragmentación externa*.
- El inconveniente es que el swapping requiere de una operación i/o con el disco duro ralentizando el rendimiento de la computadora.

Particionamiento (Fragmentación)

- **La clave:**
 - Sin memoria virtual el código de los procesos en la memoria física ha de ser **contiguo**.
 - Si no utilizamos memoria virtual el *particionamiento dinámico* (el proceso ocupa justo la región de memoria que necesita) produce fragmentación externa al eliminar particiones y el *particionamiento fijo* (el proceso ocupa menos de la región que tiene reservada) produce fragmentación interna
- **La Solución** para que un proceso pueda ocupar particiones **NO CONTIGUAS** en la memoria FISICA : memoria VIRTUAL.
 - Al poder asignar particiones pequeñas fijas no contiguas el fraccionamiento externo desaparece y el interno se reduce al máximo (inferior al tamaño de la partición)
- El **particionamiento** es una técnica para asignar memoria principal a los distintos procesos que están siendo ejecutados concurrentemente en la computadora.
- La memoria se divide en múltiples regiones o *particiones* de tamaño no uniforme.
- A un proceso se le asigna una partición de *igual o mayor tamaño*.
- Se utiliza en sistemas multitarea donde la memoria principal es compartida por múltiples procesos. De esta manera se puede *gestionar el compartir* la memoria entre los diferentes procesos, protección, permisos, superusuario, etc
- Cuando el proceso no está ready se realiza un swapping con el disco duro.
- Dos alternativas : la estructura de las particiones puede ser fijo o variable en el tiempo. En los dos casos un proceso requiere una partición, es decir, una región de posiciones de memoria **contiguas**.
- **Particionamiento FIJO :**
 - La memoria principal se parte en regiones cuyo tamaño no varía durante la ejecución de los procesos.
 - Particionamiento fijo con regiones de igual tamaño o particionamiento fijo con regiones de diferente tamaño.
 - A los procesos se les asigna una partición de tamaño mayor que el requerido. Esto produce **fragmentación interna**, ya que una zona de la partición no es aprovechada por ningún proceso.
- **Particionamiento variable o DINAMICO:**
 - A cada proceso se le asigna justo la memoria que necesita. No hay fragmentación interna.
 - El tamaño de las particiones cambia dinámicamente según se intercambian procesos con el disco duro adaptándose al tamaño de estos.
 - La **fragmentación externa** es considerable. Se podría reducir compactando los huecos dispersos, para lo cual es necesario mover o reubicar los procesos en la memoria principal. La reubicación de procesos significa resolver todas las direcciones físicas nuevamente en tiempo de ejecución → puede resultar inviable el tiempo requerido.

- Hay publicidad de programas que defragmentan la memoria principal lo cual no es posible ya que únicamente el S.O. conoce las direcciones físicas de un proceso.

Alternativa

- La solución a la fragmentación debido a la técnica del particionamiento de la memoria física es la técnica de memoria virtual bien segmentada o bien paginada o ambas.

Gestión mediante la Memoria Virtual

Alternativa

- Los problemas de gestionar la memoria de los procesos asignando a los procesos *directamente* un espacio de direcciones físico se resuelven mediante el mecanismo de la *memoria virtual*.

Espacio de direcciones virtual

- El programador, el compilador, el linker y los procesos no operan con direcciones físicas.
- El programador referencia la memoria en el módulo fuente con símbolos (etiquetas, variables, nombres de funciones, etc)
- El compilador y el linker traducen los símbolos a direcciones de una memoria imaginaria lineal y contigua denominada memoria virtual.
- Esta independencia de las direcciones físicas simplifica enormemente la gestión de la memoria.

Ejemplo: programa exit

- Desensamblado del módulo objeto ejecutable residente en el disco
 - `objdump -S exit`

```
exit:      file format elf64-x86-64
```

```
Disassembly of section .text:
```

```
0000000000400078 <.text>:  
400078: 48 c7 c0 3c 00 00 00    mov    $0x3c,%rax  
40007f: 48 c7 c7 ff 00 00 00    mov    $0xff,%rdi  
400086: 0f 05                  syscall
```

- Las direcciones 0x400078,... son direcciones del espacio virtual. El espacio de direcciones virtual es *lineal*, contiguo y único.
- La dirección virtual *0x0000000000400078* comprende 16 dígitos hexadecimales, es decir, el espacio de direcciones virtual del proceso *exit* tiene capacidad para 2^{64} Bytes.

Memoria: Recurso compartido

- Cada proceso tiene su propio espacio de direcciones virtual.
- Todos los procesos han de compartir la misma memoria física.
- Todos los espacios virtuales han de ser traducidos al mismo espacio físico.

Traducción virtual → físico

- Cada vez que la CPU acceda a memoria para capturar instrucciones y datos o escribir resultados, será necesario *traducir* la dirección lógica en una dirección física. Es decir, las direcciones
- Esta traducción la realiza la unidad hardware *Management Memory Unit (MMU)*.

Direccionamiento lógico

- En el intercambio de procesos entre la memoria y el disco duro, las direcciones físicas de memoria donde son cargados los datos y las instrucciones pueden cambiar. Debido a ello no es factible un modelo de direccionamiento que utilice direcciones físicas absolutas.
- Las direcciones del proceso se expresan de forma **relativa** respecto de una **dirección base**. Al par dirección base y offset se le denomina *dirección lógica*.
- Esta dirección lógica es una dirección virtual, no física.

Dos tipos: Segmentación y Paginación

- El espacio de memoria virtual se puede gestionar utilizando dos mecanismos o la combinación de ellos:
 - **Segmentación**
 - La memoria virtual de un proceso se divide en unidades lógicas indivisibles denominadas segmentos
 - **Paginación**
 - La memoria virtual de un proceso y la memoria física de la computadora se dividen en unidades denominadas páginas (lógicas en la memoria virtual y físicas en la memoria principal).

9.5.3. Memoria Virtual Segmentada

Interpretación de la segmentación

- La segmentación se puede aplicar tanto al espacio de direcciones físico como al espacio de direcciones virtual.
 - a. Segmentación del espacio de direcciones virtual
 - División de un programa (proceso) en unidades lógicas: código, variables inicializados, variables sin inicializar, datos read only, etc. División de la memoria virtual de un proceso en áreas de **memoria contigua** y cuyo tamaño puede variar dinámicamente. Los segmentos lógicos no se pueden dividir.
 - Facilita el trabajo del compilador, linker, sharing, etc
 - El espacio total de la memoria virtual formado por todos los procesos estaría formado por la dirección base segmento y el desplazamiento (offset) del registro contador de programa.
 - Se ha utilizado memoria virtual segmentada en las CPU: 80286, 80386, 80486 y Pentium
 - b. Segmentación del espacio de direcciones físico.
 - Se utilizó en la arquitectura intel 8086 para pasar de un bus de direcciones de 16 bits a 20 bits manteniendo el tamaño de los registros con 16 bits.
 - Incrementar el espacio de direcciones físicas añadiendo un registro de segmento y sin incrementar el tamaño del registro contador de programa. Por ejemplo un microprocesador Intel de 16 bits sin segmentación tiene limitado el espacio físico a $2^{16} = 64KB$. Con el mismo micro y un registro adicional de segmento RS de 16 bits podemos concatenar el registro RS con el contador de programa PC formando direcciones físicas de 32 bits con lo que tendríamos un espacio de direcciones físicas de $2^{32} = 4GB$

Secciones

- cada módulo objeto reubicable está estructurado en secciones
- una sección es una división lógica, no física.
- la estructura en secciones se define en el módulo fuente
- Secciones principales
 - text : instrucciones
 - data : variables inicializadas
 - rodata: variables readonly
 - bss: variables sin inicializar
- `readelf -S maximum`

There are 16 section headers, starting at offset 0x448:

Section Headers:

| [Nr] | Name | Type | Address | Offset | Size | EntSize | Flags | Link | Info | Align |
|------|-------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| [0] | | NULL | 0000000000000000 | 00000000 | 0000000000000000 | 0000000000000000 | | 0 | 0 | 0 |
| | 0000000000000000 | 0000000000000000 | | 0000000000000000 | 0000000000000000 | 0000000000000000 | | 0 | 0 | 0 |
| [1] | .interp | PROGBITS | 0000000000400158 | 00000158 | 0000000000000001 | 0000000000000000 | A | 0 | 0 | 1 |
| | 0000000000000001c | 0000000000000000 | | 0000000000000000 | 0000000000000000 | 0000000000000000 | | A | 0 | 0 |
| [2] | .hash | HASH | 0000000000400178 | 00000178 | 000000000000000c | 0000000000000004 | A | 3 | 0 | 8 |
| | 000000000000000c | 0000000000000004 | | 0000000000000004 | 0000000000000004 | 0000000000000004 | | A | 3 | 0 |
| [3] | .dynsym | DYNSYM | 0000000000400188 | 00000188 | 0000000000000000 | 0000000000000018 | A | 4 | 1 | 8 |
| | 0000000000000000 | 0000000000000018 | | 0000000000000018 | 0000000000000018 | 0000000000000018 | | A | 4 | 1 |
| [4] | .dynstr | STRTAB | 0000000000400188 | 00000188 | 000000000000000b | 000000000000000b | A | 0 | 0 | 1 |
| | 000000000000000b | 000000000000000b | | 000000000000000b | 000000000000000b | 000000000000000b | | A | 0 | 0 |
| [5] | .text | PROGBITS | 0000000000400193 | 00000193 | 0000000000000037 | 0000000000000037 | AX | 0 | 0 | 1 |
| | 0000000000000037 | 0000000000000037 | | 0000000000000037 | 0000000000000037 | 0000000000000037 | | AX | 0 | 0 |
| [6] | .eh_frame | PROGBITS | 00000000004001d0 | 000001d0 | 0000000000000000 | 0000000000000000 | A | 0 | 0 | 8 |
| | 0000000000000000 | 0000000000000000 | | 0000000000000000 | 0000000000000000 | 0000000000000000 | | A | 0 | 0 |
| [7] | .dynamic | DYNAMIC | 00000000006001d0 | 000001d0 | 00000000000000d0 | 0000000000000010 | WA | 4 | 0 | 8 |
| | 00000000000000d0 | 0000000000000010 | | 0000000000000010 | 0000000000000010 | 0000000000000010 | | WA | 4 | 0 |
| [8] | .data | PROGBITS | 00000000006002a0 | 000002a0 | 00000000000000e | 00000000000000e | WA | 0 | 0 | 1 |
| | 00000000000000e | 00000000000000e | | 00000000000000e | 00000000000000e | 00000000000000e | | WA | 0 | 0 |
| [9] | .debug_aranges | PROGBITS | 0000000000000000 | 000002b0 | 0000000000000030 | 0000000000000030 | 0000000000000030 | 0000000000000030 | 0000000000000030 | 0000000000000030 |
| | 0000000000000030 | 0000000000000030 | | 0000000000000030 | 0000000000000030 | 0000000000000030 | | 0000000000000030 | 0000000000000030 | 0000000000000030 |
| [10] | .debug_info | PROGBITS | 0000000000000000 | 000002e0 | 0000000000000078 | 0000000000000078 | 0000000000000078 | 0000000000000078 | 0000000000000078 | 0000000000000078 |
| | 0000000000000078 | 0000000000000078 | | 0000000000000078 | 0000000000000078 | 0000000000000078 | | 0000000000000078 | 0000000000000078 | 0000000000000078 |
| [11] | .debug_abbrev | PROGBITS | 0000000000000000 | 00000358 | 0000000000000014 | 0000000000000014 | 0000000000000014 | 0000000000000014 | 0000000000000014 | 0000000000000014 |
| | 0000000000000014 | 0000000000000014 | | 0000000000000014 | 0000000000000014 | 0000000000000014 | | 0000000000000014 | 0000000000000014 | 0000000000000014 |
| [12] | .debug_line | PROGBITS | 0000000000000000 | 0000036c | 000000000000004a | 000000000000004a | 000000000000004a | 000000000000004a | 000000000000004a | 000000000000004a |
| | 000000000000004a | 000000000000004a | | 000000000000004a | 000000000000004a | 000000000000004a | | 000000000000004a | 000000000000004a | 000000000000004a |
| [13] | .shstrtab | STRTAB | 0000000000000000 | 000003b6 | 000000000000008d | 000000000000008d | 000000000000008d | 000000000000008d | 000000000000008d | 000000000000008d |
| | 000000000000008d | 000000000000008d | | 000000000000008d | 000000000000008d | 000000000000008d | | 000000000000008d | 000000000000008d | 000000000000008d |
| [14] | .symtab | SYMTAB | 0000000000000000 | 00000848 | 0000000000000240 | 0000000000000240 | 0000000000000240 | 0000000000000240 | 0000000000000240 | 0000000000000240 |
| | 0000000000000240 | 0000000000000240 | | 0000000000000240 | 0000000000000240 | 0000000000000240 | | 0000000000000240 | 0000000000000240 | 0000000000000240 |
| [15] | .strtab | STRTAB | 0000000000000000 | 00000a88 | 000000000000006f | 000000000000006f | 000000000000006f | 000000000000006f | 000000000000006f | 000000000000006f |
| | 000000000000006f | 000000000000006f | | 000000000000006f | 000000000000006f | 000000000000006f | | 000000000000006f | 000000000000006f | 000000000000006f |

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

Enlace de Secciones

- el linker mezcla de forma organizada cada tipo de sección de todos los módulos objeto reubicables generando un único módulo objeto ejecutable
- Ejemplo de tres módulos objeto reubicables:
 - los tres módulos fuente p1.c, p2.c, p3.c se compilan dando lugar a p1.o, p2.o y p3.o los cuales se enlazan dando lugar al ejecutable *p*

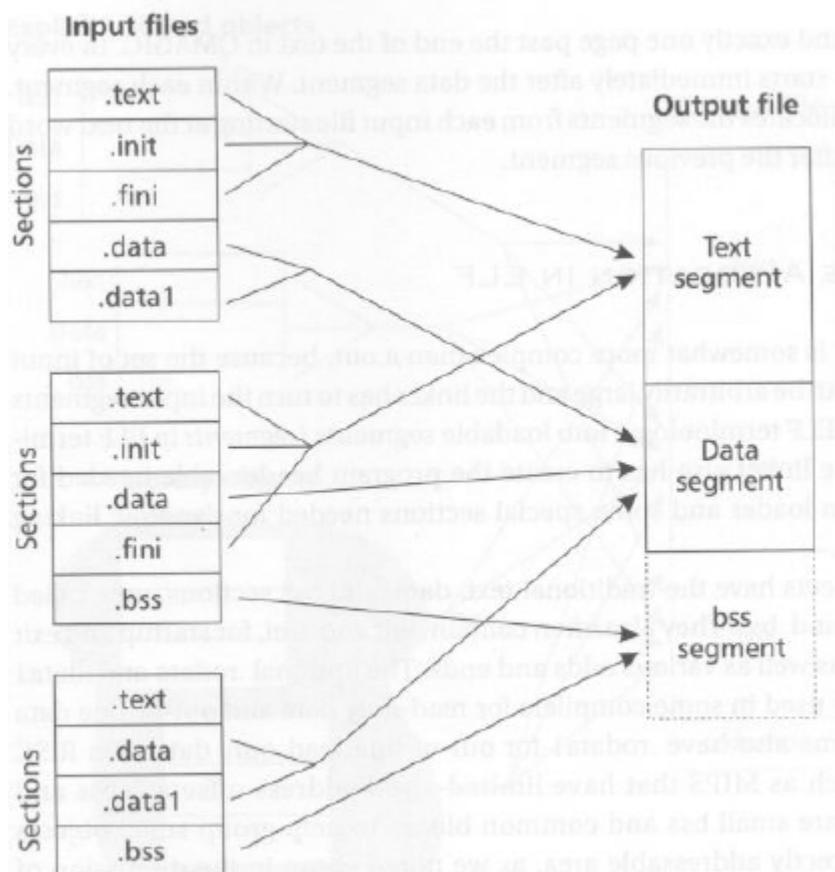


FIGURE 4.9 • ELF linking.

Segmentos lógicos

- El módulo ejecutable está estructurado en segmentos
 - text
 - código de las instrucciones a ejecutar
 - data
 - código de datos: variables inicializadas, sin inicializar
 - stack
 - pila
 - heap

- montículo
- es la asignación de memoria en tiempo de ejecución
- en C la función `malloc()`: memory allocation: `void *malloc(size_t size)`
 - size: tamaño en bytes de la memoria a asignar
 - devuelve un puntero a la región de memoria asignada
- mapa de memoria del programa en ejecución

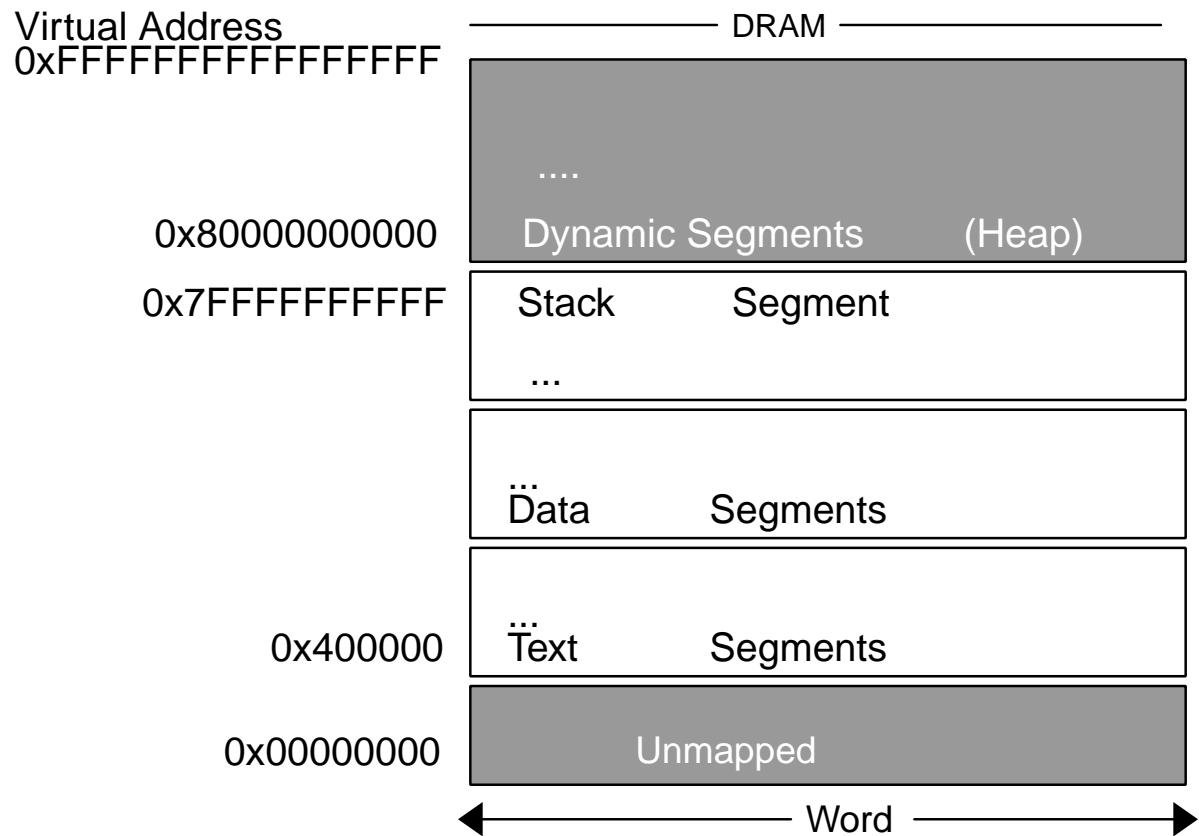


Figure 9.26
The virtual memory of a Linux process.

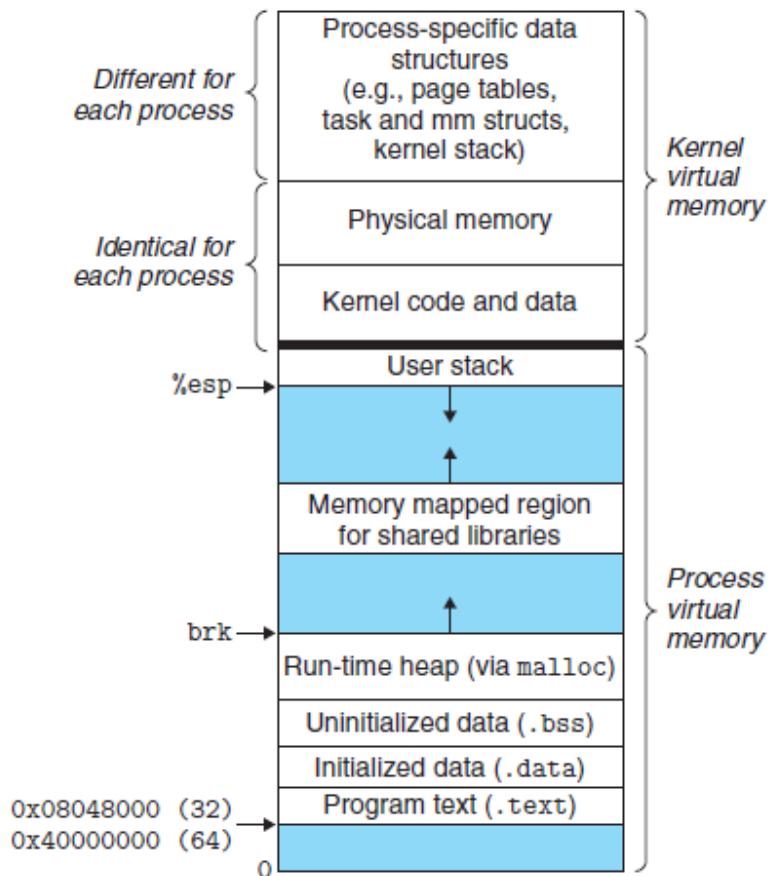


Figure 49. linux_vm_map

Figure 7.13
Linux run-time memory image.

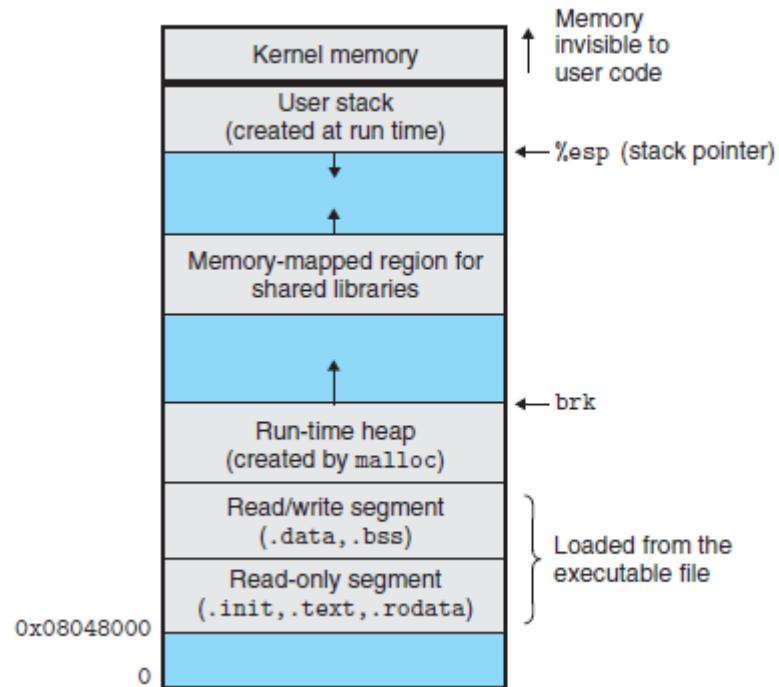
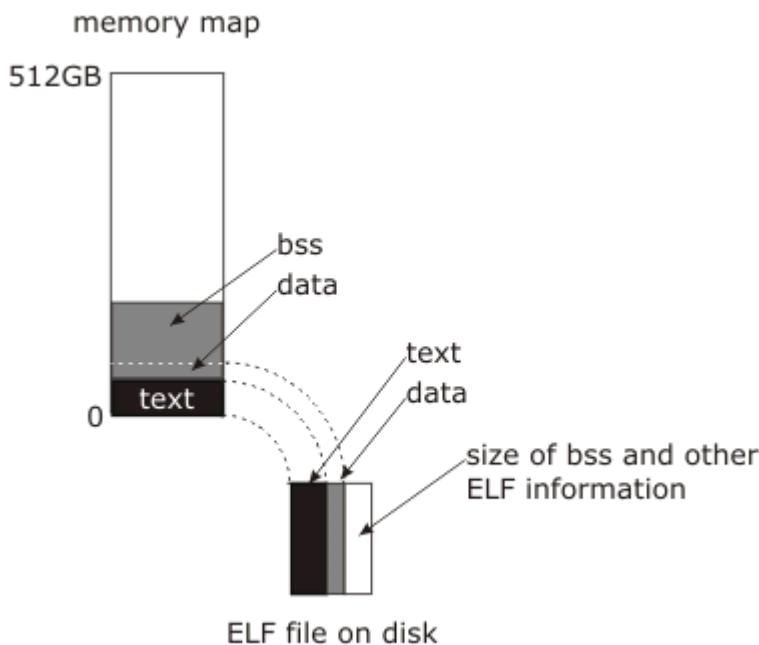


Figure 50. linux_vm_map_2

- Cada proceso tiene su propia memoria virtual independiente del resto de los procesos
- Los segmentos pueden cambiar de tamaño dinámicamente en tiempo de ejecución.
 - Carga del módulo objeto ejecutable

- El loader no carga el módulo ejecutable en DRAM, sino que mapea el fichero a memoria virtual, creando la tabla de páginas.
- La carga efectiva se realiza bajo demanda.



Evolución memoria Intel 8086-80286

8086

- 80x86 → (bits bus direcciones, bits bus datos)
- 8086 → (20,16) → 2^{20} =1MB de memoria física→ Modo Real
 - Segmentación
 - La dirección lógica esta formada por un tuple de dos valores: dirección base y offset.
 - Conversión de dirección lógica a dirección física:
 - Con un contador de programa de 16 bits se pueden direccionar 64KB. Si añadimos un registro segmento adicional de 16 bits cuyo contenido lo desplazamos 4 bits a la izda (equivale a **multiplicar por 2^4**) tendríamos una dirección base de 20 bits a la cual añadiríamos el offset del PC de 16 bits obteniendo una dirección física de 20 bits- > espacio físico de 1MB.
 - Este modo de memoria se denominó *modo real*: espacio de direcciones memoria segmentada de 20 bits.

80286

- 80286 → (24,16) → 2^{24} =16MB de memoria física→ Modos Real y protegido.
 - Concepto de memoria Virtual: memoria generada por el compilador y por los procesos al ejecutarse
 - En este caso son 4 bytes de memoria virtual → los 2 bytes más altos son el selector de segmento y los dos bytes más bajos el offset.
 - Capacidad de memoria virtual → $2^{32} = 4GB$
 - La memoria virtual de los procesos se parte en segmentos.
 - Segmentación
 - Forming different segments for data, code, and stack, and preventing their overlapping

- Cada segmento únicamente puede direccionar 64KB ya que el Contador de Programa es de 16 bits
- La conversión memoria lógica a memoria física:
 - Se utiliza uno de los 4 registros de segmento CS,DS,ES,SS: son de 64 bits: 16 bits visibles y 6 bytes escondidos
 - Se utiliza una tabla de descripción del segmento residente en la memoria principal: cada entrada de la tabla son 8 bytes de los cuales 3 bytes son la dirección base física asociada a la dirección virtual segmentada.
 - En la parte visible del registro de segmento se cargan los 2 bytes más altos de la dirección virtual (selector de segmento)
 - El selector de segmento apunta a una de las entradas de la tabla de selección de descripción de segmento y carga 6 bytes de la tabla en la zona escondida del registro de segmento el cual contiene: dirección base física (3bytes), tamaño del segmento (2 bytes) y propiedades del segmento (1byte)
 - dirección física: la dirección base (3bytes) más el offset (2bytes): con 3 bytes $\rightarrow 2^{24} = 16\text{MB}$ de espacio físico
 - El espacio de direcciones de 4GB de memoria virtual de un segmento debiera poder traducirse en el espacio de direcciones físicas de 16MB, pero únicamente puede acceder a 64KB.
 - Espacio físico total: de los 16MB posibles un segmento direcciona solo 64KB y como tenemos 4 segmentos $\rightarrow 4 \times 64\text{KB} = 256\text{KB}$ totales.
- Multitasking, memory management (on chip MMU), protected memory \rightarrow modo protegido: espacio de direcciones memoria segmentada de 24 bits.

80386

- 80386 \rightarrow (32,32) \rightarrow Espacio Físico: $2^{32} = 4\text{GB}$
 - Misma arquitectura que el 286 pero incrementa la ruta de datos de 16 bits a 32 bits, añade dos registros de segmento más (FS,GS) y añade la técnica de la paginación.
 - Memoria Virtual: 6 bytes : $2^{48} = 64\text{TB}$. Los 2 bytes altos son el selector de registro y los 4 bytes bajos el offset
 - de los 64TB posibles los 6 segmentos pueden direccionar *en un momento dado* 4GB cada uno $\rightarrow 6 \times 4\text{GB} = 24\text{GB}$
 - Segmentación
 - Selector de Segmento = 2 bytes como en el 286 \rightarrow puntero a la entrada de la tabla descriptor de segmento
 - Descriptor de Segmento = Contiene 4 bytes de la dirección base física
 - Conversión de la dirección lógica a dirección física con sólo segmentación
 - A la dirección base física (4 bytes) se le añade el offset de la dirección virtual (4bytes) \rightarrow dirección física de 32 bits.
 - En este caso, a diferencia del 80286, el espacio de memoria virtual de 4GB de cada segmento se pueden traducir en el espacio físico de 4GB.
 - instrucciones
 - `movl $42,%fs:(%eax)`
 - implícitamente
 - push, pop \rightarrow SS,DS
 - Ver paginación 80386

amd64

- amd64 → (52,64) → Espacio Físico: $2^{52} = 4\text{PetaBytes}$ y Espacio Virtual $2^{48} = 256\text{TB}$
 - https://en.wikipedia.org/wiki/X86-64#Physical_address_space_details
- **No utiliza la segmentación lógica** del espacio de direcciones virtual debido a que el espacio de memoria virtual de 256TB es suficiente para todos los procesos. Los segmentos lógicos (text,data,stack,heap, etc) de un proceso se almacenan en el mismo espacio virtual asignado a dicho proceso mediante la técnica de paginación.
- Hay que tener en cuenta la limitación de la tabla de páginas virtual que depende del número de páginas virtuales y la dirección de una página física. El área de memoria principal ocupada crece exponencialmente con el tamaño de la tabla y de forma innecesaria.

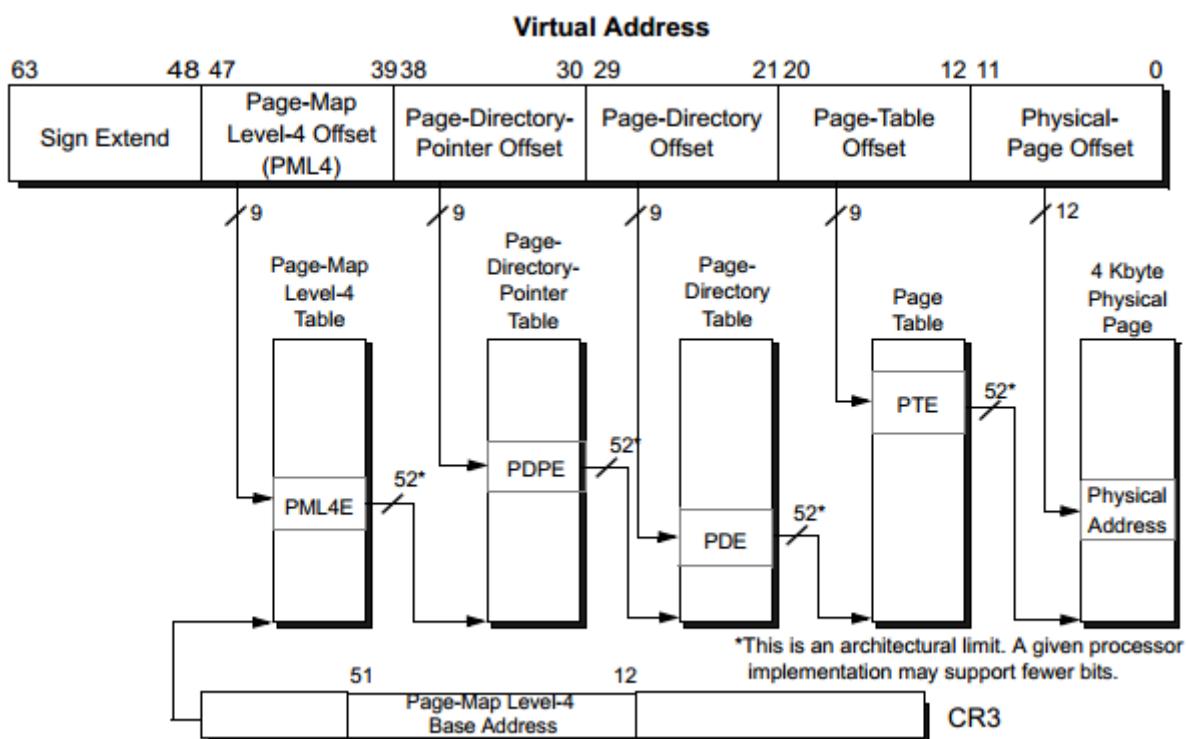


Figure 51. Formato de direcciones amd64

9.5.4. Memoria Virtual Paginada

Fundamento

- La paginación consiste en dividir tanto la memoria *física* como la memoria *virtual* de los procesos en pequeños *pedazos* denominados páginas.
- Los pedazos de memoria física se denominan *marcos de página* y los pedazos de memoria virtual del proceso se denominan *páginas*.
- En este caso se asigna cada página a un marco de página diferente, quedando los trozos de proceso diseminados en zonas **NO CONTIGUAS** de la memoria.
- De esta manera se reduce la fragmentación interna ,ya que la memoria infrautilizada siempre será menor al tamaño de una página.
- El sistema operativo genera para cada proceso la *tabla de páginas* que mapea páginas con marcos.
 - PTE: Page Table Entry → (index,PhysPageNumber)
- Dirección lógica
 - Cada dirección lógica estará formada por la dirección base de la página y el offset dentro de la página. Direccionamiento **no lineal**, (dirección base, desplazamiento)

- Traducción de dirección lógica a física.
 - El espacio de direcciones físico es único y contiguo, es decir, lineal.
 - La dirección base de la página del proceso se asocia con la dirección base del marco: tabla de páginas.
 - El offset dentro del marco será el mismo que el offset dentro de la página.
 - La gestión de la paginación la realiza la MMU

Concepto de Memoria Virtual Paginada

- Debido a que no es necesario cargar todas las páginas del proceso → el espacio de memoria del **PROCESO** puede ser **mayor** que la memoria física → concepto de **memoria virtual**
- La memoria virtual es única, contigua, es decir, *LINEAL*. Es una abstracción para no depender de las direcciones físicas.
- Por el principio de localidad en la memoria física sólo está la copia de las páginas virtuales que son necesarias dinámicamente en un momento dado.
 - **concepto de cache**
- La memoria principal es la cache de la memoria secundaria (ficheros el disco o pendrive)
- SDRAM cache

Fragmentación

- En el desalojo de áreas de memoria que no son necesarias se generan huecos que fragmentan la memoria física en una sucesión de áreas de memoria utilizadas y áreas no utilizadas
- La fragmentación interna será menor cuanto más pequeñas sean las páginas.
 - En la paginación las páginas pueden tener bytes sin utilizar, son huecos internos a las páginas.
- La fragmentación externa se reduce ya que los marcos de página pueden ser asignados a un proceso independientemente del tamaño del proceso.
 - Un proceso ocupará los huecos dejados por las páginas que no tienen porque ser contiguas. El tamaño del proceso afectará al número de páginas requeridas en caso de que queramos tener todo el proceso residente en memoria principal.

MMU

- Unidad Hardware interna a la CPU
- Su entrada es el bus de direcciones virtuales y su salida el bus de direcciones físicas.
- La MMU accede a la tabla de descripción de segmentos y a la tabla de páginas y realiza la traducción de dirección virtual en dirección física
- La tabla de páginas de la MMU es la función de correspondencia que mapea el espacio virtual y el físico.

Virtual Memory Cached

- Tabla de páginas con función de correspondencia fully associative (las VPages se asocian con cualquier Marco de página)
- Bit de validación
 - 1 → cached page
 - 0 → uncached page: allocated o unallocated

Figure 9.2
A system that uses virtual addressing.

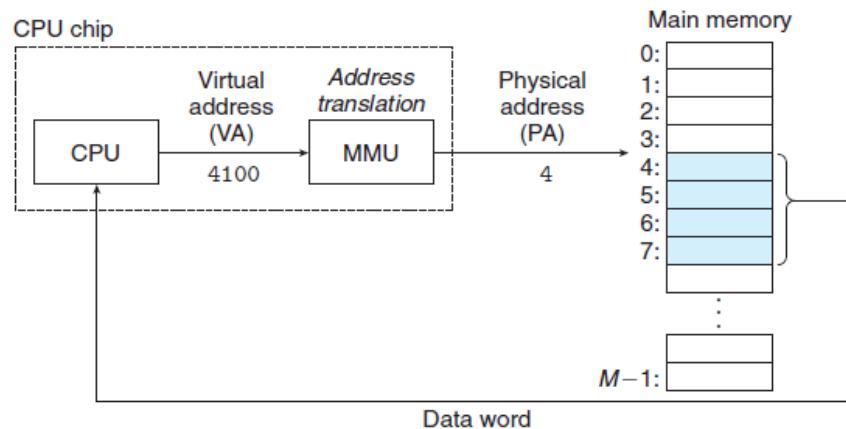


Figure 52. MMU

Figure 9.4
Page table.

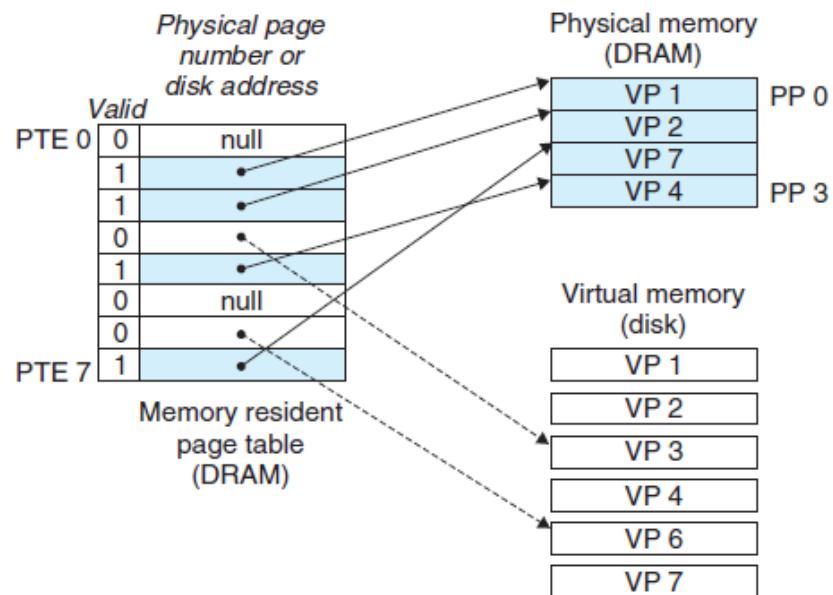


Figure 53. MMU

Tabla de páginas

- La tabla de páginas reside en la memoria principal SDRAM.
- Las entradas de la tabla son un puntero a marcos de página física
- Tantas entradas como páginas virtuales
- El número de página virtual es el índice de la tabla.
- La MMU accede a la tabla de páginas y realiza la traducción de dirección virtual en dirección física
- El kernel actualiza la tabla de páginas y activa las transferencias

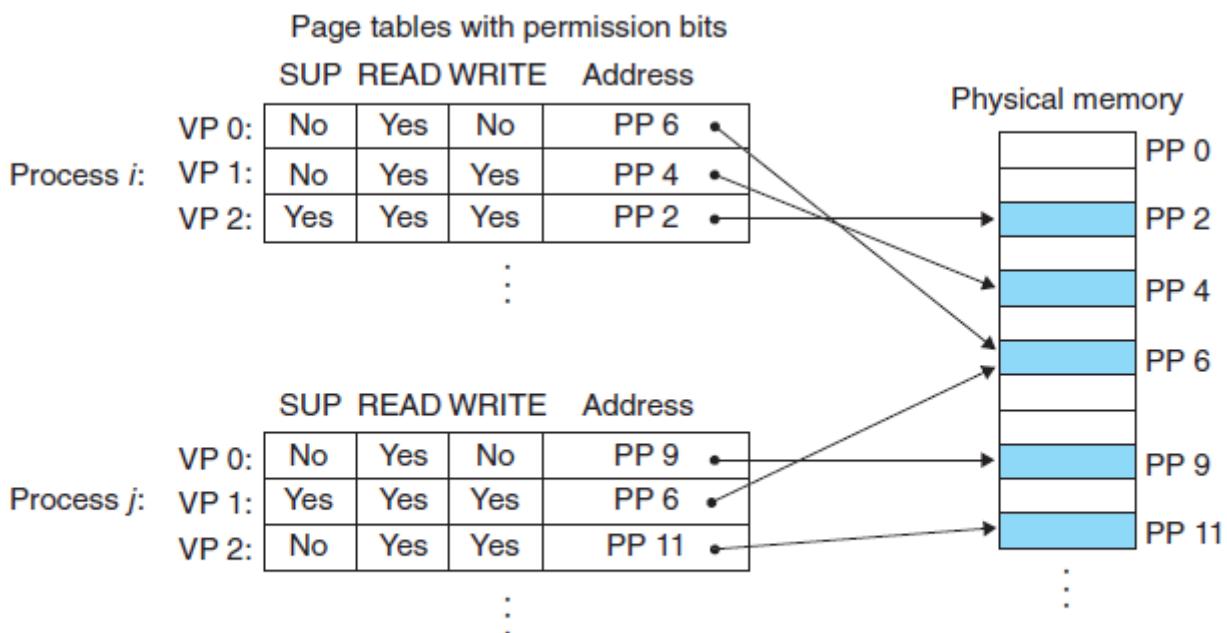


Figure 9.10 Using VM to provide page-level memory protection.

Figure 54. Protección

- SUP: SUPervisor: únicamente el kernel tiene acceso
- Write No: read only.

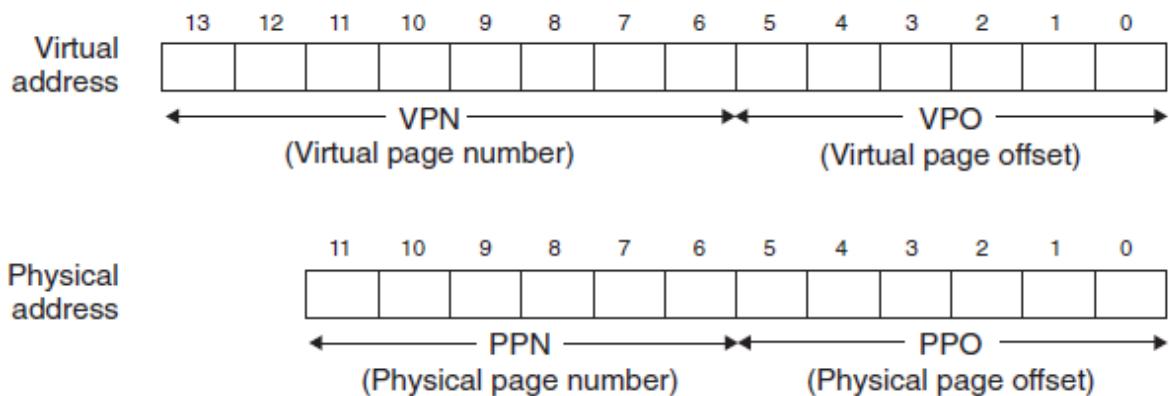


Figure 9.19 Addressing for small memory system. Assume 14-bit virtual addresses ($n = 14$), 12-bit physical addresses ($m = 12$), and 64-byte pages ($P = 64$).

Figure 55. Formato de Direcciones

Multilevel paging

- Debido a qué la tabla de páginas puede ser enorme se considera reducir el área de ram que consume la tabla mediante una organización jerárquica de múltiples tablas.
- La memoria se puede dividir jerárquicamente en agrupamientos de páginas. Superpáginas que agrupan páginas, hiperpáginas que agrupan superpáginas.
- Ejemplo: Paginación de 3 niveles: Nivel 1 de pedazos de 16 MB, nivel 2 de pedazos de 2MB, nivel 2 de páginas de 4KB.
- Cada nivel de agrupamiento lleva asociada una tabla de descripción de dicho nivel. La tabla de páginas se convierte en una jerarquía de múltiples tablas.

- En el proceso de traducción de la MMU la dirección virtual se descompone en múltiples campos. Cada campo será un índice de cada tabla asociada, enlazando tantas páginas como niveles.
- Si la tabla de nivel i tiene un contenido NULL no existirán las tablas de niveles superiores $i+1, i+2, \dots$ de la cadena de enlaces.
- El hecho de acceder a múltiples tablas no ralentiza la traducción de direcciones si las tablas están implementadas en la cache interna de la MMU. Sería distinto si dichas páginas estuviesen en la memoria DRAM.

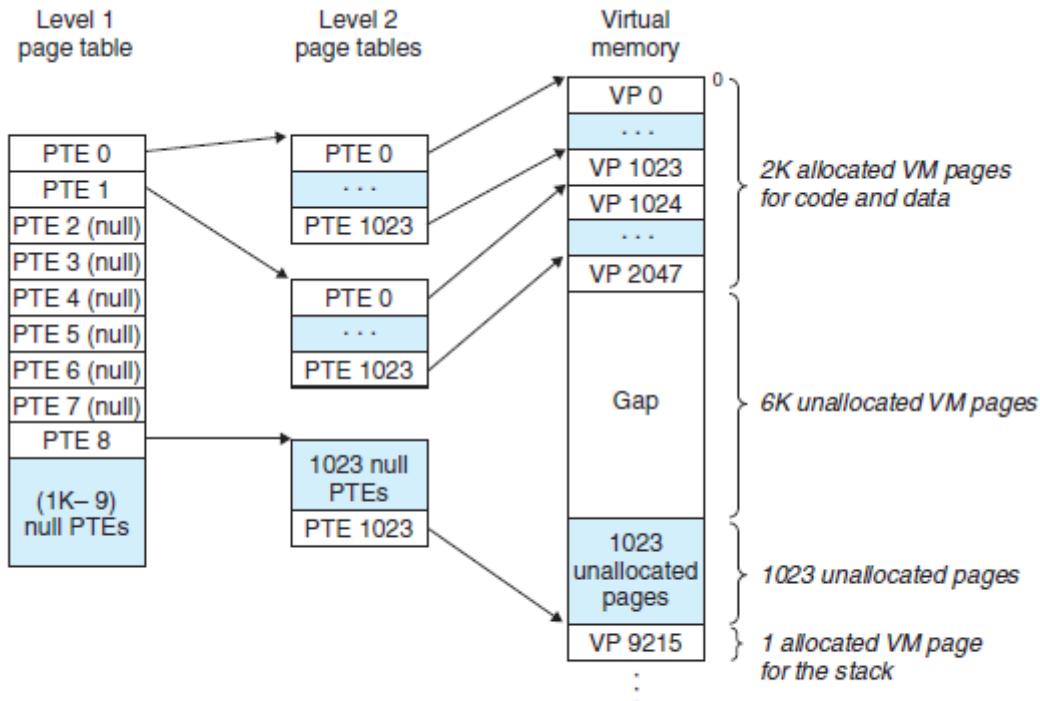


Figure 9.17 A two-level page table hierarchy. Notice that addresses increase from top to bottom.

Figure 56. Tabla de dos niveles

Figure 9.18
Address translation with a k -level page table.

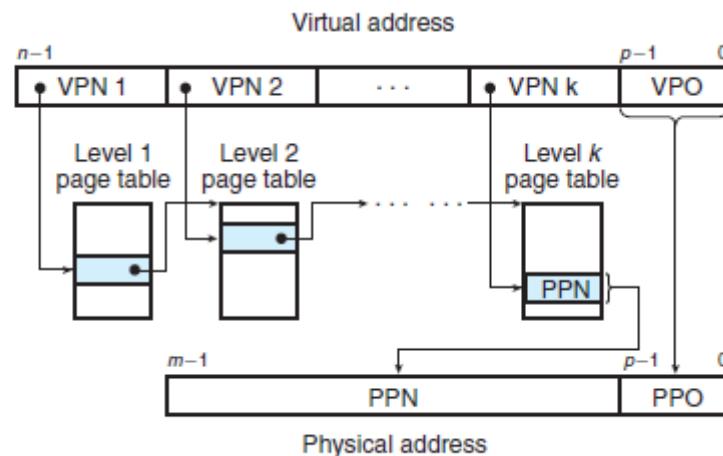


Figure 57. Tabla de K niveles

Intel: Evolución memoria virtual

80386

- Se utiliza por primera vez la paginación.
- La traducción de memoria virtual en física conlleva dos fases: primero la segmentación y a continuación la

paginación (opcional)

- Ver mecanismo de segmentación.
- La segmentación traduce el espacio virtual en un espacio lineal de 32 bits con campos:dir(10 bits)-pag(10)-offset(12)
- Se implementa dos niveles de tablas de páginas: dir es una tabla de punteros de tablas de páginas (directorio de páginas)
 - con 10 bits se consiguen 2^{10} punteros a tablas → 1K tablas
- pag es el índice de la tabla de páginas
 - con 10 bits se consiguen 2^{10} entradas de tabla → 1K páginas virtuales asociadas a 1K páginas físicas
 - La dirección de página física son 32 bits
- Con 12 bits de offset el tamaño de página es $2^{12}=4KB$
- 1K tablas donde cada tabla contiene 1K páginas son en total 1M de páginas y cada página 4KB da un total de 4GB de direcciones de memoria física.
- Por lo que de los 64TB de memoria virtual posible podemos traducir en un momento dado a 24GB de memoria segmentada y cada segmento de 4GB lineales a 4GB de memoria física.

amd64

- amd64 → 64 bits → Espacio Virtual teórico = $2^{64} = 16$ ExaBytes
 - Paginación y **no segmentación**.
 - Espacio Virtual = 256 TeraBytes ya que la CPU únicamente utiliza 48 bits para el espacio de direcciones virtual *porque* es suficiente memoria para las aplicaciones actuales, utilizar los 64 bits provocaría tablas de páginas enormes bajando el rendimiento del sistema sin necesidad. No hay ni memoria secundaria para tanta memoria virtual.

Glosario

- Espacios: Logic (segmentation) → Logic Linear (virtual,pagination) → Physical Linear
- VP: Virtual Page
- VA: Virtual Address
- PP: Physical Page
- PA: Physical Address
- VPO:VP offset
- VPN:VP number
- TLB: Translation lookaside Buffer: buffer (cache) de anticipación de la tabla de páginas. Residente en la MMU.
- PTE: Page Table Entry → (index/contenido)→(VPN/PPN)
- PTBR: Registro de control de la CPU: page table base register: pointer to TLB
- TLBI:TLB index → campo set de la cache
- TLBT: TLB tag
- PPO: PP offset
- PPN: PP number
- CO: Cache offset en el superbloque
- CT: Cache tag
- CI: Cache index ó línea

Traducción: dirección virtual a física

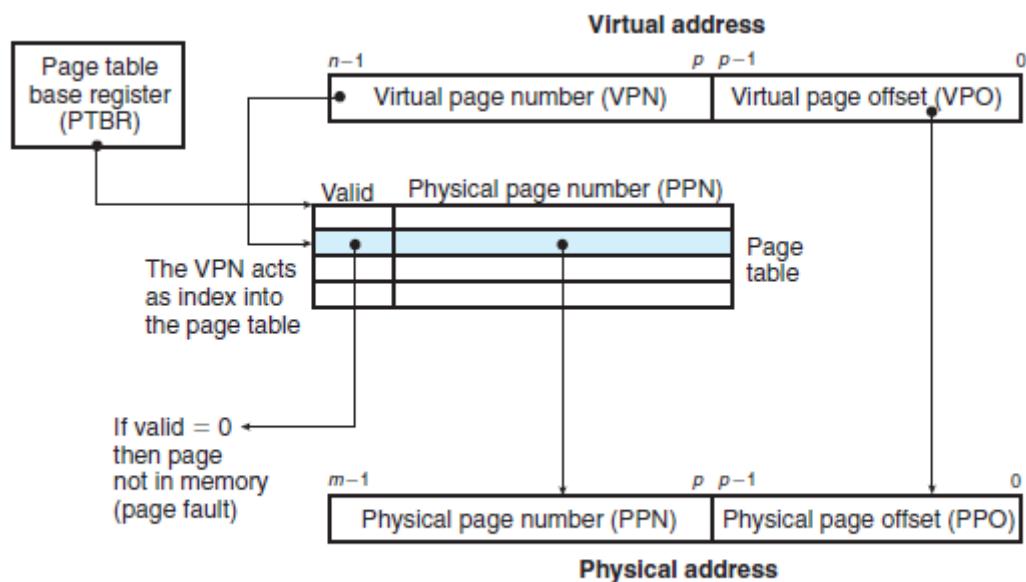


Figure 9.12 Address translation with a page table.

Figure 58. Traducción Virtual → Física

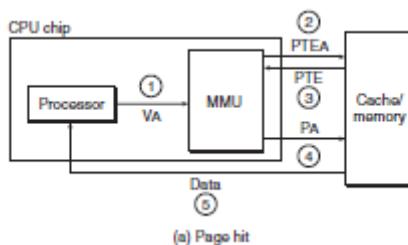


Figure 59. Resultado con éxito

1. CPU: vuela la dirección de memoria virtual
2. MMU: apunta a la entrada de la tabla de páginas ubicada en la memoria principal
3. Memoria Principal: devuelve el contenido de la entrada de la tabla. MMU: A partir de la dirección lógica obtiene al dirección física.
4. MMU: vuela la dirección física en el bus de direcciones del sistema.
5. El dato referenciado puede estar en la memoria caché o en la memoria principal.

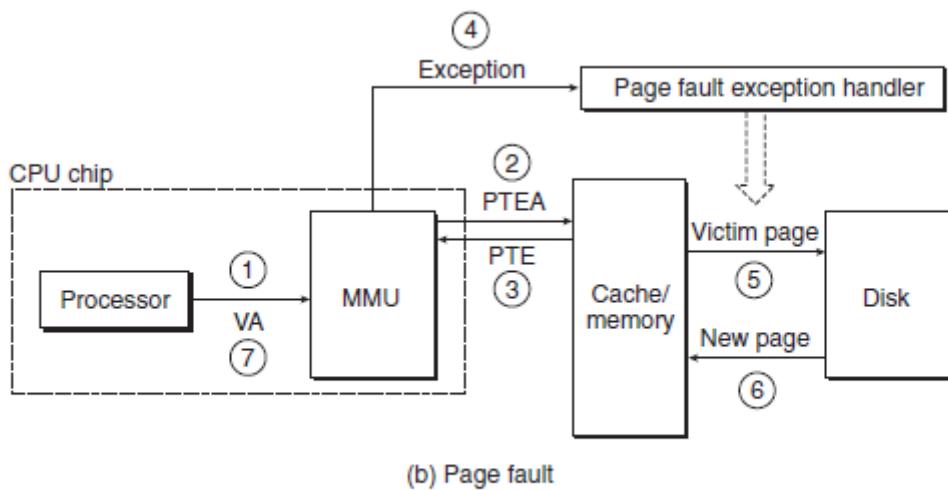


Figure 9.13 Operational view of page hits and page faults. VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

Figure 60. Resultado con fracaso

Translation Lookaside Buffer

- TLB
- Es una Caché de la tabla de páginas virtuales TPV. Además de residir la tabla de páginas en la memoria principal se tiene una copia parcial de dicha tabla en una unidad de memoria interna de la MMU. Objetivo: aumentar la velocidad de acceso a la tabla ya que la solución de múltiples tablas en niveles jerárquicos requiere múltiples accesos a la memoria principal externa.
- Formato de dirección virtual si la TLB es una caché con función de correspondencia asociativa
 - El índice es el campo set o superbloque típico de la cache

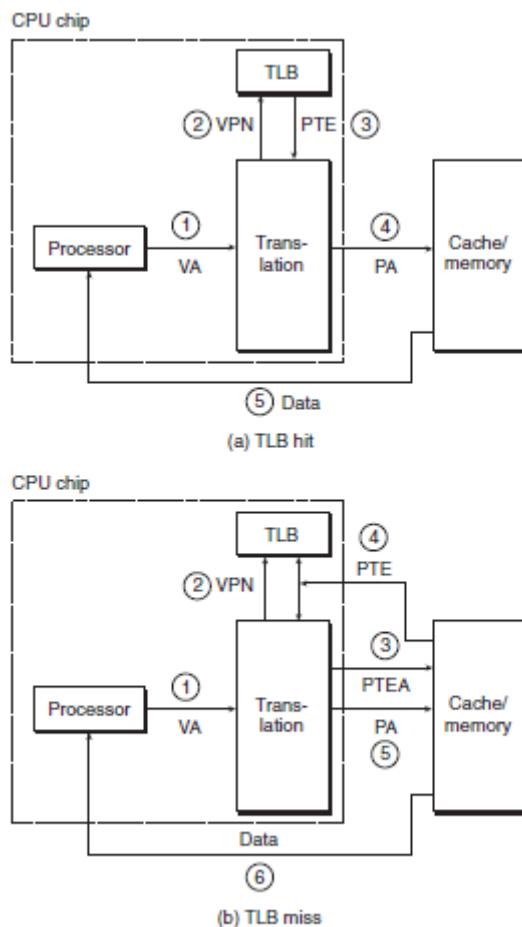


Figure 9.16 Operational view of a TLB hit and miss.

Figure 61. Operación con TLB

Figure 9.15
Components of a virtual address that are used to access the TLB.

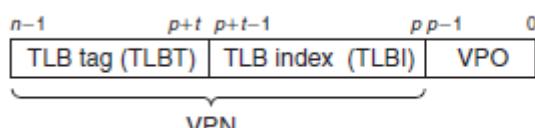
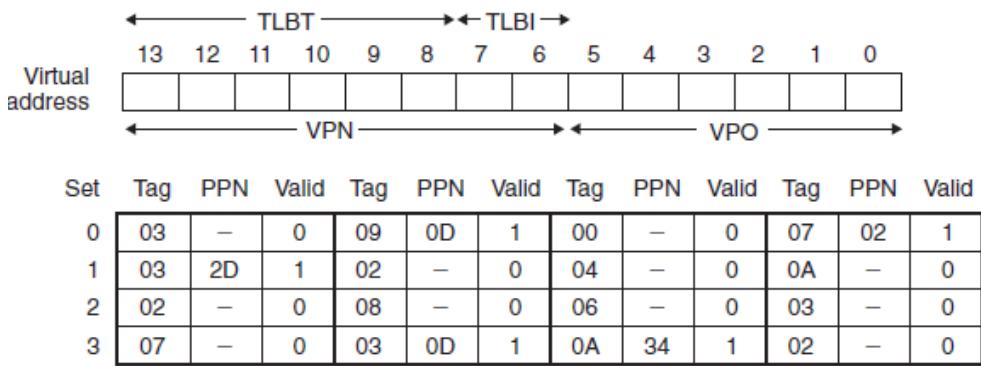


Figure 62. Formato Virtual con TLB

- TLBTag
- TLBIndex

Ejercicio

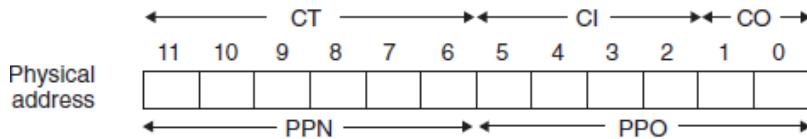
- La arquitectura de una computadora dispone de TLB y L1 d-Cache. La memoria es direccionable byte a byte y tiene palabras de 1 byte.
- La MMU tiene una Tabla TLB (Translation Lookup Buffer) y una memoria d-Cache según las figuras



(a) TLB: Four sets, 16 entries, four-way set associative

| VPN | PPN | Valid | VPN | PPN | Valid |
|-----|-----|-------|-----|-----|-------|
| 00 | 28 | 1 | 08 | 13 | 1 |
| 01 | - | 0 | 09 | 17 | 1 |
| 02 | 33 | 1 | 0A | 09 | 1 |
| 03 | 02 | 1 | 0B | - | 0 |
| 04 | - | 0 | 0C | - | 0 |
| 05 | 16 | 1 | 0D | 2D | 1 |
| 06 | - | 0 | 0E | 11 | 1 |
| 07 | - | 0 | 0F | 0D | 1 |

(b) Page table: Only the first 16 PTEs are shown



| Idx | Tag | Valid | Blk 0 | Blk 1 | Blk 2 | Blk 3 |
|-----|-----|-------|-------|-------|-------|-------|
| 0 | 19 | 1 | 99 | 11 | 23 | 11 |
| 1 | 15 | 0 | - | - | - | - |
| 2 | 1B | 1 | 00 | 02 | 04 | 08 |
| 3 | 36 | 0 | - | - | - | - |
| 4 | 32 | 1 | 43 | 6D | 8F | 09 |
| 5 | 0D | 1 | 36 | 72 | F0 | 1D |
| 6 | 31 | 0 | - | - | - | - |
| 7 | 16 | 1 | 11 | C2 | DF | 03 |
| 8 | 24 | 1 | 3A | 00 | 51 | 89 |
| 9 | 2D | 0 | - | - | - | - |
| A | 2D | 1 | 93 | 15 | DA | 3B |
| B | 0B | 0 | - | - | - | - |
| C | 12 | 0 | - | - | - | - |
| D | 16 | 1 | 04 | 96 | 34 | 15 |
| E | 13 | 1 | 83 | 77 | 1B | D3 |
| F | 14 | 0 | - | - | - | - |

(c) Cache: Sixteen sets, 4-byte blocks, direct mapped

Figure 9.20 TLB, page table, and cache for small memory system. All values in the TLB, page table, and cache are in hexadecimal notation.

Figure 63. TLB y d-cache

- Virtual addresses are 14 bits wide ($n = 14$).
 - Physical addresses are 12 bits wide ($m = 12$).
 - The page size is 64 bytes ($P = 64$).
 - The TLB is four-way set associative with 16 total entries.
 - The L1 d-cache is physically addressed and direct mapped, with a 4-byte line size and 16 total set.
- Calcular la dirección física de la DIRECCION VIRTUAL **0x03d4**
 - a. Formato de Direcciones
 - i. Dimensión de VPO
 - ii. Dimensión de PPO
 - iii. Dimensión de VPN
 - iv. Dimensión de PPN
 - b. Número de entradas de la tabla de páginas en memoria principal y la caché TLB
 - c. TLB
 - i. Líneas por set de TLB
 - ii. Sets de TLB
 - iii. Tamaño TLBI
 - iv. Tamaño TLBT
 - v. Bits por Word
 - vi. Words por línea de TLB
 - vii. Valores TLBI-TLBT
 - d. Está PPN en TLB?
 - e. Valor de PPN
 - f. Valor de PA
 - g. d-Cache
 - i. Memory Cache: Tipo
 - ii. Sets
 - iii. Líneas/Set
 - iv. Words/Línea
 - v. Bytes/Word
 - h. Formato Dirección Física
 - i. CO
 - ii. CI
 - iii. CT
 - iv. Valores CT/CI/CO → PA
 - i. Está PA en la caché?
 - j. Contenido de la PA
 - k. Resumen del resultado final

Desarrollo

- Respuestas

a. El formato de direcciones es

- i. VO y PO → tamaño de página : 64 bytes → 2^6 → 6 bits de offset tanto virtual como físico
- ii. bits VPN= VA-VPO=14-6=8 bits → 2^8 = 256 páginas virtuales
- iii. bits PPN= PA-PPO=12-6=6 bits → 2^6 = 64 marcos de página

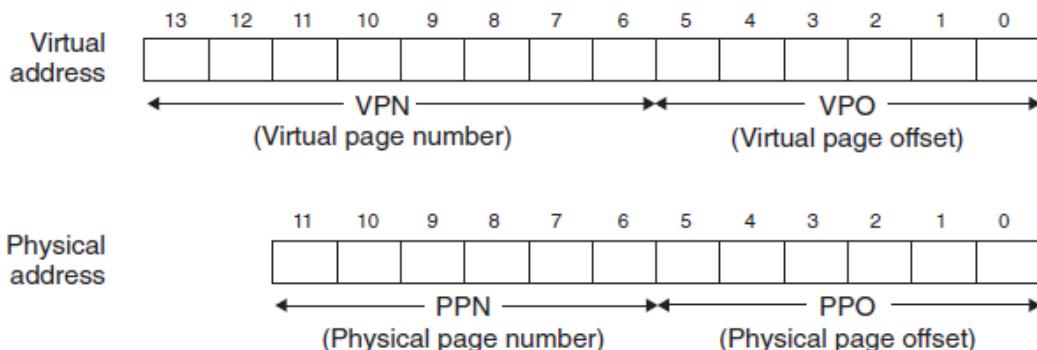


Figure 9.19 Addressing for small memory system. Assume 14-bit virtual addresses ($n = 14$), 12-bit physical addresses ($m = 12$), and 64-byte pages ($P = 64$).

Figure 64. Formato de Direcciones

b. Valores de VPN y VPO

- La dirección virtual VA de 14 bits 0x03D4 se codifica en binario como: 00-0011-1101-0100 → 00001111-010100 → VPN-VPO
 - VPO=PPO=010100=01x4
 - VPN=00001111=0x0F

c. Tabla de páginas

- 256 puntos de entrada. Cada entrada contiene la dirección de uno de los 64 marcos de página. Es decir una tabla de 256 direcciones y palabras de 6 bits más los bits de validación, protección, etc
- La tabla en MP son 256 entradas, en cambio la TLB en caché tiene 16 entradas → direccionables con 4 bits.

d. TLB

- i. Líneas por set de TLB: 4 vías → 4 líneas/set
- ii. Sets de TLB: 16 entradas son 16 líneas en total agrupadas por 4 líneas/set = 4 sets
- iii. Tamaño TLBI : para 4 set son necesarios 2 bits
- iv. Tamaño TLBT :
 - De los 8 bits necesarios para direccionar 256 entradas si 2 son para el índice TLBI, 6 serán para la etiqueta TLBT
- v. Bits por Word: 1 byte por palabra según el enunciado
- vi. Words por línea de TLB
 - Si me fijo en el dibujo de la tabla, cada línea contiene únicamente un PPN+tag, es decir, una palabra.
- vii. Valores TLBI-TLBT
 - VPN es una dirección de la tabla de páginas en la memoria RAM. El controlador de caché la descompone en TLBT-TLBI
 - VPN=00001111=000011-11=TLBT-TLBI=0x3-0x3

e. Está PPN en TLB?

- busco en el set 0x3 de TLB si alguno de las líneas tiene un tag TLBT de 0x3 y lo tiene la segunda línea.
- La segunda línea del set 3 tiene el bit de validación a 1 por lo que la página virtual está en la memoria principal y/o d-cache.

f. Valor de PPN

- La segunda línea del set 3 tiene el contenido PPN=0x0D

g. Valor de PA

- Son 12 bits
- La concatenación PPN(6)-PPO(6): 001101-010100=001101010100=0011-0101-0100= **0x354** =PA

h. d-Cache

i. Memory Cache: Tipo : mapeo directo

- Al ser de mapeo directo los set son de 1 línea por lo que es lo mismo decir set que línea.

ii. Sets

- 16 líneas

iii. Líneas/Set : 1

iv. Words/Línea: 4

v. Bytes/Word: 1

i. Formato dirección física

- CO: para direccionar 4 palabras son necesarios 2 bits
- CI: para direccionar 16 líneas son necesarios 4 bits
- CT: la dirección física PA son 12 bits → CT=PA-DI-CO=12-4-2=6 bits
- Valores CT/CI/CO → PA=001101010100=001101-0101-00
 - Línea 0005; Palabra 00: Tag 001101=0x0D

j. Está PA en la d-Cache?

- En la línea 5 el tag es 0D → coincide con el tag de la dirección física → acierto → el dato está en d-cache
- El bit de validación es 1 por lo que su contenido está actualizado y por lo tanto válido.

k. Contenido de la PA:

- El contenido de la palabra 0 de la linea 5 de la d-cache es el byte **0x36**

l. Resumen del resultado final.

- La dirección virtual **0x03d4** se corresponde con la dirección física **0x354** cuyo contenido es **0x36**

Intel Core i7

Processor package

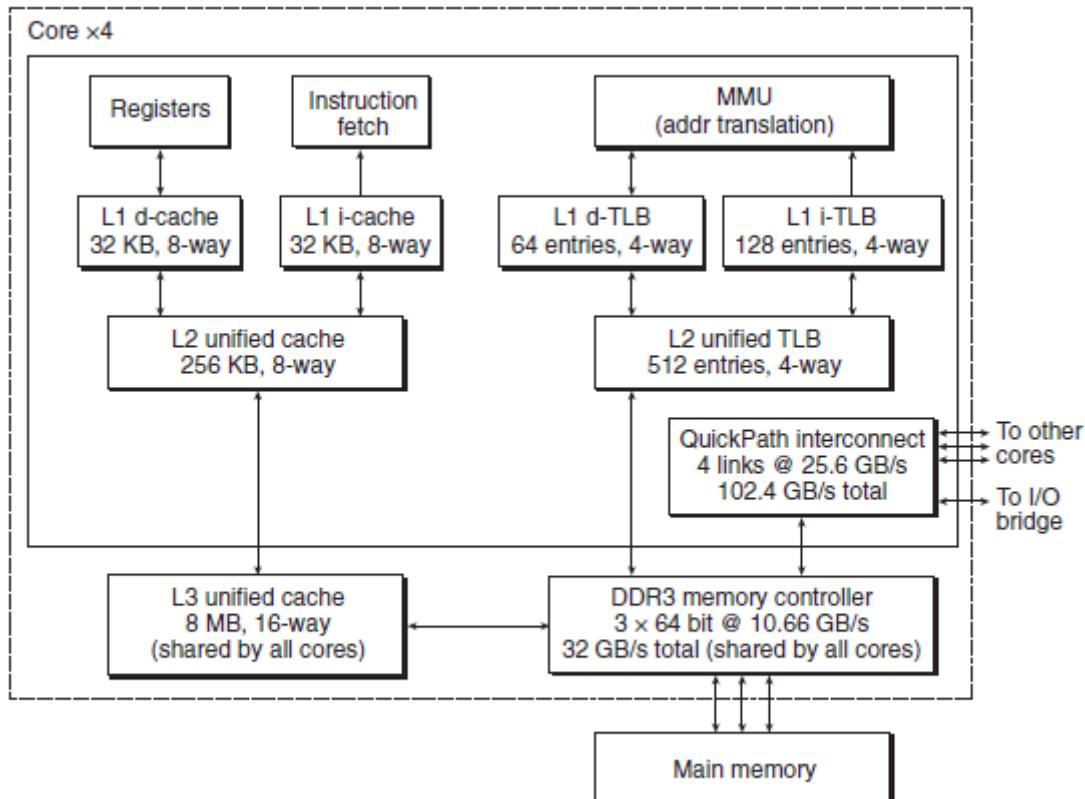


Figure 9.21 The Core i7 memory system.

Figure 65. Memoria Core i7

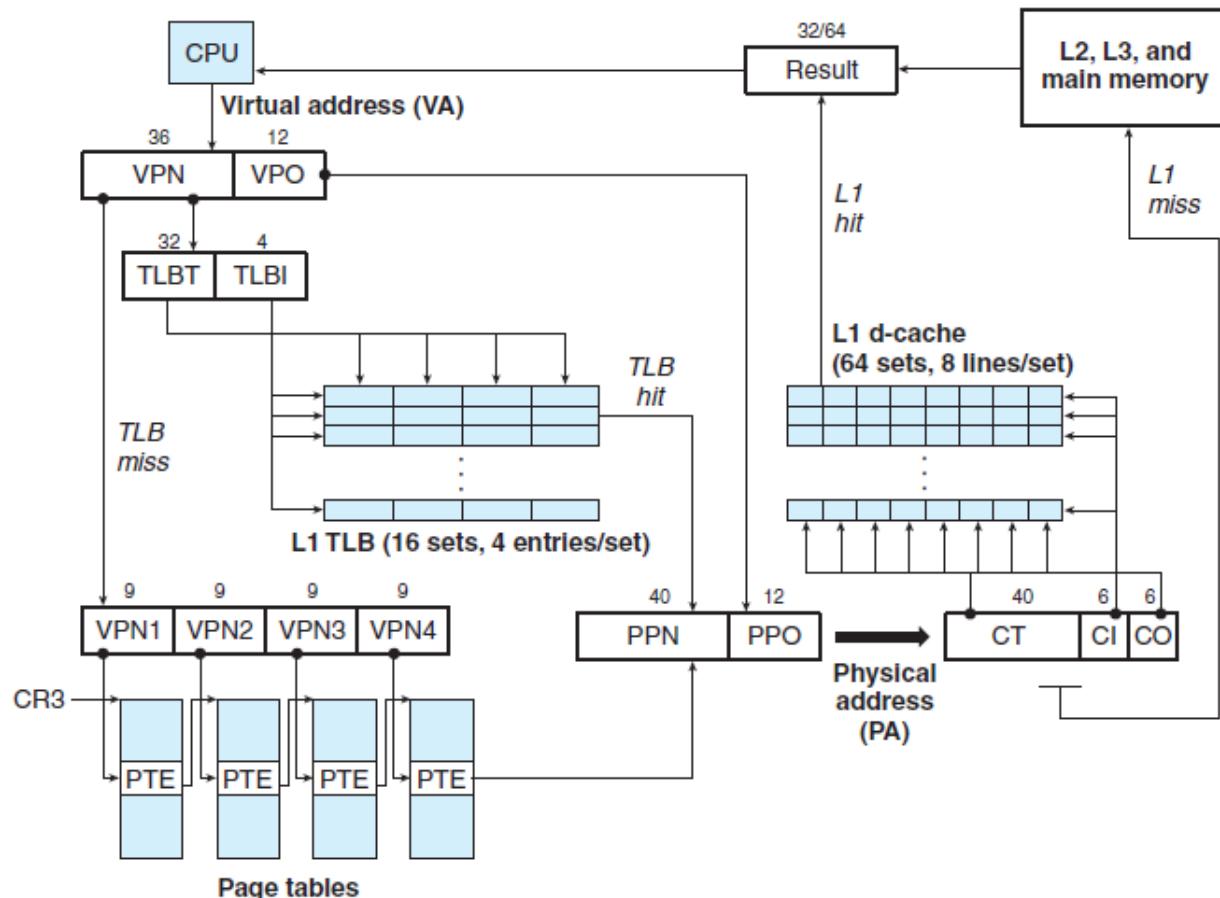
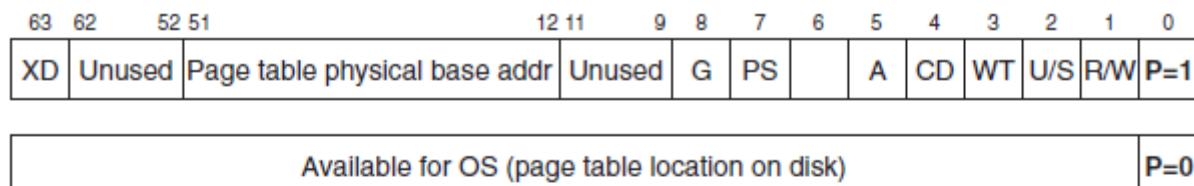


Figure 9.22 Summary of Core i7 address translation. For simplicity, the i-caches, i-TLB, and L2 unified TLB are not shown.

Figure 66. Operación MMU



| Field | Description |
|-----------|---|
| P | Child page table present in physical memory (1) or not (0). |
| R/W | Read-only or read-write access permission for all reachable pages. |
| U/S | User or supervisor (kernel) mode access permission for all reachable pages. |
| WT | Write-through or write-back cache policy for the child page table. |
| CD | Caching disabled or enabled for the child page table. |
| A | Reference bit (set by MMU on reads and writes, cleared by software). |
| PS | Page size either 4 KB or 4 MB (defined for Level 1 PTEs only). |
| Base addr | 40 most significant bits of physical base address of child page table. |
| XD | Disable or enable instruction fetches from all pages reachable from this PTE. |

Figure 9.23 Format of level 1, level 2, and level 3 page table entries. Each entry references a 4 KB child page table.

Figure 67. Formato para las tablas de los tres primeros niveles

| | | | | | | | | | | | | | | | |
|--|--------|-------------------------|----|----|----|--------|---|---|---|---|----|----|-----|-----|-----|
| 63 | 62 | 52 | 51 | 12 | 11 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| XD | Unused | Page physical base addr | | | | Unused | G | 0 | D | A | CD | WT | U/S | R/W | P=1 |
| Available for OS (page table location on disk) | | | | | | | | | | | | | | P=0 | |

| Field | Description |
|-----------|---|
| P | Child page present in physical memory (1) or not (0). |
| R/W | Read-only or read/write access permission for child page. |
| U/S | User or supervisor mode (kernel mode) access permission for child page. |
| WT | Write-through or write-back cache policy for the child page. |
| CD | Cache disabled or enabled. |
| A | Reference bit (set by MMU on reads and writes, cleared by software). |
| D | Dirty bit (set by MMU on writes, cleared by software). |
| G | Global page (don't evict from TLB on task switch). |
| Base addr | 40 most significant bits of physical base address of child page. |
| XD | Disable or enable instruction fetches from the child page. |

Figure 9.24 Format of level 4 page table entries. Each entry references a 4 KB child page.

Figure 68. Formato de la tabla del 4º nivel

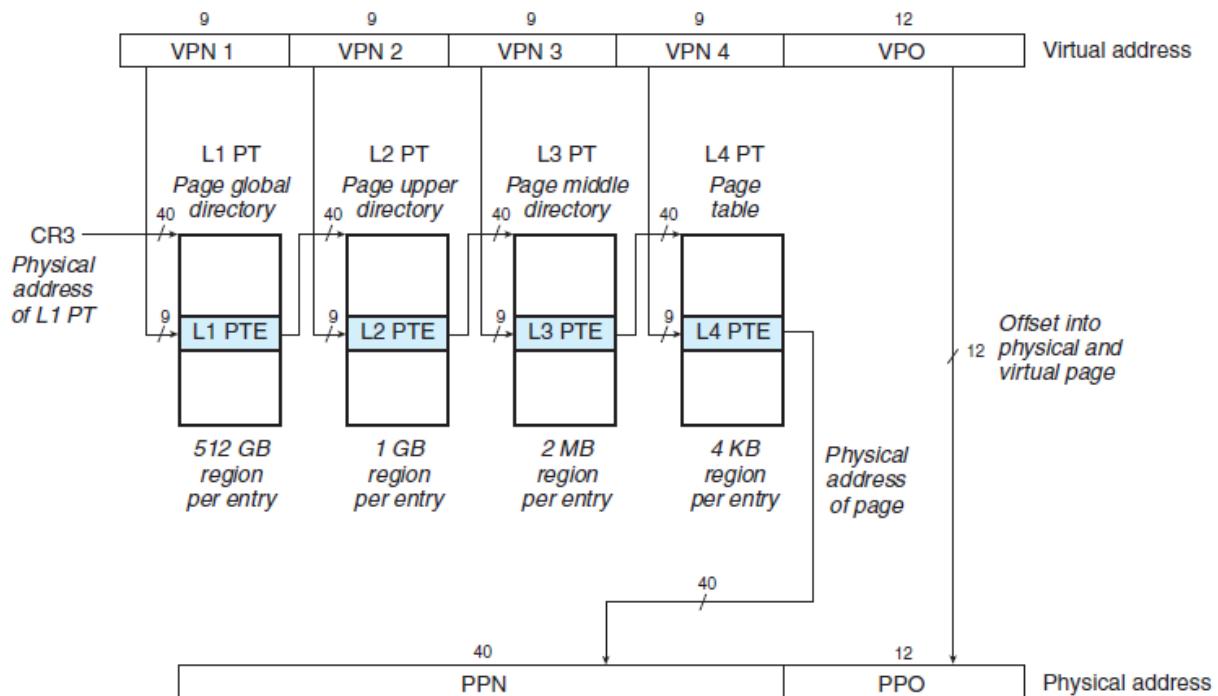


Figure 9.25 Core i7 page table translation. Legend: PT: page table, PTE: page table entry, VPN: virtual page number, VPO: virtual page offset, PPN: physical page number, PPO: physical page offset. The Linux names for the four levels of page tables are also shown.

Figure 69. Linux: 4 niveles

9.5.5. Sistemas Operativos: Gestión de la Memoria

Protección

- Page level protection. HW isolation. Las páginas que gestiona la MMU tiene *bits de control* que indican los permisos de acción, acceso, etc
- Segmentation fault o Protection Fault

Paginación Bajo Demanda

- Paginación bajo demanda
 - Las páginas de un proceso se cargan en memoria únicamente cuando son *demandadas*. No se cargan todas las páginas de un proceso de una tacada.
 - Cuando una página es requerida y no está en la memoria, se genera un *page fault* por parte de la MMU y el SO se encargará de cargar la página requerida.
- Principio de localidad
 - Un proceso en un momento dado tiene en memoria únicamente las páginas que están utilizándose o con las que tienen probabilidad alta de ser utilizadas.
 - Furthermore, time is saved because unused pages **are not swapped** in and out of memory, ya que ese trozo puede ser requerido con immediatez.

Reemplazo

- Reemplazar una página
 - Dilema: ¿qué página extraigo de la memoria? Algoritmos de reemplazo → Least Recently Used LRU, First Input Output (FIFO)
 - La política de reemplazo la gestiona el SO.

VM Tool

- La Memoria Virtual es una herramienta para:
 - que la M. Principal sea una cache del disco
 - Gestionar la Memoria
 - Simplifying linking: mezcla con direcciones independientes de la dirección física final
 - Simplifying loading: se carga bajo demanda las páginas requeridas
 - Simplifying sharing: procesos (librerías) que son compartidos.
 - Simplifying memory allocation: En memoria virtual el SO o compilador distribuye los segmentos de forma contigua y luego está la flexibilidad de ubicarlos arbitrariamente en memoria física.
 - Proteger los segmentos: control con los bits sup(supervisor),read,write

III Ejercicios de Teoría

Chapter 10. Ejercicios

10.1. Lista de Ejercicios Mínima

- El siguiente listado selecciona un número de ejercicios mínimo que hay que realizar:

1. Arquitectura de Von Neumann: 1.1,1.2,1.3,1.4,1.6,2.1,2.2,2.3
2. Representación de los datos: 2,3,4,5
3. Operaciones Aritméticas: 2,5,8,10
4. Operaciones Lógicas: 1,2,3
5. Representación de las instrucciones: 1,2,5,7,9,15,16
6. Programación en Ensamblador: 1.1,2.1,2,2

10.2. Arquitectura von Neumann

10.2.1. Computadoras: IAS, ENIAC, ..

1. You are to write an IAS program to compute the results of the following equation. $Y = \text{Sum}\{X\}$ para $X=1$ hasta $X=N$. Assume that the result of the computation does not arithmetic overflow and that X , Y , and N are positive integers with $N > 1$. Note: The IAS did not have assembly language only machine language.
 - a. Use the equation **Sum(Y)= N(N+1)/2** when writing the IAS program. Comentar los conceptos de: datos, instrucciones, memoria principal, registros, secciones, operación, operando, campo de operaciones, campo de operando, dirección de memoria, contenido de memoria, referencia al operando, operando implícito, direccionamiento del operando, direccionamiento directo del operando, código binario, código hexadecimal, código de operación, código del campo de operando. Ejecutar el programa fuente con el emulador IASSIm.

- Desarrollo:

```
; 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
```

; El algoritmo que seguimos es : primero la suma **N+1**, a continuación la multiplicación **N(N+1)** y finalmente la división. Cada resultado se guarda en la memoria principal si fuera necesario.

; Describimos el programa en 4 lenguajes: lenguaje ensamblador iassim, lenguaje máquina binario, lenguaje RTL y lenguaje ensamblador WStalling

```
; SECCION DE INSTRUCCIONES
S(x)->Ac+ n ; 01 n ; AC <- M[n] ; LOAD M(n)
S(x)->Ah+ uno ; 05 uno ; AC <- AC+1 ; ADD M(uno)
At->S(x) y ; 11 y ; M[y] <- AC ; STORE M(y)
S(x)->R y ; 09 y ; AR <- M[y] ; LOAD MQ,M(y)
S(x)*R->A n ; 0B n ; AC:AR <- AR*M[n] ; MUL M(n)
R->A ; 0A ; AC <- AR ; LOAD MQ
A/S(x)->R dos ; 0C 2 ; AR <- AC/2 ; DIV M(dos)
```

```

R->A          ;      0A      ;      AC    <- AR      ;LOAD  MQ
At->S(x) y    ;      11 y    ;      M[y] <- AC      ;STORE y
halt
; como el número de instrucciones es par no es necesaria la directiva
.empty

; 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

```

b.

Do it the “hard way” without using the equation from part (a): $\sum_{i=1}^5 i$

- Desarrollo:

```

; adds up the values n+...+3+2+1(+0) in a loop and stores
; the sum in memory at the location labeled "sum"

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

```

2. On the IAS, what would the machine code instruction look like to load the contents of memory address 2 to the accumulator? How many trips to memory does the CPU need to make to complete this instruction during the instruction cycle?

- Desarrollo:

- 0x01002: Código de Operación 01 y referencia al operando 002
- Dos accesos a la memoria principal: captura de la instrucción y captura del operando

3. On the IAS, describe in English the process that the CPU must undertake to read a value from memory and to write a value to memory in terms of what is put into the MAR, MBR, address bus, data bus, and control bus.

- Desarrollo:

Lectura

MAR <- address
 Address Bus <- MAR
 Control Bus <- Read
 Data Bus <- Data
 MBR <- Data Bus

Escritura

MAR <- address
 Data Bus <- MBR
 Control Bus <- Write

- A esta descripción del flujo de información se le denomina ruta de datos de la CPU.

4. Given the memory contents of the IAS computer shown below,

Address Contents

08A 010FA210FB
 08B 010FA0F08D
 08C 020FA210FB

- show the assembly language code for the program, starting at address 08A. Explain what this program does.
- Desarrollo:

| Address | Contents | RTL | | Instructions | |
|---------|------------|------------|-----------------------|--------------|-------------------|
| 08A | 010FA210FB | AC←M[0FA] | M[0FB]←AC | LOAD M[0FA] | STORE M[0FB] |
| 08B | 010FA0F08D | AC←M[0FA] | AC>0:PC ← 0x08D(0:19) | LOAD M[0FA] | JMP +M[08D(0:19)] |
| 08C | 020FA210FB | AC←-M[0FA] | M[0FB]←AC | LOAD -M[0FA] | STORE M[0FB] |

- A este proceso inverso a ensamblar (lenguaje ensamblador → código binario) se le denomina desensamblar (código binario → lenguaje ensamblador)
- El programa realiza la siguiente función:
 - Si el contenido de 0x0FA es positivo copia el contenido de memoria de la posición 0x0FA a la posición 0xFB y salta a la posición 0x08D, dejando en el acumulador el contenido de 0xFA. Si el contenido de 0x0FA es negativo copia el contenido de memoria de la posición 0x0FA a la posición 0xFB cambiado de signo y salta a la posición 0x08D, dejando en el acumulador el contenido de 0xFA en positivo, es decir, el módulo.

5. Indicate the width, in bits, of each data path (e.g., between AC and ALU) of IAS microarchitecture.

- Desarrollo:
 - AC, AR y MBR 40 bits
 - IBR 20 bits
 - MAR y PC 12 bits
 - IR 8 bits

6. Considerar una computadora con las siguientes instrucciones:

| Código de Operación | Descripción |
|---------------------|-----------------------|
| 0001 | Load AC from memory |
| 0010 | Store AC to memory |
| 0101 | Add to AC from memory |

a. ¿Cuál es el formato de instrucciones? ¿ Cuál es el tamaño de los registros contador de programa y registro de instrucciones? ¿Cuál es el tamaño del bus de datos y del bus de direcciones?

b. Describir las fases de los ciclos de instrucción de la CPU al ejecutar dicho programa teniendo en cuenta la siguiente figura:

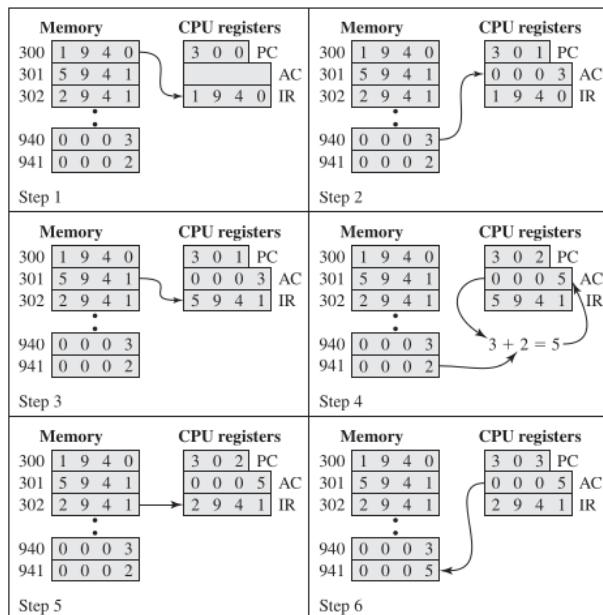


Figure 70. Ciclos de Instrucción

7. The ENIAC was a **decimal machine**, where a register was represented by a ring of 10 vacuum tubes. At any time, only one vacuum tube was in the ON state, representing one of the 10 digits. Assuming that ENIAC had the capability to have multiple vacuum tubes in the ON and OFF state simultaneously, why is this representation “wasteful” and what range of integer values could we represent using the 10 vacuum tubes?

- Desarrollo: Con 10 tubos únicamente podemos representar los dígitos 0-9

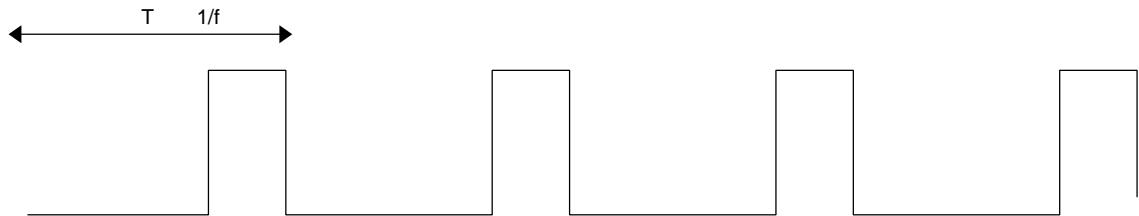
8. A benchmark program is run on a 40 MHz processor. The executed program consists of 100,000 instruction executions, with the following instruction mix and clock cycle count:

| Instruction_Type | Instruction_Count | Cycles_per_Instruction |
|--------------------|-------------------|------------------------|
| Integer_arithmetic | 45,000 | 1 |
| data_transfer | 32,000 | 2 |
| Floating_point | 15,000 | 2 |
| Control_transfer | 8000 | 2 |

- Determine the effective CPI, MIPS rate, and execution time for this program.

- Desarrollo:

- Reloj de la CPU



- $T=1/f = 25\text{ns}$: ciclo del reloj de la CPU: duración mínima de una microoperación.
- CPI: ciclos por instrucción: valor medio $= 1*(45/100)+2*(32/100)+2*(15/100)+2*(8/100)=0,45+0,64+0,30+0,16=1.55 \text{ cpi}$
- MIPS: Millones de Inst. por seg: $(1/\text{CPI})(\text{inst/ciclo}) * F_{\text{clock}}(\text{ciclos/seg}) * 10^{-6} = (1/1.55) * 40 * 10^6 * 10^{-6} = 25.8 \text{ mips}$
- $T=(1/\text{MIPS})(\text{seg/millones de instr}) * 100.000 * 10^{-6} = 3.87\text{ms}$

10.2.2. Interconexión CPU-Memoria

1. Consider a hypothetical microprocessor generating a 16-bit address (for example, assume that the program counter and the address registers are 16 bits wide) and having a 16-bit data bus.
 - a. What is the maximum memory address space that the processor can access directly if it is connected to a “16-bit memory”?
 - b. What is the maximum memory address space that the processor can access directly if it is connected to an “8-bit memory”?
 - c. What architectural features will allow this microprocessor to access a separate “I/O space”? many 8-bit I/O ports can the microprocessor support? How many 16-bit I/O ports? Explain.
 - Tener en cuenta que:
 - Espacio de direcciones: conjunto de direcciones de un mismo bus de direcciones. La capacidad se expresa en BYTES.
 - El Espacio Memoria Principal y el Espacio Controlador E/S son espacios **diferentes**. Comparten el **mismo bus de direcciones** del bus del sistema pero hay una señal de control que activa la conexión con la memoria principal o con el controlador E/S.
 - "16 bit memory": 16 bits word size . Data bus de 16 bits
 - "8 bit memory": 8 bits word size . Data bus de 16 bits
 - I/O port number: numero de puertos del controlador E/S. Cada puerto para un periférico. Se diferencia el puerto de entrada del puerto de salida. 8 bit I/O port es un puerto con un data buffer de 8 bits y un 16 bit I/O port es un puerto con un data buffer de 16 bits.
 - Desarrollo:
 - Dibujar el esquema de buses que visualice:
 - las interconexiones entre periféricos, puertos, controlador E/S, memoria principal, CPU, buses indicando el modo de operación con cada uno de los buses.

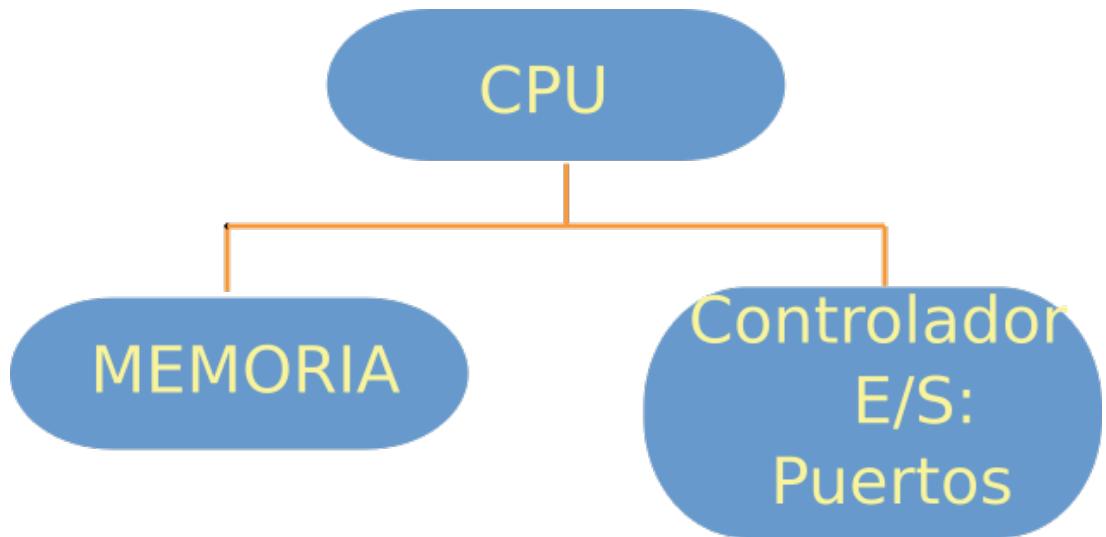


Figure 71. Esquemas Buses

- a) 2^{16} Bytes. 64KB. En el bus de datos se transfieren datos de dos bytes.
 - b) 2^{16} Bytes. 64KB. En el bus de datos se transfieren datos de un byte.
 - c) En el bus del sistema hace falta una señal de control : señal I/O
 - d) $2^8 =$ Direcciónamiento de 256 puertos de entrada y 256 puertos de salida en el controlador E/S independientemente del tamaño del buffer I/O si es de 8 bits o 16 bits.
2. Consider a 32-bit microprocessor, with a 16-bit external data bus, driven by an 8-MHz input clock. Assume that this microprocessor has a bus cycle whose minimum duration equals four input clock cycles. What is the maximum data transfer rate across the bus that this microprocessor can sustain, in bytes/s? To increase its performance, would it be better to make its external data bus 32 bits or to double the external clock frequency supplied to the microprocessor? State any other assumptions you make, and explain. Hint: Determine the number of bytes that can be transferred per bus cycle.
- Desarrollo:
 - 32-bit CPU : tamaño de los registros internos de la CPU. Bus de datos local (interno) de la CPU
 - 16-bit external data bus: bus de datos del sistema
 - CPU input clock: 8MHz
 - bus cycle: ciclo del bus del sistema: duración 4 veces el de la CPU : 2 MHz.
 - a) Data transfer rate: teóricamente número de datos en la secuencia continua de una transferencia cada *bus cycle* durante 1 segundo: $2M$ Transferencias/s. Cada transferencia el bus de datos transfiere 16 bits, es decir, $2 \text{ bytes} = 2\text{M/s} * 2\text{B} = 4\text{MB/s}$
 - b) Doblar el ancho del bus de datos, dobla el ancho de banda $\rightarrow 8\text{MB/s}$
 - c) Doblar la frecuencia de reloj reduce proporcionalmente el ciclo de bus y dobla el ancho de banda $\rightarrow 8\text{MB/s}$
3. Consider two microprocessors having 8- and 16-bit-wide external data buses, respectively. The two processors are identical otherwise and their bus cycles take just as long.
- a. Suppose all instructions and operands are two bytes long. By what factor do the maximum data transfer rates differ?
 - b. Repeat assuming that half of the operands and instructions are one byte long.
- Desarrollo:
 - a) CPU1 de 8 bits tiene un ancho de banda mitad (50%) respecto de CPU2 de 16 bits
 - b1) CPU1: 50% de 2 bytes a 2 ciclos de bus por cada 2 bytes y el otro 50% de 1 byte en 1 ciclo por byte $= 2\text{ciclos} * 50\% + 1\text{ciclo} * 50\% = 1.5\text{ciclos}$

- b2) CPU2: 50% de 2 bytes a 1 ciclo de bus por cada 2 bytes y el otro 50% de 1 byte a 1 ciclo de bus (unicamente se puede acceder a una instrucción o un dato en cada ciclo de bus)= $1\text{ciclo} \cdot 50\% + 1\text{ciclos} \cdot 50\% = 0.5 + 0.5 = 1\text{ciclos}$
 - b) según b1 y b2 la CPU1 tiene un ancho de banda 150% menor que la CPU2, es decir, el 66.6% del CPU2.
4. A microprocessor has an increment memory direct instruction, which adds 1 to the value in a memory location. The instruction has five stages: fetch opcode (four bus clock cycles), fetch operand address (three cycles), fetch operand (three cycles), add 1 to operand (three cycles), and store operand (three cycles).
- a. By what amount (in percent) will the duration of the instruction increase if we have to insert two bus wait states in each memory read and memory write operation?
 - b. Repeat assuming that the increment operation takes 13 cycles instead of 3 cycles.
- Desarrollo:
 - instruction cycle: $4+3+3+3+3= 16\text{ciclos}$
 - a) accesos a memoria en las 3 etapas fetch y en la etapa store → incremento de 2^*4 ciclos de espera → incremento de $8/16$ → un incremento del 50%
 - b) instruction cycle: $4+3+3+13+3= 26\text{ciclos}$ → incremento del ciclo de instrucción en un $8/26$ → incremento en 34%
5. The Intel 8088 microprocessor has a read bus timing similar to that of Figure 3.19, but requires four processor clock cycles. The valid data is on the bus for an amount of time that extends into the fourth processor clock cycle. Assume a processor clock rate of 8 MHz.

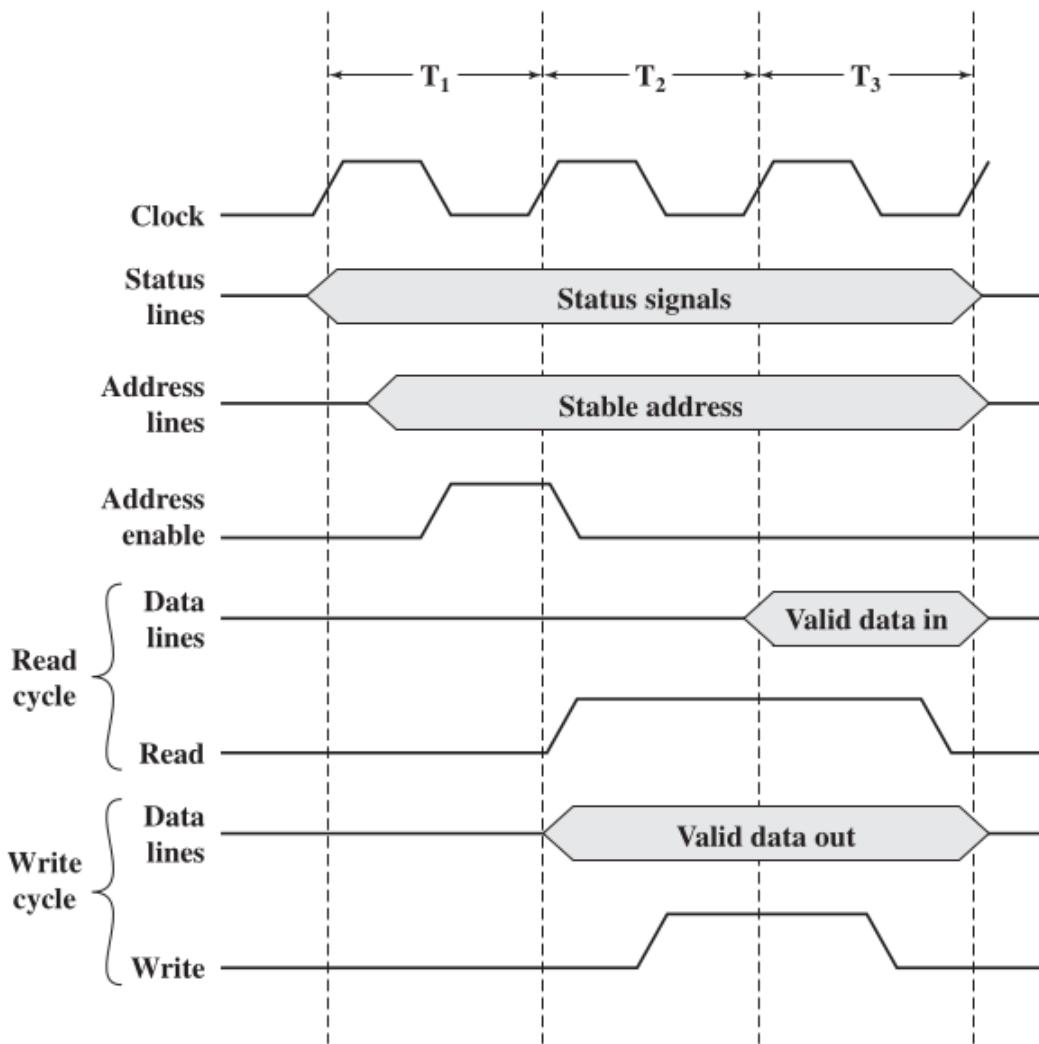


Figure 3.19 Timing of Synchronous Bus Operations

Figure 72. Cronograma de una operación Read/Write de la Memoria Principal

- What is the maximum data transfer rate?
- Repeat but assume the need to insert one wait state per byte transferred.
 - Desarrollo:
 - 8088: bus data: 1 byte
 - read time : 4 cpu cycles
 - data valid: 1 processor clock cycle. El cuarto ciclo del read time.
 - cpu clock: 8MHz
 - a) 4 ciclos por transferencia. $8\text{MHz}/4\text{ciclos} = 2\text{MT/s} = 1 \text{ byte por transferencia} \rightarrow 2\text{MB/s}$
 - b) cada transferencia está un ciclo sin transferir (4,1) \rightarrow throughput = 4/5 del máximo $\rightarrow (4/5)*2\text{MB/s} \rightarrow 1.6\text{MB/s}$
- The Intel 8086 is a 16-bit processor similar in many ways to the 8-bit 8088. The 8086 uses a 16-bit bus that can transfer 2 bytes at a time, provided that the lower-order byte has an even address. However, the 8086 allows both even- and odd-aligned word operands. If an odd-aligned word is referenced, two memory cycles, each consisting of four bus cycles, are required to transfer the word. Consider an instruction on the 8086 that involves two 16-bit operands. How long does it take to fetch the operands? Give the range of possible answers. Assume a clocking rate of 4 MHz and no wait states
 - Desarrollo:
 - 8086: bus data: 2 bytes

- intel : little endian: el LSB byte se guarda en la dirección menor y el MSB byte en la dirección superior.
- alineación del dato par requiere 1 ciclo de memoria.
- palabras con alineación impar requieren 2 ciclos de memoria. Cada ciclo de memoria son 4 ciclos de bus.
- instrucción de 2 operandos de 2 bytes cada uno. CPU clock de 4MHz → 0.250 microsegundos → 250 ns
- a) los dos operandos tienen alineación par
 - 1 ciclo de memoria cada operando: 2 ciclos de memoria: 8 ciclos de bus → 2 microsegundos
- b) un operando tiene alineación par y el otro impar
 - 1 ciclo de memoria el par y 2 ciclos el impar: 3 ciclos de memoria: 12 ciclos de bus → 3 microsegundos
- c) los dos operandos tienen alineación impar
 - 2 ciclos cada operando: 4 ciclos de memoria: 16 ciclos de bus → 4 microsegundos.

7. Consider a 32-bit microprocessor whose bus cycle is the same duration as that of a 16-bit microprocessor. Assume that, on average, 20% of the operands and instructions are 32 bits long, 40% are 16 bits long, and 40% are only 8 bits long. Calculate the improvement achieved when fetching instructions and operands with the 32-bit microprocessor.

- Desarrollo

- En cada flanco positivo del ciclo de bus se realiza una transferencia entre memoria y CPU. La cpu de 16 bits realiza una transferencia de 2 bytes o menos y el de 32 bits una transferencia de 4 bytes o menos.
- Media ciclos (CPU 16 bits)= $0.2 \times (2 \text{ ciclos para las dos transferencias de 2 bytes cada una}) + 0.4 \times 1 + 0.4 \times 1 = 1.2$ ciclos de media
- Media ciclos (CPU 32 bits)= $0.2 \times 1 + 0.4 \times 1 + 0.4 \times 1 = 1$ ciclo de media
- Mejora de $(1.2 - 1)$ sobre 1.2 = $(1.2 - 1) / 1.2 = 17\%$

10.3. Representación de Datos

1. Representar el número decimal 1197 en las bases:

Divisiones sucesivas por 16
0x4AD

a. octal:

Divisiones sucesivas por 8
0o2255

b. binaria:

Divisiones sucesivas por 2
0b10010101101

c. Representar el número 0x4AD en base binaria y base octal mediante una conversión directa, sin

- calcular su valor.
2. Representar el número -1197 en base binaria y hexadecimal y en formato:
 - a. Signo-magnitud: 0b110010101101 → 0xCAD
 - b. Complemento a 2: 0b101101010011 → 0xB53
 3. Calcular el rango de los números enteros de 8 bits en complemento a 2. ($2^7-1, -2^7$)
 4. Utilizar notación hexadecimal:
 - a. Representar el valor 23 en el formato BCD..
 - En el formato Binary Code Decimal (BCD) cada dígito decimal se expande independientemente en su código binario de 4 bits
 - 2→0010 ; 3→0011 ; 23 → 0010-0011
 - b. The ASCII characters 23
 - 0x32-0x33 → 0011-0010-0011-0011
 5. Escribir el carácter **a** con acento **á** en el código universal UTF-8 (hexadecimal), en el código Unicode(U+XXXX), en ascii extendido e iso-8859-1 consultando el enlace [UTF-8](#), los manuales: man ascii, man iso-8859-1, man 7 utf-8 y el comando **showkey -a** → teclear **á**. Qué código utiliza la dirección URL y qué código utiliza HTML
 - [UTF-8](#) → leer la columna de la derecha con los links a tools, html converter, character sets, etc
 - [UTF-8](#) → 225 U+00E1 C3 A1 → decimal 225, unicode point U+00E1 y unicode hex point 0xC3A1
 - ascii extendido es igual a iso-8849-1 → ascii extendido 0xE1 , utf-8 point code U+00E1, 0xE1 es 225 en decimal → HTML utiliza el código á
 - código utilizado por URL %C3%A1 correspondiente al hexadecimal 0xC3A1 → escribir en la barra URL del navegador y luego copiar y pegar
 - **showkey -a** á → hexadecimal utf-8 code 0xC3A1 -
 - man iso-8859-1 → ascii extendido
 6. For each of the following packed decimal numbers, show the decimal value
 - a. 0111 0011 0000 1001
 - 7309
 - b. 0101 1000 0010
 - 582
 - c. 0100 1010 0110
 - No es posible ya que 1010 corresponde al valor 10 que no tiene un dígito decimal sino dos.
 7. (NO hacer) Another representation of binary integers that is sometimes encountered is ones complement. Positive integers are represented in the same way as sign magnitude. A negative integer is represented by taking the Boolean complement of each bit of the corresponding positive number. Note: Ones complement arithmetic disappeared from hardware in the 1960s, but still survives checksum calculations for the Internet Protocol (IP) and the Transmission Control Protocol (TCP).
 - a. Provide a definition of ones complement numbers using a weighted sum of bits.
 - Poner ejemplos de conversión con n=3 bits
 - 000→111 (luego el cero tiene dos representaciones), +1: 001→ -1:110, +2:010→'-2':101, +1:011→'-3':100
 - positivos con n bits $\sum_{i=0}^{n-1} b_i 2^i$
 - negativos con n bits

- Tenemos en cuenta que complemento a dos = complemento_a_1 + 1
- el complemento a dos con n bits de X se puede calcular como la resta binaria 2^n (en binario) - X : por ejemplo con 3 bits el complemento a dos de +1 es $1000-1=111$
- el complemento a 1 es el complemento a 2 menos 1 → $2^n - X - 1$. Por ejemplo con 3 bits el complemento a uno de +1 $1000-1-1 = 110$

b. What is the range of numbers that can be represented in ones complement with n bits?

- El máximo positivo → 011...1 : 2^n-1
- El máximo negativo → 100...0 : $-(2^n-1)$

8. Representar 0.56789 en binario utilizando multiplicaciones sucesivas

$$\begin{aligned}0.56789 * 2 &= 1.13578 \rightarrow 1, \text{ bit de la posición -1} \\0.13578 * 2 &= 0.27156 \rightarrow 0, \text{ bit de la posición -2} \\0.27156 * 2 &= 0.54312 \rightarrow 0, \text{ bit de la posición -3} \\0.54312 * 2 &= 1.08624 \rightarrow 1, \text{ bit de la posición -4}\end{aligned}$$

9. Representar 0.0625 en binario sin utilizar multiplicaciones sucesivas.

- $0.0625 = M \cdot 2^E$ tal que E es un entero
- $\log_{10}(0.0625) = \log_{10}(M) + E$
- $-4 = \log_{10}(M) + E \rightarrow E = -4$ y $\log_{10}(M) = 0 \rightarrow M = 1$

10. Representar el número real 1234.56789 en base binaria:

- En formato coma fija

Parte Entera: 1234 : 10011010010
 Parte Fracción: 0.56789: 0.100100010110000101
 Número 1234.56789: 10011010010.100100010110000101

- En notación científica: $1.001101001010010001011000010 \cdot 2^{+10}$
- En precisión simple punto flotante:

Campo signo: + : 0
 Campo Exponente (8 bits): $10+127 = 137 = 10001001$
 Campo fracción mantisa (23 bits)= 00110100101001000101100

- En precisión doble punto flotante:

Campo signo: + : 0
 Campo Exponente (11 bits): $10+1023 = 1033 = 10000001001$
 Campo fracción mantisa (52 bits) = 00110100101001000101100001010_0

11. Codificar el número entero 3 en single precision FP

- $3 = 11 = 1.1 \cdot 2^1$
 - S=0, E=1+127=128, Mn=0.1
 - 0-1000-0000-1000-0000-ceros

- No es necesario redondear
 - Resultado= 0x40400000
12. Representar el número natural 123456789 en precisión simple Punto Flotante (IEEE-754)
- $123456789 = 0x075BCD15 = 111-0101-1011-1100-1101-0001-0101 = 1.1101011011100110100010101 \cdot 2^{26}$
 - Redondear= $1.1101011011100110100011 \cdot 2^{26}$
 - Campo Signo= 0
 - Campo Exp= $26+127=153=10011001$
 - Campo fracción Mantisa= 0.1101011011100110100011
 - Resultado 0x4CEB79A3
13. Float Point:
1. Consider a fixed-point representation using decimal digits, in which the implied radix point can be in any position (e.g., to the right of the least significant digit, to the right of the most significant digit, and so on). How many decimal digits are needed to represent the approximations of both Planck's (6.63×10^{-27}) constant and Avogadro's number (6.02×10^{23}) The implied radix point must be in the same position for both numbers.
 - para el número de planck hace falta correr la coma 27 posiciones a la izda más los dos dígitos a la derecha (63) → fracción de 27 dígitos
 - para el número de avogrado hace falta correr la coma 23 posiciones a la dcha más el dígito de la izda (7) → parte entera de 24 dígitos
 - para los dos $29+24=53$ dígitos
 2. Now consider a decimal floating-point format with the exponent stored in a biased representation with a bias of 50. A normalized representation is assumed. How many decimal digits are needed to represent these constants in this floating point format?
 - planck → $0.63 \times 10^{-26} \rightarrow 0.63 \times 10^{-26+50} \rightarrow 0.63 \times 10^{+24}$
 - avogrado → $0.602 \times 10^{24} \rightarrow 0.602 \times 10^{24+50} \rightarrow 0.602 \times 10^{74}$
 - para los dos hacen falta = 3 dígitos fracción y 2 dígitos para el exponente.
14. Any floating-point representation used in a computer can represent only certain real numbers exactly; all others must be approximated. If A_p is the stored value approximating the real value A , then the relative error, r , is expressed as $r=(A-A_p)/A$. Represent the decimal quantity +0.4 in the following floating-point format: base=2 exponent: biased, 4 bits; significand, 7 bits. What is the relative error?
- En binario coma fija
 - $0.4 \times 2 = 0.8 \rightarrow 0$
 - $0.8 \times 2 = 1.6 \rightarrow 1$
 - $0.6 \times 2 = 1.2 \rightarrow 1$
 - $0.2 \times 2 = 0.4 \rightarrow 0$
 - $0.4 \times 2 \rightarrow$ otra vez 0110, luego es un número periódico → 0.0110-0110-0110-período
 - normalizado $1.10-0110-0110\text{-etc } \times 2^{-1} \rightarrow$ se representa la fracción 10-0110-0110-
 - con fracción de 7 bits 1001100
 - Valor del número aproximado $1.1001100 \times 2^{-1} = 110011 \times 2^{-6} = (32+16+2+1) \times 2^{-6} = 51/64 = .796875$
 - Error = $(0.8 - 0.796875) / 0.8 = .003906250 = 0.4\%$
 - En exceso de 4 bits. Con el número excedido el rango es (0,15). El exceso es la mitad de combinaciones $-1=16/2-1=7$, luego se pueden representar los exponentes (-7,8).
15. Representar el número Pi en coma flotante IEEE de simple y doble precisión

- SOLUCION:
- Formato decimal: 3.1415926535897932384626433832795028841968
- Binario Coma fija :

$$11.00100100001111101101010001000101101000110000100011010011000100110001001100010100010110000000110111$$
- Hexadecimal Coma fija: 0011.0010-0100-0011-1111-0110-1010-1000-1000-0101-1010-0011-0000-1000-1101-0011-0011-0111
 - 3.243F6A8885A308D313198A2E03E

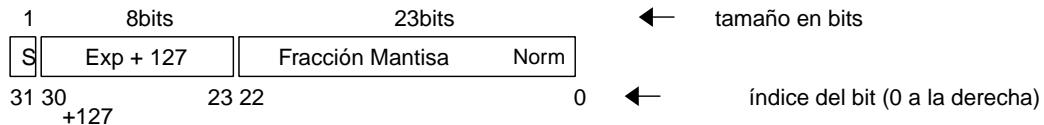


Figure 73. IEEE-754 precisión simple (32 bits)

- $v = s \times 2^e \times m$
 - Notación científica con la mantisa normalizada y su parte fracción truncada a 23 bits y redondeada:
 - $+ (1 + 0.1001001000011111011011) * 2^{+1}$
- Campos:
 - Signo : positivo $\rightarrow 0 \rightarrow 1$ bit
 - Exponente: $+1$
 - Exponente desplazado $+ 127 = +1+127 = 128 \rightarrow 10000000 \rightarrow 8$ bits
 - Mantisa normalizada: $1 + 0.1001001000011111011011$
 - Fracción de la mantisa normalizada: $1001001000011111011011 \rightarrow 23$ bits

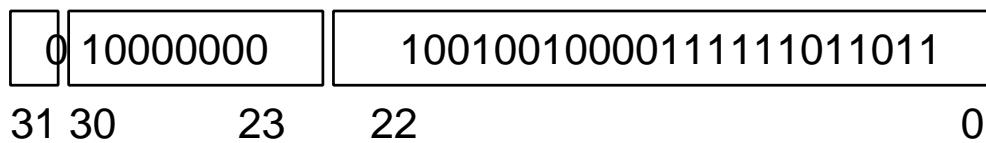


Figure 74. Formato IEEE-754 precisión simple

- Resultado= 0x40490fdb

10.4. Operaciones Aritméticas

1. Sumar en binario puro
 - $10011011 + 00011011$ SOLUCION: = 10110110
 - $0x3A1F + 0xF4E1$ SOLUCION: = 0x12f00
 - $10011011 + 10011011$ SOLUCION: = 100110110
2. Sumar en complemento a 2 : 50+23
 - Realizar las operaciones en código binario SOLUCION: = 0110010 + 0010111 = 01001001

- Realizar las operaciones en código hexadecimal SOLUCION: = $0x32+0x17 = 0x49$
3. Representar el valor -66 en complemento a 2
- SOLUCION: $+66 = 01000010 \rightarrow -66 = 10111101+1 = 10111110$
4. Resta en complemento a 2 : 33-66
- Realizar las operaciones en código binario: SOLUCION: $0100001+10111110=1011111$
 - Realizar las operaciones en código hexadecimal: SOLUCION: $0x21+0xBE=0xDF$
5. Representar en hexadecimal el mayor número en módulo que se puede representar en complemento a 2 con 16 bits
- SOLUCION: $1000-0000-0000-0000 = 0x8000$
6. Sumar en complemento a 2 con 16 bits $0x8000+0x8000$
- SOLUCION: $0x8000+0x8000=0x0000$
7. Restar en binario puro $0110010 - 0010111$
- SOLUCION:

```

0110010  <- minuendo
0010111  <- sustraendo
      1111  <- llevadas
*****
0011011

```

8. Restar en hexadecimal $0x32-0x17$

```

0x32  <- minuendo
0x17  <- sustraendo
      1   <- llevadas
*****
0x1B

```

9. Multiplicación

- ¿A qué equivale en base binaria multiplicar por una potencia de 2 positiva 2^n ? :

 - SOLUCION:
 - en un número real mover la coma n posiciones hacia la dcha
 - en un número entero añadir n ceros a la dcha
 - en un registro es desplazar los bits n posiciones hacia la izda e introducir n ceros por la dcha

- ¿A qué equivale en base binaria multiplicar por una potencia de 2 negativa 2^{-n} ? :

 - SOLUCION:
 - en un número real mover la coma n posiciones hacia la izda
 - en un número entero añadir n ceros a la izda
 - en un registro es desplazar los bits n posiciones hacia la dcha e introducir n ceros por la izda

10. Realizar la multiplicación de los siguientes números naturales:

- $1010*1010$
- $1010*1111$

- SOLUCION:

| | |
|---|---|
| $ \begin{array}{r} 1010 \\ \times 1010 \\ \hline \end{array} $ | $ \begin{array}{r} 1010 \\ \times 1111 \\ \hline \end{array} $ |
| ***** | |
| $ \begin{array}{r} 0000 \\ 1010 \\ 0000 \\ 1010 \\ \hline \end{array} $ | $ \begin{array}{r} 1010 \\ 1010 \\ 1010 \\ 1010 \\ \hline \end{array} $ |
| ***** | |
| $ \begin{array}{r} 1100100 \end{array} $ | $ \begin{array}{r} 10010110 \end{array} $ |

10.5. Operaciones Lógicas

- Realizar las operaciones lógicas \tilde{A} , $\tilde{A}+1$, $A+B$, $A \cdot B$, $A \oplus B$ siendo $A=10101010$ y $B=11110000$
 - SOLUCION:
 - $A+B \rightarrow A|B \rightarrow AVB$
 - $A \cdot B \rightarrow A\&B \rightarrow A\wedge B$
 - $\tilde{A} = 01010101$
 - $\tilde{A}+1 = 01010110$
 - $A+B = 11111010$
 - $A \cdot B = 10100000$
 - $A \oplus B = 01011010$
- Dado un operando de 20 bits, indicar la operación lógica a realizar para: (expresar la operación con los operandos en código hexadecimal)
 - Set el bit 7 (posición 7^a)
 - SOLUCION: 20 bits son 5 dígitos hex → Operando | 0x00080
 - Clear el bit 15
 - SOLUCION: Operando & 0xF7FFF
 - Toggle el bit 19
 - SOLUCION: Operando \oplus 0x8000
 - Set toda la palabra
 - SOLUCION: Operando | 0xFFFF
 - Clear toda la palabra
 - SOLUCION: Operando \oplus Operando ó Operando & 0x00000
- Dadas las operaciones lógicas SAR X,n (shift arithmetic right X, n), SLR X,n (shift logic right X,n) , SAL X,n (shift arithmetic left X, n) y SLL X,n (shift logic left X, n) donde X es el operando, xxR significa derecha, xxL significa izquierda y n es el número de posiciones a desplazar. Realizar las siguientes operaciones con el operando A=10101010: SAR A,4, SLR A,4, SAL A, 4 y SLL A,4 de forma manual, mediante un programa en lenguaje C y también mediante el depurador GDB.
 - $SAR A,4 = 11111010$
 - $SLR A,4 = 00001010$
 - $SAL A,4 = 10101111$

- SLL A,4 = 10100000

- op_log.c

```
#include <stdio.h>

void main(void)
{
    unsigned char x = 0b01010101, y = 0b10101011, opor, opand, opxor, shra,
    shr1;
    printf("Operaciones Lógicas BITWISE\n");
    opor = x | y;
    opand = x & y;
    opxor = x ^ y;
    printf("Operación x OR y = 0x%x\n", opor);
    printf("Operación x AND y = 0x%x\n", opand);
    printf("Operación x XOR y = 0x%x\n", opxor);
    shra = (char)y >> 4;
    shr1 = y >> 4;
    printf("Operación shift right arith 0x%X\n", shra);
    printf("Operación shift right logic 0x%X\n", shr1);
}
```

```
/*
(gdb) p /x (y >> 4)
$17 = 0xa
(gdb) set y = y >> 4
(gdb) p y
$18 = 10 '\n'
(gdb) p /x y
$19 = 0xa
*/
```

4. Realizar manualmente la multiplicación $A \cdot 2^2$ y $A \cdot 2^{-2}$ donde $A=10101010$ primero aritméticamente y después mediante operaciones lógicas.

- $A \cdot 2^2 = 1010101000$
- $A \cdot 2^{-2} = 101010.10$
- Con operaciones lógicas la operación $A \cdot 2^2$
 - Doblar el tamaño de A → D:A ← 0000000001010101
 - desplazar SLL D:A,2 → D:A ← 0000000101010100

10.6. Representación de las Instrucciones

1. Sea un computador con palabras de 32 bits. La CPU tiene 64 instrucciones diferentes (operaciones) de un operando, 32 registros de propósito general de 32 bits y posibilidad de direccionamiento directo a registro (el campo de operando es directamente el registro) o indirecto con desplazamiento a registro-base. a) Diseñar el formato de instrucción para este computador. Se debe especificar un registro para las direcciones de memoria y un valor del desplazamiento, además del modo de direccionamiento y código de operación. b) ¿Cuál es el máximo valor del desplazamiento (el desplaz. es un número en C2)?

◦ SOLUCION

a. Formato

Palabra de 32 bits → Registros de propósito general de 32 bits.

Formato de instrucciones con una estructura en 4 campos: código de operación, modo de direccionamiento, campo de operando (registro o registro con desplazamiento)

1º Campo: código de operación: 64 instrucciones : $2^6 \rightarrow 6$ bits

2º Campo: modo de direccionamiento: 2 tipos: $2^1 \rightarrow 1$ bit

3º Campo: registro: 32 registros: $2^5 \rightarrow 5$ bits

4º Campo: Desplazamiento nº entero: $(32-(6+1+5))$ bits → 20 bits



b. Desplazamiento de 20 bits

- En complemento a 2: El valor Positivo máximo 0111-1111-1111-1111-1111 de valor $2^{19}-1$ y el valor Negativo mínimo 1000-0000-0000-0000-0000 que cambiado de signo es el 0-1000-0000-0000-0000 de valor $+2^{19}$, por lo que el rango es $[+2^{19}-1, -2^{19}] \rightarrow [+524287, -524288]$

2. Un computador con palabras de 24 bits posee 16 instrucciones diferentes de un operando, 8 registros de propósito general, y 3 modos de direccionamiento (directo a registro, indirecto con registro e indirecto con desplazamiento a registro-base)

a. Diseñar un formato de instrucción para este computador. Debe especificar el código de operación, el modo de direccionamiento, un registro y un desplazamiento.

◦ SOLUCION:

Word Size = 24 → Registros de Propósito general de 24 bits y el registro de instrucción IR también de 24 bits

4 campos en el formato de instrucción:

+



código de operación- - Registro - Desplazamiento

1º Campo: código de operación: 16 instrucciones : $2^4 \rightarrow 4$ bits

2º Campo: modo de direccionamiento: 3 tipos: $2^2 \rightarrow 2$ bit : para 3 tipos (Directo Reg, Indirecto Reg e Indirecto RegyDesp)

3º Campo: registro: 8 registros: $2^3 \rightarrow 3$ bits

4º Campo: Desplazamiento nº entero: $(24-(4+2+3))$ bits $\rightarrow 15$ bits

b. ¿Cuál es el rango de valores del desplazamiento en magnitud?, ¿y en C2?

- Formato Magnitud: Mínimo el cero y el máximo 111-1111-1111-1111 = $2^{15}-1 = 32768$
- Complemento a 2: Positivo máximo 0111-1111-1111-1111 de valor $2^{14}-1$ y Negativo mínimo 100-0000-0000-0000 que cambiado de signo es el 0100-0000-0000-0000 de valor $+2^{14}$, por lo que el rango es $[+2^{14}-1, -2^{14}] \rightarrow [+16383, -16384]$

3. Un computador tiene un formato de instrucción de 11 bits donde el campo de operando es de 4 bits. ¿Es posible codificar en este formato 5 instrucciones de dos operandos, 45 de un operando y 48 sin operando?. Justificar la respuesta.

◦ SOLUCION

◦ 3 tipos de formatos

- Tipo 1: campo tipo - Cod. Op. - Op1 - Op2
 - 2 bits - x bits - 4 bits - 4 bits $\rightarrow x=11-(2+4+4)= 1$ bit \rightarrow Máximo de 2 instrucciones < 5 instrucciones \rightarrow No es posible
 - Tipo 2: campo tipo - Cod. Op. - Op
 - 2 bits - y bits - 4 bits $\rightarrow y=11-(2+4)= 5$ bits \rightarrow Máximo de 32 instrucciones < 45 instrucciones \rightarrow No es posible
 - Tipo 3: campo tipo - Cod. Op.
 - 2 bits - z bits \rightarrow 48 instrucciones
- Alternativa: 3 Registros IR , uno para cada tipo
- Sin campo de tipo : $5+45+48=98$ instrucciones $\rightarrow 2^7 \rightarrow 7$ bits \rightarrow la instrucción tipo 1 ocuparía $7+4+4=15$ bits > 11 \rightarrow No es posible

4. Un computador de 16 bits de ancho de palabra (instrucciones, palabra de memoria, registros) y 8 registros, tiene el siguiente repertorio de instrucciones:

- 14 instrucciones de referencia de un solo operando en memoria, con direccionamiento directo e indirecto de memoria
- 31 instrucciones con dos operandos con los modos de direccionamiento directo e indirecto de registro.
- 32 instrucciones sin operando explícito.
 - a. Especificar la codificación de las instrucciones.
 - b. Especificar la zona de memoria alcanzable en cada tipo de direccionamiento y rango posible de valores de los operandos (en C'2).

◦ SOLUCION

- Word Size = 16 \rightarrow Registros de Propósito general de 16 bits
- Repertorio con 3 tipos de formatos
- 1º Tipo: Tipo-Cod.Op.-Modo Direc-Op1 \rightarrow 14 instrucciones (2^4), Bits:2-4-1-x $\rightarrow x=16-(2+4+1)=9$ bits
- 2º Tipo: Tipo-Cod.Op.-Modo Direc1-Op1-Modo Direc2-Op2 \rightarrow 31 instrucciones (2^5) Bits:2-5-1-x-1-x $\rightarrow x=(16-(2+5+2))/2=3$ bits

- 3º Tipo: Tipo-Cod.Op. → 32 instrucciones (2^5) Bits:2-5 → 7 bits ocupados de los 16.

5. Un computador basado en el procesador de Motorola M68000 presenta los siguientes contenidos en registro y memoria:

| REGISTROS | | MEMORIA | |
|-----------|-----------|-----------|-----------|
| Registro | Contenido | Dirección | Contenido |
| A1 | 100 | 99 | 104 |
| A2 | 2 | 100 | 108 |
| | | 101 | 106 |
| | | 102 | 107 |
| ... | ... | ... | ... |
| | | 199 | 100 |
| | | 200 | 34 |
| | | 201 | 96 |
| | | 202 | 201 |

- Si el contenido del desplazamiento de la instrucción en ejecución es desp=99 ¿Cuál sería el valor del operando (de tamaño byte) con los siguientes modos de direccionamientos?.

- i. Directo de memoria o absoluto (dirección = desplazamiento).
- ii. Directo de registro con A1.
- iii. Indirecto de registro con A1.
- iv. Indirecto con desplazamiento con registro base A1
- v. Indirecto con desplazamiento con registro base A2.
- vi. Indirecto con desplazamiento con registro base A1 e indexado con A2.
- vii. Indirecto de registro con predecremento con A1.

- SOLUCION:

Directo de memoria o absoluto (dirección = desplazamiento) -> $M[99]=104$

NOTA: El linker resuelve la dirección absoluta del operando especificando un desplazamiento respecto de un registro que apunta al inicio del segmento o sección de instrucciones. De esta forma el campo de operando es más corto que poniendo la dirección absoluta. En este ejemplo suponemos que la dirección base del segmento es cero.

Directo de registro con A1. -> $R[A1]=100$

Indirecto de registro con A1. -> $M[A1]=M[100]=108$

Indirecto con desplazamiento con registro base A1 -> $M[A1+99]=M[199]=100$

Indirecto con desplazamiento con registro base A2.-> $M[A2+99]=M[101]=106$

Indirecto con desplazamiento con registro base A1 e indexado con A2.->

$M[A1+99+A2]=M[100+99+2]=M[201]=96$

Indirecto de registro con predecremento con A1.-> $M[A1-1]=M[100-1]=104$

6. Un computador presenta los siguientes contenidos de los registros y la memoria:

| REGISTROS | | MEMORIA | |
|-----------|-----------|-----------|-----------|
| Registro | Contenido | Dirección | Contenido |
| R1 | 99 | 96 | 100 |
| R2 | 6 | 97 | 102 |
| | | 98 | 101 |
| | | 99 | 104 |
| | | 100 | 108 |
| | | 101 | 106 |
| | | 102 | 107 |
| | | 103 | 109 |
| | | 104 | 110 |

- Si el contenido del desplazamiento de la instrucción en ejecución es 96 ¿Cuál sería el valor del operando con los siguientes direccionamientos?

- a) Directo de memoria (dir = desp). $\rightarrow M[96] = 100$
- b) Indirecto de memoria (dir memoria = desp). $\rightarrow M[M[96]] = M[100] = 108$
- c) Directo de registro con R1 $\rightarrow R1 = 99$
- d) Indirecto de registro con R1. $\rightarrow M[R1] = M[99] = 104$
- e) Indirecto con desplazamiento con registro base R2 $\rightarrow M[R2+96] = M[6+96] = M[102] = 107$

7. Se tiene un computador con un ancho de palabra de 32 bits y con un banco de registros de 32 registros de 32 bits. El computador tiene 64 instrucciones diferentes y los siguientes modos de direccionamiento: directo de memoria, indirecto de memoria e indirecto con desplazamiento a registro-base.

- a. Diseñar los dos formatos de las instrucciones de dos operandos sabiendo que siempre un operando está en memoria y otro en registro.

▪ SOLUCION:

- CodOp/Modo/Memoria_fuente/Registro_destino
 - CodOp=2⁶=64 instrucciones
 - Modo: directo o indirecto: 2¹
 - Memoria=x bits de direcciones
 - Reg=2⁵=32 registros \rightarrow 5 bits
 - Tota IRI=32 bits \rightarrow Memoria=32-(6+1+5)=32-12=20 bits
- CodOp/Desplazamiento-Registro_fuente/Registro_destino
 - CodOp=2⁶=64 instrucciones
 - Desplazamiento= x bits de direcciones
 - Reg_base=2⁵=32 registros \rightarrow 5 bits
 - Registro=2⁵=32 registros \rightarrow 5 bits
 - Total=32 bits \rightarrow Desplazamiento=32-(6+5+5)=16 bits

- b. Si cada dirección de memoria especifica un byte ¿qué zona de memoria se puede acceder con cada uno de los modos de direccionamiento?

- SOLUCION

directo memoria -> campo de 20 bits
 Rango: 0 hasta ($2^{20}-1$) ó 0-1048575 ó 0-0xFFFF

indirecto memoria
 El contenido de una palabra de memoria son 32 bits
 Rango: 0 hasta ($2^{32}-1$) ó 0-4294967296 ó 0-0xFFFFFFFF

indirecto con desplazamiento a registro base
 El rango del desplazamiento es 0 hasta ($2^{16}-1$) ó 0-6553 ó 0-0xFFFF

8. Consideremos cuatro arquitectura de procesador: acumulador, pila, memoria-memoria y registro-registro con 16 registros. Para las cuatro arquitectura se tienen los siguientes datos comunes:

- El código de operación es siempre 1 byte
 - Todas las direcciones de memoria son 2 bytes
 - Todos los datos son 4 bytes
 - Todas las instrucciones tienen una longitud igual a un numero entero de bytes
- a. Escribir de forma genérica los programas en lenguaje ensamblador de cada una de las arquitecturas para realizar la siguiente operación; A=B+C. Para cada programa, calcular el tráfico con memoria y el tamaño del código. ¿Cuál es mas eficiente?.

| Stack | Acumulator | Register/Memor y | Load/Store |
|---------------|----------------|--------------------|---------------------|
| Push C | Load C | Load R1,C | Load R1,C |
| Push B | Add B | Add R3,R1,B | Load R2,B |
| Add | Store A | Store R3,A | Add R3,R1,R2 |
| Pop A | | | Store R3,A |

- Tráfico con la memoria de la arquitectura Stack

La 1^a instrucción : lectura instrucción push C, lectura C, escritura C -> 3
 La 2^a instrucción : lectura instrucción push B, lectura B, escritura B -> 3
 La 3^a instrucción : lectura instrucción add y lectura de dos datos de la pila y escritura en la pila del resultado -> 4
 La 2^a instrucción : lectura instrucción pop A y escritura del resultado -> 2
 Total: 12 accesos a memoria

- Tráfico con la memoria de la arquitectura Acumulator

La 1^a instrucción : lectura instrucción y lectura dato
 La 2^a instrucción : lectura instrucción y lectura dato
 La 3^a instrucción : lectura instrucción y escritura dato

Total: 6 accesos a memoria

- b. Escribir los cuatro programas ensamblador para la siguiente secuencia de operaciones A=B+C ; B=A+C ; D=A-B Calcular el tráfico con memoria y el tamaño del código. ¿Cuál es más eficiente?.
9. Considerando que en un procesador cada captura de instrucción y cada acceso a un operando consumen un ciclo de reloj y teniendo en cuenta para 3 programas los siguientes datos en millones de referencias.

| | TEX | Spice | C |
|----------------------|------------|--------------|----------|
| Arquitectura R-R | | | |
| Referencias de datos | 5.4 | 4.9 | 1.4 |
| Palabras de instr. | 14 | 18.9 | 3.9 |
| Arquitectura M-M | | | |
| Referencias a datos | 12.4 | 10.5 | 4.1 |
| Palabras de instr | 7.5 | 8.4 | 2.4 |

- a. Calcular el porcentajes de accesos a memoria que se realizan para buscar instrucciones de los tres programas para la arquitectura R-R y para la arquitectura M-M.
- b. ¿Cuál es la relación de accesos totales entre ambas arquitecturas?
- palabras de instrucciones significa número de instrucciones
 - referencias de datos: son las referencias del campo de operaciones
 - para la arquitectura R-R las instrucciones de operaciones aritmético-logicas hacen referencia a los operandos en registro, no hacen referencia a la memoria, las instrucciones load y store hacen una referencia a memoria.
 - Del número total de instrucciones ($14+18.9+3.9=36.8$ millones) hay ($5.4+4.9+1.4=11.7$ millones) que hacen referencia a memoria, luego el total de accesos es $36.8+11.7=48.5$ millones
 - % de referencias a instrucciones = $100*36.8/48.5 =75\%$
 - para la arquitectura M-M hay instrucciones que hacen 0 referencias a memoria, otras instrucciones 1 referencia, otras 2 y otras 3.
 - El total de accesos es el número de instrucciones($12.4+10.5+4.1=27$ millones) más el número de referencias a memoria ($7.5+8.4+2.4 =18.3$ millones),luego el total de accesos es $27+18.3=45.3$ millones
 - % de referencias a instrucciones = $100*27/45.3 =59\%$
 - Relación R-R/M-M = $48.5/45.3=1.07$ veces accesos a memoria , R-R respecto M-M.
- c. Calcular el porcentajes de accesos a memoria que se realizan para buscar instrucciones de los tres programas para la arquitectura R-R y para la arquitectura M-M.
- d. ¿Cuál es la relación de accesos totales entre ambas arquitecturas?

10. Para la arquitectura M68000 de Motorola de 32 bits, mostrar el contenido de todos los registros y posiciones de memoria afectadas (sin incluir el PC) por la ejecución de cada una de las instrucciones, suponiendo que partimos siempre de las condiciones iniciales siguientes:

| Instrucciones: | |
|-----------------------|-----------|
| a. CLR.L | -(A1) |
| b. CLR.W | D2 |
| c. MOVE.W | \$1204,D1 |

| Instrucciones: | |
|----------------|--------------|
| d. MOVE.W | #\$1204,D1 |
| e. MOVE.B | (A2)+,\$1200 |
| f. MOVE.L | D1,-(A2) |
| g. MOVE.L | (A1)+,D2 |

| Condiciones iniciales: | |
|------------------------|-------------|
| REGISTROS | MEMORIA |
| A1:00001202 | 0011FE:7777 |
| A2:00001204 | 001200:1111 |
| D1:01020304 | 001202:2222 |
| D2:F0F1F2F3 | 001204:3388 |
| | 001206:4444 |
| | 001208:5555 |
| | 00120A:6666 |

- Sufijos → Long=4 bytes, Word=2Bytes, Byte=1Byte
- El incremento o decremento de la dirección efectiva se escala con el tamaño del operando
- SOLUCION

CLR.L -(A1) :

Clear operando long

Predecremento del registro A1 seguido de indirección

A1<-A1-4 ;(A1-4=0x1202-0x4=0x11FE) A1:000011FE

M[A1]<-0,M[A1+1]<-0,M[A1+2]<-0,M[A1+3]<-0 M[0011FE]:0000 M[001200]:0000

CLR.W D2 :

Clear operando Word

D2(15:0)<-0 ; D2:F0F10000

MOVE.W \$1204,D1

Copiar 2bytes de Op_fuente (Dir. Directo) en Op_destino (Registro)

D1(15:0)<-M[0x1204] ;(M[0x1204]=3388); D1:01023388

MOVE.W #\$1204,D1

Copiar 2bytes de Op_fuente (Dir. Inmediato) en Op_destino (Registro)

D1(15:0)<-0x1204 ; (D1:01021204)

MOVE.B (A2)+,\$1200

Copiar 1byte Op_fuente (indirecto con postincremento), Op_destino (Directo)

0x1200<-M[A2][LSB] ;(M[A2][LSB]=M[A2+1]=M[1205]=88) ; M[1200]:1188

A2<-A2+1 ;(A2+1=0x1204+0x1=0x1205) ; A2:00001205

MOVE.L D1,-(A2)

Copiar 4bytes Op_fuente(Registro) a Op_destino(indirecto con predecremento)

A2<-A2-4 ;(A2-4=0x1204-0x4=0x1200) A2:00001200

M[A2+3]<-D1(7:0),M[A2+2]<-D1(15:8),M[A2+1]<-D1(23:16),M[A2]<-D1(31:24)
; M[001200]:01020304

MOVE.L (A1)+,D2

```

Copiar 4bytes Op_fuente(Indirecto con postincremento) a
Op_destino(Registro)
D2(7:0)<-M[A1+3] ; D2(15:8)<-M[A1+2]; D2(23:16)<-M[A1+1]; D2(31:24)<-
M[A1] ; D2:11112222
A1<-A1+4; (A1=0x1202+0x4=0x1206) A1:00001206

```

11. Mostrar el contenido de todos los registros y posiciones de memoria afectadas (sin incluir el PC) por la ejecución de cada una de las instrucciones, suponiendo que partimos siempre de las condiciones iniciales especificadas:

| | |
|-----------------------|-------------------------------|
| Instrucciones: | a. MOVE.W -(A1),A3 |
| b. CLR.B -11(A2) | c. MOVE.W (A4)+,-100(A1,D5.W) |

| Condiciones iniciales: | |
|-------------------------------|-------------|
| REGISTROS | MEMORIA |
| A1:00001504 | 001500:1234 |
| A2:00001510 | 001502:5678 |
| A3:11122233 | 001504:9ABC |
| A4:00001506 | 001506:EF11 |
| D5:FA000064 | 001508:2233 |
| | 00150A:4455 |

◦ SOLUCION: FALTA POR HACER

12. Para la arquitectura M68000-32 de Motorola, suponiendo que se dan las siguientes condiciones iniciales, mostrar el contenido de todos los registros y posiciones de memoria afectadas (incluyendo el PC) por la ejecución de cada una de las instrucciones. Suponer, además, que las instrucciones están en posiciones consecutivas de memoria, a partir de la dirección \$2000, y que se ejecutan en secuencia.

| Condiciones iniciales: | |
|-------------------------------|-------------|
| REGISTROS | MEMORIA |
| A1:00001504 | 001500:1234 |
| A2:00001510 | 001502:5678 |
| A4:00001506 | 001504:9ABC |
| D3:11122233 | 001506:EF11 |
| D5:FA000070 | 001508:2233 |
| D6:AB00FF9B | 00150A:4455 |

| Instrucciones: | CLR.B -(A4) |
|------------------------------|--------------------------|
| MOVE.L -124(A2, D5.W), -(A1) | MOVE.W \$64(A4,D6.W), D3 |

13. Comparar los computadores de 1, 2 y 3 direcciones escribiendo los programas para calcular la expresión $X = (A+B*C)/(D-E*F)$ siendo los repertorios de cada uno de ellos los siguientes:

0 Address 1 Address 2 Address 3 Address

| | | | |
|--------|---------|---------------------|---------------------|
| PUSH M | LOAD M | MOVE X,Y ;(X<-Y) | MOVE X,Y ;(X<-Y) |
| POP M | STORE M | ADD X,Y ;(X <- X+Y) | ADD X,Y ;(X <- Y+Z) |
| ADD | ADD M | SUB X,Y ;(X <- X-Y) | SUB X,Y ;(X <- Y-Z) |
| SUB | SUB M | MUL X,Y ;(X <- X*Y) | MUL X,Y ;(X <- Y*Z) |
| MUL | MUL M | DIV X,Y ;(X <- X/Y) | DIV X,Y ;(X <- X/Y) |
| DIV | DIV M | | |

◦ SOLUCION:

| | | | |
|--------|---------|------------|---------------|
| PUSH A | LOAD E | MOV R0, E | MUL R0, E, F |
| PUSH B | MUL F | MUL R0, F | SUB R0, D, R0 |
| PUSH C | STORE T | MOV R1, D | MUL R1, B, C |
| MUL | LOAD D | SUB R1, R0 | ADD R1, A, R1 |
| ADD | SUB T | MOV R0, B | DIV X, R0, R1 |
| PUSH D | STORE T | MOV R0, C | |
| PUSH E | LOAD B | ADD R0, A | |
| PUSH F | MUL C | DIV R0, R1 | |
| MUL | ADD A | MOV X, R0 | |
| SUB | DIV T | | |
| DIV | STO X | | |
| POP X | | | |

14. If the last operation performed on a computer with an 8-bit word was an addition in which the two operands were 00000010 and 00000011, what would be the value of the following flags?

- Carry
- Zero
- Overflow
- Sign
- Even Parity
- Half-Carry

◦ SOLUCION:

- 0010+0011=0101 → No hay llevada en el MSB, el resultado no es cero, no hay overflow ya que no hay llevada, positivo, número de unos par, no hay llevada en el bit de posición 3. Por lo que todos los flags desactivados excepto el de paridad par . El flag parity estará a 1.

15. The x86 Compare instruction (CMP) **subtracts** the source operand from the destination operand; it updates the status flags (C, P, A, Z, S, O) but does not alter either of the operands. The CMP instruction can be used to determine if the destination operand is greater than, equal to, or less than the source operand. Cuando se ejecuta CMP hay dos flags de overflow en el registro EFLAGS: el flag **CF** no hace la función de Carrier Flag sino la función de overflow flag si se comparan números sin signo. Si se comparan números con signo el flag de overflow es **OF**.

- Suppose the two operands are treated as unsigned integers. Show which status flags are relevant to determine the relative size of the two integer and what values of the flags correspond to greater than, equal to, or less than.
- Suppose the two operands are treated as twos complement signed integers. Show which status flags are relevant to determine the relative size of the two integer and what values of the flags correspond to greater than, equal to, or less than.
- Cómo varían los flags de signo SF y overflow OF para operandos con signo de 1 byte: si el operando

destino es 0x80 y el operando fuente es 0x7F; si el operando destino es 0x7F y el operando fuente 0x81.

- SOLUCION:
- Ver [Programación en Lenguaje Ensamblador \(x86\)](#),
- CMP \Rightarrow Dest - Source
- a. Enteros sin signo

Table 20. CMP/EFLAFS

| Caso | CF | PF | AF | ZF | SF | OF |
|-------------|----|----|----|----|----|----|
| Dest>Source | 0 | x | x | 0 | x | x |
| Dest=Source | 0 | x | x | 1 | x | x |
| Dest<Source | 1 | x | x | 0 | x | x |

- Relación entre SF y OF
- Datos de 1 Byte: Ejemplo A > B

1 Byte : valor máximo +127, valor mínimo -128 y operaciones de la computadora en módulo $2^8=256$
 Si $A=+127$ y $B= -8 \rightarrow A > B$
 instrucción CMP $\rightarrow A-B=+127-(-8)=+135$
 $+135 > +127 \rightarrow OF=1$
 La computadora computa +135 en módulo 256, es decir, el resultado $A-B=+135$ -módulo= $+135-256=-121 \rightarrow$ signo negativo $\rightarrow SF=1$
 La relación entre OF y SF es siempre $OF=SF$ para el caso $A > B$

- Datos de 1 Byte: Ejemplo A < B

1 Byte : valor máximo +127, valor mínimo -128 y operaciones de la computadora en módulo $2^8=256$
 Si $A=-128$ y $B= +8 \rightarrow A < B$
 instrucción CMP $\rightarrow A-B=-128-(+8)=-136$
 $-136 < -128 \rightarrow OF=1$
 La computadora computa -136 en módulo 256, es decir, el resultado $A-B=-136$ -módulo= $-136+256=+120 \rightarrow$ signo positivo $\rightarrow SF=0$
 La relación entre OF y SF es siempre $OF <> SF$ para el caso $A < B$

- interpretación gráfica: describir los dos ejemplos anteriores bien posicionando los vectores A y B en una recta o en un círculo y marcando el valor de los valores máximo, mínimo y módulo.
- OF es el flag de overflow.
- Con signo \rightarrow para el caso Destino > Fuente, la resta Destino - Fuente da lugar a CF según la resta sin signo. $SF=0$ ó 1 . $OF=SF$
- Con signo \rightarrow para el caso Destino < Fuente, la resta Destino - Fuente da lugar a CF según la resta sin signo. $SF=0$ ó 1 . $OF <> SF$
- CONCLUSION

CMP realiza los dos casos simultáneamente (con signo y sin signo) por lo que afecta a los dos flags de overflow CF y OF
 para el caso Destino > Fuente, la resta Destino - Fuente da lugar a SF=OF y CF=0
 para el caso Destino < Fuente, la resta Destino - Fuente da lugar a SF<>OF y CF=1

Table 21. Ejemplos

| Destino | Fuente | relación | Destin o- Fuente | SF | OF |
|---------|--------|------------------|---------------------|----|----|
| 3 | 6 | destino < fuente | -3 | 1 | 0 |
| -1 | +1 | destino < fuente | -2 | 1 | 0 |
| -6 | -3 | destino < fuente | -3 | 1 | 0 |
| 0x80 | 0x7F | destino < fuente | 0xF10 | 0 | 1 |
| 0x7F | 0x81 | destino > fuente | 0xFE | 1 | 1 |

- $0x80-0x7F = 0xF80-0x070=0xF01 \rightarrow 0x80-0x7F=0xF80+0xF81=0xF01 \rightarrow$ en el caso de suma $0x80+0x81$ los dos operandos de 1 byte son negativos y el resultado de 1 byte 0x01 tiene resultado positivo \rightarrow overflow \rightarrow SF<>OF
- $0x7F-0x81=0x7F+0x7F=0xFE \rightarrow$ la suma de dos números positivos da resultado negativo \rightarrow overflow \rightarrow SF=OF

16. Many microprocessor instruction SETS include an instruction that tests a condition and sets a destination operand if the condition is true. Examples include the *SETcc* on the x86 processor, the *Scc* on the Motorola MC68000 processor, and the *Scond* on the National NS32000 processor. [Manual Intel quick](#): interpretar los nmémonicos y operando de la instrucción SETcc

- a. There are a few differences among these instructions:

- SETcc and Scc operate only on a byte, whereas Scond operates on byte, word, and doubleword operands.
- SETcc and Scond set the operand to integer **one if true** and to **zero if false**, es decir, lógica positiva. Scc sets the byte to all binary ones if true and all zeros if false. What are the relative advantages and disadvantages of these differences?

- b. None of these instructions set any of the condition code flags, and thus an explicit test of the result of the instruction is required to determine its value. Discuss whether condition codes should be set as a result of this instruction (test). [Manual Intel quick](#): interpretar la instrucción TEST.

- c. A simple IF statement such as *IF b > a THEN* can be implemented using a numerical representation method, that is, making the *Boolean value* manifest (utilizando una variable booleana en memoria), as opposed to a *flow of control* method, which represents the value of a Boolean expression by a point reached in the program (con saltos condicionales). (Primero transcribir el lenguaje ASM del enunciado a lenguaje RTL). Leer los apuntes referentes a [instrucciones condicionales](#) Interpretar las instrucciones: [Manual Intel quick](#). A compiler might implement *IF b > a THEN* with the following **x86 code** for implement the *flow of control* method:

```
; Sintaxis de Intel: Opeación Op_destino, Op_fuente
SUB CX, CX ;set register CX to 0
MOV AX, B ;move contents of location B to register AX
CMP AX, A ;compare contents of register AX and location A
```

```

JLE TEST ;salto si Op_destino < Op_fuente or Op_destino=Op_fuente -> B<A
or B=A
INC CX ;add 1 to contents of register CX
TEST: JCXZ OUT ;jump if contents of CX equal 0

THEN: XXXXX ; B>A

OUT: XXXXX ; B<A or B=A

```

- Desarrollar el módulo fuente del programa if_ba.s : Editar el nuevo código y comentarios, compilar el módulo fuente y ejecutar el módulo binario paso a paso con el debugger hasta que el programa funcione correctamente.
 - Escribir este mismo programa utilizando la instrucción SETcc que ahorra memoria y tiempo de ejecución.
- d. Now consider the high-level language statement, (Primero transcribir el lenguaje ASM del enunciado a lenguaje RTL):
- $A := (B > C) OR (D == F)$ donde $: =$ significa asignación
 - A compiler might generate the following code:

```

MOV EAX, B ;move contents of location B to register EAX
CMP EAX, C ;compare contents of register EAX and location C
MOV BL, 0 ;0 represents false -> la instrucción MOV NO afecta a los
registros
JLE N1 ;jump if B<C or B=C
MOV BL, 1 ;1 represents true
N1 MOV EAX, D
CMP EAX, F
MOV BH, 0 ;0 represents false
JNE N2 ;jump if F<>D
MOV BH, 1 ;1 represents true
N2 OR BL, BH

```

- Desarrollar el módulo fuente del programa if_doble_condicion.s : Editar el nuevo código y comentarios, compilar el módulo fuente y ejecutar el módulo binario paso a paso con el debugger hasta que el programa funcione correctamente.
- ¿Cuál es el valor de BL para los casos falso y verdadero?
- Escribir este mismo programa utilizando la instrucción SETcc que ahorra memoria y tiempo de ejecución.
- SOLUCION:
 - a. **JLE Op_destino**
- Salta si el ultimo resultado activa el banderín ZF=1 ó los banderines SF y OF son diferentes (SF<>OF)
- CMP Op_destino_B,Op_fuente_A** → B-A → operandos con signo en complemento a dos
 - ZF=1 → B==A
 - SF <> OF

- SF=1,OF=0 → B<A
- SF=0,OF=1 → B>A

```

#### Programa: if_ba.s
#### Descripción: IF B > A THEN devuelve al S0 el valor lógico true ELSE
devuelve el valor lógico false
#### Lógica positiva . True=1 y False=0

.section .data
A: .short 0xFFFF
B: .short 0x00

.global main
.section .text

main: sub %cx, %cx #set register CX to 0
      mov B, %ax #move contents of location B to register AX
      cmp A, %ax # B-C
      jle TEST #salto si Op_destino < Op_fuente or Op_destino=Op_fuente
-> B<A or B=A
      inc %cx #add 1 to contents of register CX
TEST: jcxz ELSE #jump if contents of CX equal 0
THEN: # B>A
      mov $1,%ebx    # ebx  valor logico TRUE -> la condición de if es
verdad
      jmp salida
ELSE: # B<A or B=A
      mov $0,%ebx    # ebx valor logico FALSE
salida:
      mov $1,%eax
      int $0x80
.end

```

- Sustituir el salto **JLE Op_destino** por **SETcc**

```

#### Programa: if_ba_set.s
#### Descripción: IF B > A THEN devuelve al S0 el valor lógico true ELSE
devuelve el valor lógico false
#### Lógica positiva . True=1 y False=0

.section .data
A: .short 0xFFFF
B: .short 0x00

.global main
.section .text

```

```

main: sub %cx, %cx #set register CX to 0
      mov B, %ax #move contents of location B to register AX
      cmp A, %ax # B-C
      setge %cl # B>A ó B=A cx=cl=1=true . El operando ha de ser 1 byte
para setge.
TEST: jcxz ELSE # cx=0=false -> B<A ->ELSE
THEN: # B>A
      mov $1,%ebx    # ebx valor logico TRUE -> la condición de if es
verdad
      jmp salida
ELSE: # B<A or B=A
      mov $0,%ebx    # ebx valor logico FALSE
salida:
      mov $1,%eax
      int $0x80
.end

```

- b.
- módulo fuente sin SET

```

### Programa: if_doble_condicion.s
### doble condición -> A:=(B > C) OR (D == F)
### Descripción: asignar el valor booleano TRUE a la variable A si se cumple
la doble condición
### La variable A la implementamos con el registro BL
### Lógica positiva: A=TRUE=1 ó A=FALSE=0
### Devuelve el valor de la variable A al Sistema Operativo

.section .data

A: .int 0
B: .int -1
C: .int +1
D: .int -1
F: .int +1

.section .text
.global main

main: mov B,%eax #move contents of location B to register EAX
      cmp C,%eax # B-C
      mov $0,%bl #0 represents false -> la instrucción MOV NO afecta a los
registros
      jle N1 #jump if B<C or B=C
      mov $1,%bl #1 represents true
N1:   mov D,%eax
      cmp F,%eax # D-F
      mov $0,%bh #0 represents false
      jne N2 #jump if F<>D
      mov $1, %bh #1 represents true

```

```
N2:    or %bh, %bl # BL=0 si las dos condiciones son falsas y BL=1 si una o
ambas son verdad
      movsx %bl,%ebx # extensión del signo de BL a EBX

salida: mov $1,%eax
        int $0x80
        .end
```

- Si BL ó BH es verdadero la sentencia a evaluar ($B > C$) OR ($D == F$) es verdadera
- BL : variable lógica A con lógica positiva
- módulo fuente con SET

```
### Programa: if_doble_condicion_set.s
### doble condición -> A:=(B > C) OR (D == F)
### Descripción: asignar el valor booleano TRUE a la variable A si se cumple
la doble condición
### La variable A la implementamos con el registro BL
### Lógica positiva: A=TRUE=1 ó A=FALSE=0
### Devuelve el valor de la variable A al Sistema Operativo

.section .data

A: .int 0
B: .int -1
C: .int +1
D: .int -1
F: .int +1

.section .text
.global main

main:   mov B,%eax # move from location B to register EAX
        cmp C,%eax # B-C
        setg %bl # Setcc SETGreater ;BL = 1 si B>C
        mov D,%eax
        cmp F,%eax # D-F
        mov $0,%bh
        sete %bh # Setcc SETEqual ;BH = 1 si D=F
        or %bh,%bl # BL=0 si B>C es falso y si D=F es falso. El resto BL=1.

salida: movsx %bl,%ebx # extensión del signo de BL a EBX
        mov $1,%eax
        int $0x80
        .end
```

17. En la estructura de datos siguiente, dibujar el layout de memoria little-endian, teniendo en cuenta que el compilador alinea los datos con direcciones múltiplo de 4 rellenando los huecos con ceros, y así minimizar el número de transferencias entre la memoria y la CPU en la captura de los datos.

- Declaración:

```

#include <stdio.h>
void main (void)
{
    struct{
        int a;
        int pad; //
        double b;
        int* c;
        char d[7];
        short e;
        int f;
        char q[4];
    } s={.a=0x11121314,.pad=0,.b=0x2122232425262728,.d={'A','B','C','D','E','F'
,'G'},.e=0x5152,.f=0x61626364,.q="abc"};
    s.c=&s.e;
}

```

- SOLUCION Little Endian:

| | |
|-----|-------------|
| 00: | 14 13 12 11 |
| 04: | xx xx xx xx |
| 08: | rr rr rr rr |
| 0C: | rr rr rr rr |
| 10: | aa aa aa aa |
| 14: | 41 42 43 44 |
| 18: | 45 46 47 pp |
| 1C: | 52 51 pp pp |
| 20: | 64 63 62 61 |
| 24: | 61 62 63 00 |

- xx: indeterminado
- rr: código ieee-754 doble precisión
- aa: dirección de la variable e con la que es inicializado el puntero c.
- char q[4]={"abc"} : array de 4 elementos tipo carácter. Equivale a: char q[4]={a,b,c,NULL} donde el carácter NULL vale 00.

18. Para una arquitectura little endian el mapa de direcciones en memoria es el de la figura de abajo. Asociar la declaración de las estructuras s1 y s2 en lenguaje C y su inicialización con el mapa de direcciones indicando las direcciones en memoria de los elementos de las estructuras.

| Little-endian address mapping | | | | | | | | Byte address |
|-------------------------------|-----|-----|-----|----|-----|-----|-----|--------------|
| | | | | 11 | 12 | 13 | 14 | |
| 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 | 00 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 08 |
| 0F | 0E | 0D | 0C | 0B | 0A | 09 | 08 | 10 |
| 'D' | 'C' | 'B' | 'A' | 31 | 32 | 33 | 34 | 18 |
| 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 20 |
| | | 51 | 52 | | 'G' | 'F' | 'E' | |
| 1F | 1E | 1D | 1C | 1B | 1A | 19 | 18 | |
| | | | | 61 | 62 | 63 | 64 | |
| | | | | 23 | 22 | 21 | 20 | |

Figure 75. Little Endian

1. declaración de la variable s1: tipo estructura

```
struct {
    double i; //0x1112131415161718 ; 8 bytes
} s1;
```

- SOLUCION:

- MSB(i):0x11 en la dirección 0x03 y LSB(i):18 en la dirección de MSB-7

2. declaración de la variable s2: tipo estructura

```
struct {
    int i; //0x11121314 ; 4 bytes
    int j; //0x15161718
} s2;
```

- SOLUCION:

- i:

- MSB(i):0x11 en la dirección 0x03 y LSB(i) en la dirección 0x00

- j:

- j se reservar secuencialmente a continuación de i, por lo que LSB(j) estará en la dirección 0x04 y MSB(j) en la dirección 0x07

- 0x04 : 18 17 16 15

- Write a small program to determine the endianness of machine and report the results. Run the program on a computer available to you and turn in the output.

```
#include <stdio.h>
main()
{
    int integer; /*4 bytes*/
    char *p;
```

```

integer = 0x30313233; /* ASCII for chars '0', '1', '2', '3' */
p = (char *)&integer
    if (*p=='0' && *(p+1)=='1' && *(p+2)=='2' && *(p+3)=='3')
        printf("This is a big endian machine.\n");
    else if (*p=='3' && *(p+1)=='2' && *(p+2)=='1' && *(p+3)=='0')
        printf("This is a little endian machine.\n");
    else
        printf("Error in logic to determine machine endian-ness.\n");

```

- SOLUCION:

- p apunta al primer byte de la variable integer, (p+1) al siguiente byte y así sucesivamente
- si p apunta al carácter 0 significa que el MSB de integer se almacena en la dirección más baja → Big endian
- si p apunta al carácter 3 significa que el MSB de integer se almacena en la dirección más alta → Little endian

10.7. Programación asm

- Explicación breve del modus operandi de los códigos mnemónicos. Para información más detallada ir al Manual del Repertorio e Instrucciones de INTEL

10.7.1. Datos

1. Interpretar las instrucciones siguientes de un programa en lenguaje ensamblador x86-64 describiéndolas en lenguaje RTL:

- a. mov da1,da4
- b. mov \$0xFF00FF00FF00FF00,%rax
- c. mov \$0xFF,%rsi
- d. mov \$da1,%rsp
- e. lea da1,%rsp
- f. mov da4,%ebx
- g. movb da4,%ebx
- h. movl da4,%ax
- i. movw %ebx,da4
- j. movw %ebx,da1

- si la sección de datos presenta el código siguiente:

```

.data

da1:   .byte  0x0A
da2:   .2byte 0x0A0B
da4:   .4byte 0xA0B0C0D
saludo: .ascii  "hola"
lista:  .int   1,2,3,4,5

```

- SOLUCION

```

mov da1,da4
mov $0xFF00FF00FF00FF00,%rax
mov $0xFF,%rsi
mov $da1,%rsp
lea da1,%rsp
mov da4,%ebx
movb da4,%ebx
movl da4,%ax
movw %ebx,da4
movq %ebx,da4

```

10.7.2. Modos de Direccionamiento

1. Deducir la dirección efectiva del operando en las expresiones siguientes:

- \$0
- %rax
- loop_exit
- data_items(%rdi,4)
- (%rbx)
- (%rbx,%rdi,4)

- SOLUCIONES

- \$0 : inmediato : el operando está en la propia instrucción, es 0.
- %rax : directo registro . El operando está en el registro. Operando R[rax]
- loop_exit : directo memoria . La etiqueta es la dirección efectiva del operando en memoria. Operando M[loop_exit]
- data_items(%rdi,4) : indexado y desplazamiento inmediato. Dirección efectiva = data_item+4*RDI. Operando M[data_item+4*RDI]
- (%rbx) : Indirecto a registro . Dirección efectiva=RBX . Operando M[RBX]
- (%rbx,%rdi,4) : indexado y desplazamiento en registro base. Dirección efectiva = RBX+4*RDI .Operando M[RBX+4*RDI]

2. Describir en lenguaje RTL el código

```

lea buffer,%eax
mov da2,(%eax)
mov da2,%bx
mov %bx, (%eax)
incw da2
lea da2,%ebx
incw 2(%ebx)
mov $3,%esi
mov da2(%esi),%ebx

```

- SOLUCION:

```

lea buffer,%eax
mov da2,(%eax)
mov da2,%bx
mov %bx, (%eax)
incw da2
lea da2,%ebx
incw 2(%ebx)
inc 2(%ebx)
mov $3,%esi
mov da2(%esi,2),%ebx

```

10.7.3. Aritmética

- Suponer que los números enteros con signo x e y están almacenados en las posiciones 8 y 12 relativas al registro %ebp , y se desea almacenar en el top de la pila el producto x*y de 8 bytes, siendo el registro %esp (stack pointer) el puntero al top de la pila. a)Desarrollar el código ensamblador gas para la arquitectura i386 y b) dibujar el contenido de la pila sabiendo que la anchura de la pila es una palabra en la arquitectura i386 y suponiendo que el registro *ebp* y el puntero de pila *esp* están distanciados 4 palabras.

- SOLUCION: Módulo asm:

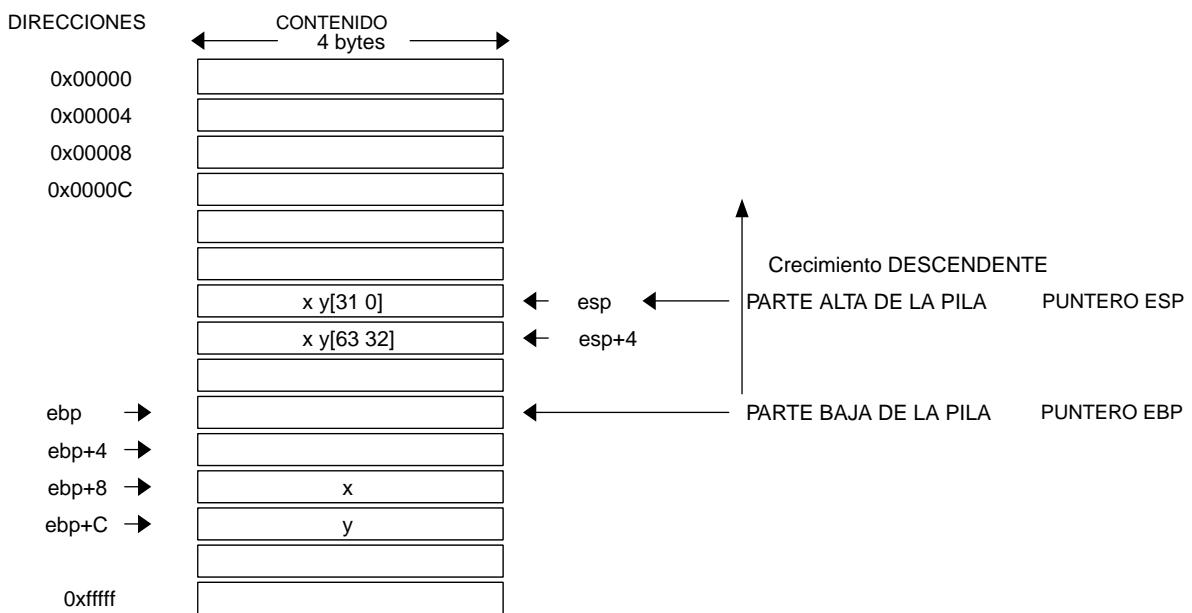
x en %ebp+8, y en %ebp+12

```

1 movl 12(%ebp), %eax
2 imull 8(%ebp)           ; EDX:EAX<-x*y[63-32:31-0] ; imul: multiplicacióñ de
enteros con signo
3 movl %eax, (%esp)       ; pila <-x*y[31:0]
4 movl %edx, 4(%esp)       ; pila <-x*y[63:32]

```

- Pila: arquitectura i386 → x86-32 → palabra=4bytes. EL crecimiento de la pila es hacia direcciones de la memoria más bajas.



- EBP: base pointer register: apunta al bottom de la pila
 - ESP: stack pointer register: apunta al top de la pila
 - La pila está ocupada desde la dirección bottom hasta la dirección top, donde la dirección top < dirección bottom.
2. Suponer que los números enteros con signo x e y están almacenados en las posiciones 8 y 12 relativas al registro %ebp , y se desea almacenar en el top de la pila el producto x/y y también x mod y, siendo el registro %esp (stack pointer) el puntero al top de la pila. Desarrollar el código ensamblador gas para la arquitectura i386.

- Solución: Módulo asm

x en ebp+8, y en ebp+12

```

1 movl 8(%ebp), %edx      ; edx<-x
2 movl %edx, %eax        ; copiar x en eax
3 sarl $31, %edx         ; edx lo lleno con el bit de signo de x , ya que forma parte
                           ; del dividendo.
                           ; edx:eax <- x
4 idivl 12(%ebp)         ; EAX<-Cociente{x/y} , EDX<-Resto{x/y}
5 movl %eax, 4(%esp)      ; pila <- Cociente{x/y}
6 movl %edx, (%esp)       ; pila <- x%y ; x%y = Resto{x/y}

```

- Mismo enunciado anterior pero utilizando la instrucción de extensión de signo cltd
 - Módulo asm

x en ebp+8, y en ebp+12

```

1 movl 8(%ebp), %edx      ; edx<-x
2 movl %edx, %eax        ; copiar x en eax
3 cltd                  ; extiende el signo del operando en eax a edx
4 idivl 12(%ebp)         ; EAX<-Cociente{x/y} , EDX<-Resto{x/y}

```

```
5 movl %eax, 4(%esp) ; pila <- Cociente{x/y}
6 movl %edx, (%esp) ; pila <- x%y ; x%y = Resto{x/y}
```

3. Del módulo fuente en lenguaje C:

- Deducir el tipo num_t del argumento del prototipo de la función store_prod
- Interpretar el módulo ASM en lenguaje RTL

```
void store_prod(num_t *dest, unsigned x, num_t y) {
    *dest = x*y;
}
```

dest en ebp+8, x en ebp+12, y en ebp+16

```
1 movl 12(%ebp), %eax
2 movl 20(%ebp), %ecx
3 imull %eax, %ecx
4 mull 16(%ebp)
5 leal (%ecx,%edx), %edx
6 movl 8(%ebp), %ecx
7 movl %eax, (%ecx)
8 movl %edx, 4(%ecx)
```

◦ SOLUCION:

- El argumento dest está implementado en la dirección ebp+8
- dest es un puntero a un objeto de tipo num_t
- línea 6: carga ecx con el valor de dest
- línea 7: carga eax en la dirección de memoria a la que apunta dest
- línea 1: carga eax con la variable x que es de tipo sin signo, luego num_t es unsigned.

10.7.4. Saltos

- Calcular las direcciones de salto en código máquina en el siguiente bloque de código ensamblador:

◦ módulo fuente:

```
1 jle .L2 if <=, goto dest2
2 .L5: dest1:
3 movl %edx, %eax
4 sarl %eax
5 subl %eax, %edx
6 leal (%edx,%edx,2), %edx
7 testl %edx, %edx
8 jg .L5 if >, goto dest1
9 .L2: dest2:
10 movl %edx, %eax
```

- Módulo objeto reubicable : las posiciones de memoria son relativas a la dirección de referencia "silly" (dirección cero del módulo reubicable)

```

1 8: 7e 0d jle 17 <silly+0x17> Target = dest2
2 a: 89 d0 mov %edx,%eax dest1:
3 c: d1 f8 sar %eax
4 e: 29 c2 sub %eax,%edx
5 10: 8d 14 52 lea (%edx,%edx,2),%edx
6 13: 85 d2 test %edx,%edx
7 15: 7f f3 jb a <silly+0xa> Target = dest1
8 17: 89 d0 mov %edx,%eax dest2:

```

- Módulo objeto ejecutable : El linker ha resuelto las posiciones de memoria relativas del módulo objeto reubicable convirtiéndolas en direcciones de memoria absolutas.

```

1 804839c: 7e 0d jle 80483ab <silly+0x17>
2 804839e: 89 d0 mov %edx,%eax
3 80483a0: d1 f8 sar %eax
4 80483a2: 29 c2 sub %eax,%edx
5 80483a4: 8d 14 52 lea (%edx,%edx,2),%edx
6 80483a7: 85 d2 test %edx,%edx
7 80483a9: 7f f3 jb 804839e <silly+0xa>
8 80483ab: 89 d0 mov %edx,%eax

```

◦ SOLUCION:

- los saltos están en las líneas 1 y 7 del código.
- el operando del salto de la línea 1 es la dirección absoluta 80483ab etiquetada como dest2
 - Cuando se ejecuta la línea 1 el PC ó RIP apunta a la línea 2, es decir, 804839e
 - El salto será la resta 80483ab - 804839e = 0D
- el operando del salto de la línea 7 es la dirección 804839e etiquetada como dest1
 - El salto será la resta 804839e - 80483ab
 - Como la resta va a dar negativo invierte los operandos y después cambio el signo del resultado
 - 80483ab - 804839e = 0D
 - El complemento a 2 de tamaño 1 byte de 0D es F2+1 = F3

10.7.5. If-Then-Else

1. Relacionar un programa en lenguaje C (Ref. Randal194) y el programa equivalente en lenguaje ensamblador mediante comentarios en el módulo fuente de bajo nivel:

- Módulo C:

(a) Original C code

```

1 int absdiff(int x, int y) {
2 if (x < y)
3 return y - x;

```

```
4 else
5 return x - y;
6 }
```

(b) Equivalent `goto` version

```
1 int gotodiff(int x, int y) {
2 int result;
3 if (x >= y)
4 goto x_ge_y;
5 result = y - x;
6 goto done;
7 x_ge_y:
8 result = x - y;
9 done:
10 return result;
11 }
```

- Módulo ASM:

(c) Generated assembly code

x at %ebp+8, y at %ebp+12

```
1 movl 8(%ebp), %edx Get x
2 movl 12(%ebp), %eax Get y
3 cmpl %eax, %edx Compare x:y
4 jge .L2 if >= goto x_ge_y
5 subl %edx, %eax Compute result = y-x
6 jmp .L3 Goto done
7 .L2: x_ge_y:
8 subl %eax, %edx Compute result = x-y
9 movl %edx, %eax Set result as return value
10 .L3: done: Begin completion code
```

10.7.6. Do-While Loops

1. Relacionar un programa en lenguaje C (Ref. Randal199) y el programa equivalente en lenguaje ensamblador mediante comentarios en el módulo fuente de bajo nivel:

- Módulo C:

```
1 int dw_loop(int x, int y, int n) {
2 do {
3 x += n;
4 y *= n;
5 n--;
6 } while ((n > 0) && (y < n));
7 return x;
8 }
```

- Módulo ASM:

x at %ebp+8, y at %ebp+12, n at %ebp+16

```

1 movl 8(%ebp), %eax
2 movl 12(%ebp), %ecx
3 movl 16(%ebp), %edx
4 .L2:
5 addl %edx, %eax
6 imull %edx, %ecx
7 subl $1, %edx
8 testl %edx, %edx
9 jle .L5
10 cmpl %edx, %ecx
11 jl .L2
12 .L5:

```

- Write a goto version of the function (in C) that mimics how the assembly code program operates.
2. Relacionar un programa en lenguaje C (Ref. Randal201) y el programa equivalente en lenguaje ensamblador mediante comentarios en el módulo fuente de bajo nivel:

- Módulo C:

```

1 int loop_while(int a, int b)
2 {
3     int result = 1;
4     while (a < b) {
5         result *= (a+b);
6         a++;
7     }
8     return result;
9 }

```

- Ensamblaje: In generating the assembly code, gcc makes an interesting transformation that, in effect, introduces a new program variable. Register %edx is initialized on line 6 and updated within the loop on line 11. Describe how it relates to the variables in the C code. Create a table of register usage for this function.

a at %ebp+8, b at %ebp+12

```

1 movl 8(%ebp), %ecx
2 movl 12(%ebp), %ebx
3 movl $1, %eax
4 cmpl %ebx, %ecx
5 jge .L11
6 leal (%ebx,%ecx), %edx
7 movl $1, %eax
8 .L12:
9 imull %edx, %eax
10 addl $1, %ecx
11 addl $1, %edx

```

```
12 cmpl %ecx, %ebx
13 jg .L12
14 .L11:
```

3. Relacionar un programa en lenguaje C (Ref. Randal204) y el programa equivalente en lenguaje ensamblador mediante comentarios en el módulo fuente de bajo nivel:

- Módulo C:

```
1 int fact_for_goto(int n)
2 {
3     int i = 2;
4     int result = 1;
5     if (!(i <= n))
6         goto done;
7     loop:
8     result *= i;
9     i++;
10    if (i <= n)
11        goto loop;
12    done:
13    return result;
14 }
```

- Módulo asm

Argument: n at %ebp+8

Registers: n in %ecx, i in %edx, result in %eax

```
1 movl 8(%ebp), %ecx Get n
2 movl $2, %edx Set i to 2 (init)
3 movl $1, %eax Set result to 1
4 cmpl $1, %ecx Compare n:1 (!test)
5 jle .L14 If <=, goto done
6 .L17: loop:
7 imull %edx, %eax Compute result *= i (body)
8 addl $1, %edx Increment i (update)
9 cmpl %edx, %ecx Compare n:i (test)
10 jge .L17 If >=, goto loop
11 .L14: done:
```

10.8. Lenguaje de Programación C

10.8.1. Punteros

1. Editar y ejecutar el siguiente programa en lenguaje C interpretando el resultado sabiendo que lista es una "variable puntero" que apunta al elemento lista[0].

```
#include <stdio.h>

void main (void)
{
    int lista[]={1,2,3,4,5};

    printf ("\n *****ARRAY LISTA*****");
    printf ("\n *****lista es una VARIABLE PUNTERO*****");
    printf ("\n **la variable lista contiene la dirección de lista[0]*****\n\n");
    printf("\n lista[0] = %d es el 1º elemento de lista \n", lista[0]);
    printf("\n lista = %p es la dirección del 1º elemento \n", lista);
    printf("\n &lista[0] = %p es la dirección del 1º elemento \n", &lista[0]);
    printf("\n *lista = %d equivale a lista[0] \n", *lista);
    printf("\n *&lista = %p equivale a lista \n", *&lista);
    printf("\n **&lista = %d equivale a lista[0] \n", **&lista);

    printf ("\n\n *****ARITMETICA DE PUNTEROS*****");
    printf("\n *(lista+1) = %d equivale a lista[1] \n", *(lista+1));
    printf("\n *(lista+4) = %d equivale a lista[4] \n", *(lista+4));

    printf ("\n\n *****CASTING*****");
    printf("\n (int *)lista = %p \n", (int *)lista);
    printf("\n *(int *)lista = %d \n", *(int *)lista);
    printf("\n *(int)lista+1 = %d \n", *(int *)lista+1);
    printf("\n *(int)lista+4 = %d \n", *(int *)lista+4);

    printf("\n *((short *)lista+1) = %d . Ahora lista lo declaro con elementos de 2
bytes \n", *((short *)lista+1));
}
```

2. Al depurar con el depurador GDB un programa escrito en lenguaje ensamblador donde se ha declarado la directiva `lista: .int 1,2,3,4,5`, deducir las siguientes sentencias si la etiqueta lista es ensamblada como la dirección 0x55bf0000 y por lo tanto no es una variable puntero, sino que es el mismo puntero, es decir, la dirección del primer elemento del array lista. Indicar la relación entre el mapa de direcciones de memoria y el mapa de posiciones de elementos del array lista.

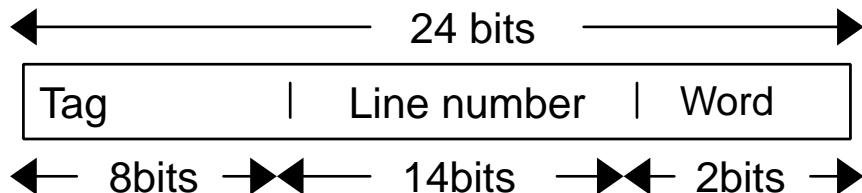
```
&lista : dirección de la "variable array" lista para el elemento de la posición 0
: 0x55bf0000
&lista+1      : 0x55bf0004 -> dirección para el elemento de la posición 1.
(void *)&lista+1   : 0x55bf0001
(int *)&lista+1   : 0x55bf0004
lista          : 1 -> equivale a lista[0]
(int)lista       : 1
(int *)&lista     : 0x55bf0000
(int [5])lista   : {1,2,3,4,5}
*(int *)&lista+1  : 2
```

10.9. Capítulo 4: Memoria Cache

- Ejemplo 4.2a Pg118. The system has a Cache memory of 64KB and Main Memory of 16MB with a byte word size and four word block size. For a cache controller with direct mapping correspondence function search the main memory block addresses correspondences to cache memory 0x0CE7 number line .

◦ Desarrollo:

- Memoria principal: 16MB, byte word, 4 byte block.
 - $16MB \rightarrow 2^{24} \rightarrow 24$ bits address bus
- Memoria cache: 64KB, 4 byte line, 16K lines.
 - $16K \rightarrow 2^{14} \rightarrow 14$ bits campo de línea
- Direct mapping correspondence function: 0x0CE7 cache line
- $i=j \bmod m$ donde i es el número de línea, j el número de bloque y m el número de líneas de la caché.
- la dirección de 24 bits se descompone en : etiqueta-línea-palabra



- Cada tag agrupa 16K bloques $\rightarrow 16K \text{ bloques} \times 4 \text{ bytes/bloque} \times \text{número de tags } N = 16MB \rightarrow 2^{14} \times 2^2 \times N = 2^{24} \text{ bytes} \rightarrow N = 2^8 \text{ Tags}$
- 0CE7 : 14 bits: 00-1100-1110-0111. Buscamos las direcciones de memoria asociadas a dicha línea.
 - Tag 0, Línea 0CE7, Palabra 0 $\rightarrow 0000-0000-00-1100-1110-0111-00 = 0000-0000-0011-0011-1001-1100 \rightarrow 00339C$
 - Tag 1, Línea 0CE7, Palabra 0 \rightarrow cambia el primer dígito a 1 $\rightarrow 01339C$
 - Tag 255, Línea 0CE7, Palabra 0 \rightarrow cambia el primer dígito a FF $\rightarrow FF339C$
 - Las direcciones de memoria son la dirección de la primera palabra de bloque en Memoria Principal.
- 4.1 A set-associative cache consists of 64 lines, or slots, divided into four-line sets. Main memory contains 4K blocks of 128 words each. Show the format of main memory addresses.
- Desarrollo:
 - Caché: 64 líneas de 128 palabras cada una agrupadas en sets de 4 líneas
 - 128 palabras $\rightarrow 7$ bits para direccionar la palabra dentro de la línea
 - 16 sets $\rightarrow 2^4 \rightarrow 4$ bits para direccionar los sets dentro de la caché
 - Main memory: 4Kblocks de 2^7 palabras
 - 12 bits para direccionar un bloque
 - 2^{19} palabras $\rightarrow 512K$ palabras $\rightarrow 19$ bits para direccionar una palabras \rightarrow ancho bus de direcciones
 - set associative $\rightarrow i = j \bmod v$ donde v es el número de sets, j el bloque e i el set

- Tag → código para diferenciar los bloques que van al mismo set. bits Tag=bits totales - bits Set - bits Word=19-4-7=8 bits.
- Tag/Set/Word → 19 address bits descompuestos en los 3 campos de 8/4/7 bits
- Sol:
 - Tag/Set/Word : 8/4/7
- 4.3 For the hexadecimal main memory addresses 111111, 666666, BBBBBB, show the following information, in hexadecimal format:
 1. Tag, Line, and Word values for a direct-mapped cache, using the format of Figure 4.10
 2. Tag and Word values for an associative cache, using the format of Figure 4.12
 3. Tag, Set, and Word values for a two-way set-associative cache, using the format of Figure 4.15
- Desarrollo:
 - a) Direct mapped Tag/Line/Word → 24 address bits descompuestos en los 3 campos de 8/14/2 bits
 - 111111 = 0001-0001-0001-0001-0001-0001 = 00010001-00010001000100-01=0001-0001-00-0100-0100-0100-01=11-0444-1 → El 0 no se escribe en hex por la izda
 - b) Full associative cache Tag/Word → 24 address bits descompuestos en los 2 campos de 22/2 bits
 - 111111 = 0001-0001-0001-0001-0001-0001 = 0001000100010001000100-01=00-0100-0100-0100-0100-01=044444-1 → El 0 no se escribe en hex por la izda
 - c) Set associative cache Tag/Set/Word → 24 address bits descompuestos en los 3 campos de 9/13/2
 - 111111 = 0001-0001-0001-0001-0001-0001 = 000100010-0010001000100-01=0-0010-0010-0-0100-0100-0100-01=022/0444/1-1 → El 0 no se escribe en hex por la izda
- Sol:

Table 22. Direcciones

| Address | 111111 | 666666 | BBBBBB |
|------------------|----------|-----------|-----------|
| a. Tag/Line/Word | 11/444/1 | 66/1999/2 | BB/2EEE/3 |
| b. Tag/Word | 44444/1 | 199999/2 | 2EEEEEE/3 |
| c. Tag/Set/Word | 22/444/1 | CC/1999/2 | 177/EEE/3 |

- 4.5 Consider a 32-bit microprocessor that has an on-chip 16-KByte four-way set-associative cache. Assume that the cache has a line size of four 32-bit words.
 1. Draw a block diagram of this cache showing its organization and how the different address fields are used to determine a cache hit/miss.
 2. Where in the cache is the word from memory location ABCDE8F8 mapped
- Desarrollo:
 - Memoria principal
 - no dice nada del bus externo, supongo el máximo de 32 bits → 2^{32} Bytes → 4GB
 - Cache on-chip: bus local: 32 bits data bus y address bus: Set associative de 4 líneas por set.
 - 4 palabras de 4 bytes cada una por línea hacen un total de 16 bytes por línea (4 bits en el campo word). El código de 4 bits direcciona el primer byte de cada palabra (0x0 la palabra 0, 0x4 la palabra 1, 0x8 la palabra 2, 0xC la palabra 3)
 - El número de sets es capacidad total/bytes por set = 16KB / (4líneas/set)(4palabras/línea)(4bytes/palabra) = 16KB/64B = 2^8 =256 sets → 8bits

- el número de bloques en cache es $\text{capacidad}/\text{bytes_por_línea} = 16\text{KB}/(4\text{palabras/línea}) * (4\text{bytes/palabra}) = 1\text{Kbloques}$
- los 1kbloques se asocian en sets de 4 líneas.
- address bus=tag bits+set bits+word bits $\rightarrow 32 = \text{tag bits} + 8 + 4 \rightarrow \text{tag_bits} = 32 - 8 - 4 = 20$ bits. El campo Tag distingue bloques dentro del mismo set.
- ¿que bloques van al mismo set? $i = j \bmod v$, donde i es el número de set al que va el bloque j , v es el número de sets. Es decir, 2^{20} bloques están asociados al mismo set por lo que han de compartir 4 líneas $\rightarrow 4$ para 20.
- Tag/Set/Word $\rightarrow 32$ address bits descompuestos en los 3 campos de 20/8/4 bits
- a) Después de la descomposición tag/set/word se selecciona el set direccionado y se comparan los tags de las 4 líneas con el tag de la dirección absoluta. Son 4 comparadores, uno por vía. Al Comparador_1 irán la primera línea de cada set en que dividimos la memoria principal. 2^{32} bytes los agrupo en sets de 16 palabras por set \rightarrow la memoria principal queda dividida en 2^8 sets
- b) Descomposición 20/8/4 de la dirección ABCD8F8 \rightarrow ABCD/8F/8 \rightarrow 8F es el set 143 y el byte 8 es la palabra número 2.
- Sol: a... Descomposición: Tag/Set/Offset . 4 comparadores: 1 por cada vía del Set. b... Set 143, cualquier línea, la doblepalabra número 2.
- 4.7 The Intel 80486 has an on-chip, unified cache. It contains 8 KBytes and has a four-way set-associative organization and a block length of four 32-bit words. The cache is organized into 128 sets. There is a single “line valid bit” and three bits, B0, B1, and B2 (the “LRU” bits), per line. On a cache miss, the 80486 reads a 16-byte line from main memory in a bus memory read burst.
 1. Draw a simplified diagram of the cache
 2. show how the different fields of the address are interpreted.
 - Desarrollo:
 - Intel 80486 (Pag 38,47,130) tiene un bus de memoria de 32 bits \rightarrow address bus de 32 bits que direccionan 1 byte.
 - Caché: 8KB, set-associative de 4 vías, cada línea 4 palabras de 4 bytes (16 bytes con 4 bits), y 128 sets (7 bits)
 - 4 palabras de 4 bytes cada una por línea hacen un total de 16 bytes por línea (4 bits en el campo word). El código de 4 bits dirige el primer byte de cada palabra (0x0 la palabra 0, 0x4 la palabra 1, 0x8 la palabra 2, 0xC la palabra 3)
 - Descomposición de los 32 bits : Tag/Set/Offset $\rightarrow 21/7/4$
 - Además de los 32 bits es necesario añadir:
 - 3 bits USO para indicar de las cuatro líneas quien es la MENOS recientemente utilizada, la de menor valor de los 8 posibles: 000-001-010-011-100-101-110-111
 - 1 bit de validación que indica con el valor 1 que hace falta su actualización en MP antes de sobreescribir la línea \rightarrow técnica de postescritura.
 - Línea: Valid/LRU/Tag/Set/Offset $\rightarrow 1/3/21/7/4$
 - Solución: a.. Esquema Set associative b.. Valid/LRU/Tag/Set/Offset $\rightarrow 1/3/21/7/4$
- 4.15 Consider the following code:

```

for (i=0; i=20; i++)
  for ( j=0; j=10; j++)
    a[i] = a[i] * j ;
  
```

1. Give one example of the spatial locality in the code.
2. Give one example of the temporal locality in the code.
 - Desarrollo
 - en el bucle interno siempre se repite la misma instrucción, siempre accedes a la misma dirección donde esta la instrucción → localidad espacial
 - en el bucle interno siempre se repite la misma instrucción, el futuro es el presente → localidad temporal
 - en el bucle interno con $j=0$ accedes al operando $a[0]$ y en la siguiente iteración se repite el mismo operando $a[0]$ → localidad espacial y temporal.
- 4.18 Consider a cache of 4 lines of 16 bytes each. Main memory is divided into blocks of 16 bytes each. That is, block 0 has bytes with addresses 0 through 15, and so on. Now consider a program that accesses memory in the following sequence of addresses:

Once: 63 through 70
Loop ten times: 15 through 32; 80 through 95

1. Suppose the cache is organized as direct mapped. Memory blocks 0, 4, and so on are assigned to line 1; blocks 1, 5, and so on to line 2; and so on. Compute the hit ratio.
2. Suppose the cache is organized as two-way set associative, with two sets of two lines each. Even-numbered blocks are assigned to set 0 and odd-numbered blocks are assigned to set 1. Compute the hit ratio for the two-way set-associative cache using the least recently used replacement scheme.
- 4.21 Consider a single-level cache with an access time of 2.5 ns, a line size of 64 bytes, and a hit ratio of $H = 0.95$. Main memory uses a block transfer capability that has a firstword (4 bytes) access time of 50 ns and an access time of 5 ns for each word thereafter.
 1. What is the access time when there is a cache miss? Assume that the cache waits until the line has been fetched from main memory and then re-executes for a hit.
 2. Suppose that increasing the line size to 128 bytes increases the H to 0.97. Does this reduce the average memory access time?
- 4.24 On the Motorola 68020 microprocessor, a cache access takes two clock cycles. Data access from main memory over the bus to the processor takes three clock cycles in the case of no wait state insertion; the data are delivered to the processor in parallel with delivery to the cache.
 1. Calculate the effective length of a memory cycle given a hit ratio of 0.9 and a clocking rate of 16.67 MHz.
 2. Repeat the calculations assuming insertion of two wait states of one cycle each per memory cycle. What conclusion can you draw from the results?
- 4.27 For a system with two levels of cache, define T_{C1} first-level cache access time; T_{C2} second-level cache access time; T_m memory access time; H_1 first-level cache hit ratio; H_2 combined first/second level cache hit ratio. Provide an equation for T_a for a read operation.

10.10. Capítulo 5: Memoria Sincrona Dinamica RAM (SDRAM)

- 5.x La arquitectura de un computador Intel tiene un bus del sistema con una frecuencia de reloj de 100MHz, el ancho del bus de datos son 64 bits y el ancho del bus de direcciones de la placa base es de 48 bits.
 1. Calcular el ancho de banda del bus en transferencias/s y en bytes/s
 2. Calcular la capacidad de memoria

3. Calcular el ciclo de memoria teniendo en cuenta que la latencia de la memoria DRAM son 10ns.
- Desarrollo
 - i. $100 \times 10^6 \text{ ciclos/seg} \times 1 \text{ Transferencia/ciclo} = 100\text{MT/s}$
 - ii. bus de direcciones $\rightarrow 2^{48} \text{ Words} = 2^8 \times 2^{40} = 256 \text{ TWords} = 2^{48} \times 2^3 \text{ Bytes} = 2 \times 2^{50} = 2 \text{ petabytes}$
 - iii. ciclo de memoria ideal (sin bus multiplexado, sin precarga, etc)= 1 Transferencia=latencia_memoria + latencia_bus_transferencia = $10\text{ns} + 1/(10^8) = 10\text{ns} + 10\text{ns} = 20 \text{ ns}$
 - 5.2 Consider a dynamic RAM that must be given a refresh cycle 64 times per ms. Each refresh operation requires 150 ns; a memory cycle requires 250 ns.What percentage of the memory's total operating time must be given to refreshes?
 - Desarrollo
 - en 1 ms 64 refrescos de 150ns $\rightarrow 9600 \text{ ns refrescando}$
 - $9600\text{ns}/1\text{ms} = 0.0096 = 1\%$
 - Sol:
 1. 1% - 5.3 Figure 5.16 shows a simplified timing diagram for a DRAM read operation over a bus. The access time is considered to last from t1 to t2. Then there is a recharge time, lasting from t2 to t3, during which the DRAM chips will have to recharge before the processor can access them again.
 1. Assume that the access time is 60 ns and the recharge time is 40 ns.What is the memory cycle time? What is the maximum data rate this DRAM can sustain, assuming a 1-bit output?
 2. Constructing a 32-bit wide memory system using these chips yields what data transfer rate?
 - Desarrollo:
 - $t_1 \rightarrow t_2$: direccionamiento
 - $t_2 \rightarrow t_3$
 - acceso al dato
 - recarga del bus de direcciones a medio camino entre el 0 y el 1
 - 60ns de latencia y 40 de precarga = 100 ns de ciclo de memoria entre 2 lecturas consecutivas
 - durante la precarga se realizaría el burst que puede ser mayor, menor o igual a la precarga
 - El ciclo de bus de 100 ns son $1/100\text{ns} = 10\text{MHz}$. Si transferimos un bit por ciclo de bus= 10Mbps
 - Si utilizamos 32 líneas en paralelo = $32 \text{ bits/transferencia} \times 10\text{MT/s} = 320\text{Mbps} = 40 \text{ MB/s}$
 - Sol:
 - i. $t_{cycle} = 100\text{ns}$. $BW = 10\text{Mbps}$
 - ii. 40MB/s
 - 5.4 Figure 5.6 indicates how to construct a module of chips that can store 1 MByte based on a group of four 256-Kbyte chips. Let's say this module of chips is packaged as a single 1-Mbyte chip, where the word size is 1 byte. Give a high-level chip diagram of how to construct an 8-Mbyte computer memory using eight 1-Mbyte chips. Be sure to show the address lines in your diagram and what the address lines are used for.
 - Desarrollo:
 - Con 4 chips de 256Kbit creo un módulo-chip de 1Mb
 - Con 8 chips de 1Mb creo un módulo de 8Mb llevando distintos chip select a cada chip de 1Mb. Para seleccionar 1 chip de 8 necesito 3 bits de direcciones, por ejemplo los 3 bits de mayor

posición. Para direccionar un bit de un chip 1Mb necesito un bus de direcciones de 20 bits. En total necesito un bus de $20+3=23$ bits.

◦ Sol:

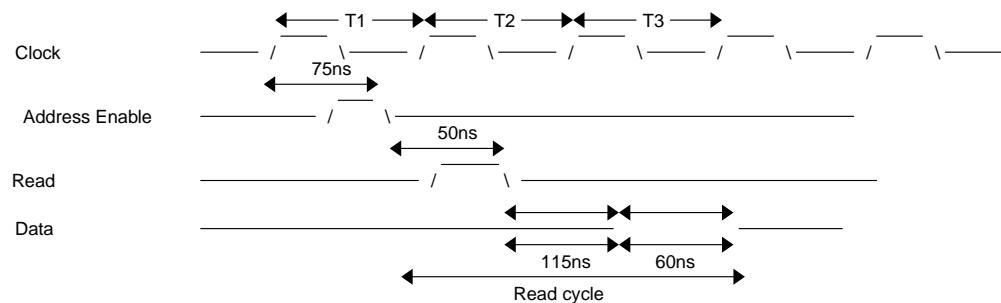
- i. 8 chips x1 de capacidad 1M donde cada entrada chip-select es la salida de un decodificador de 3 líneas de dirección

• 5.5 On a typical Intel 8086-based system, connected via system bus to DRAM memory, for a read operation, RAS is activated by the trailing edge of the Address Enable signal (Figure 3.19). However, due to propagation and other delays, RAS does not go active until 50 ns after Address Enable returns to a low. Assume the latter occurs in the middle of the second half of state T1 (somewhat earlier than in Figure 3.19). Data are read by the processor at the end of T3. For timely presentation to the processor, however, data must be provided 60 ns earlier by memory. This interval accounts for propagation delays along the data paths (from memory to processor) and processor data hold time requirements. Assume a clocking rate of 10 MHz.

- i. How fast (access time) should the DRAMs be if no wait states are to be inserted?
- ii. How many wait states do we have to insert per memory read operation if the access time of the DRAMs is 150 ns?

▪ Desarrollo:

- Ciclo de lectura
 - Load address - Address Enable (EA)- Address Command - Access Data
 - Trailing edge = fall edge = negative edge
 - AE fall = en la segunda mitad del ciclo T1. Instante 75ns
 - RAS = Read Command : Retardo de 50ns respecto de AE fall. Instante 75+50=125ns
 - La presentación del dato en el bus debe ser realizada con 60 ns de antelación a la carga del dato en la CPU la final del ciclo T3(300ns), es decir, 300ns-60ns=240ns
 - Reloj del bus del sistema = 10 MHz = 100 ns.
- A. Tiempo de acceso (desde la orden de lectura hasta volcar el dato la memoria) sin estados de espera = Tiempo de acceso mínimo impuesto por los retardos de la ruta de datos (CPU y bus): $240\text{ns} - 125\text{ns} = 115\text{ns}$



B. Si la memoria DRAM tiene un tiempo de acceso 150ns, superior a un ciclo de bus, desde la orden de lectura la cpu debe de esperar dos ciclos de reloj, uno el propio ciclo de la orden de lectura y otro ciclo extra o ciclo de ESPERA. Por lo que 200ns son suficientes para superar los 150ns del tiempo de acceso. Si el ciclo de espera comienza después de los 115ns, tenemos 215ns que superan a los 150ns.

▪ Sol:

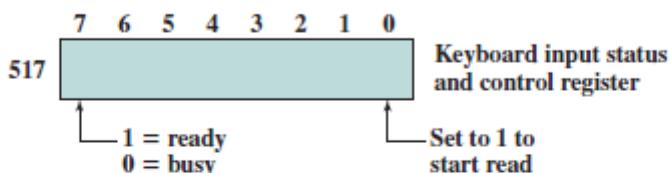
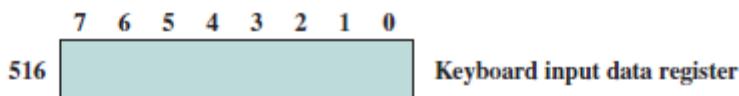
iii. 115ns

iv. 1

10.11. Capítulo 7: Sistemas Entrada/Salida

- 7.1 On a typical microprocessor, a distinct I/O address is used to refer to the I/O data registers and a distinct address for the control and status registers in an I/O controller for a given device. Such registers are referred to as ports. In the Intel 8088, two I/O instruction formats are used. In one format, the 8-bit opcode specifies an I/O operation; this is followed by an 8-bit port address. Other I/O opcodes imply that the port address is in the 16-bit DX register. How many ports can the 8088 address in each I/O addressing mode? .
 - Desarrollo:
 - memory mapped i/o : se reservan direcciones RAM para i/o
 - controlador i/o: registros datos, estado y control : puerto
 - 2 formatos
 - CodOP/Address(Dir Directo) : 8bits/8bits
 - CodOP/DX Register(Dir Indirecto) : 8bits/8bits → DX:16bits
 - Número de puertos : Directo → 2^8 e Indirecto → 2^{16} ⇒ total= $256+65536=65792$ ports
 - Sol:
 - 65792 puertos
- 7.2 A similar instruction format is used in the Zilog Z8000 microprocessor family. In this case, there is a direct port addressing capability, in which a 16-bit port address is part of the instruction, and an indirect port addressing capability, in which the instruction references one of the 16-bit general purpose registers, which contains the port address. How many ports can the Z8000 address in each I/O addressing mode?
 - Desarrollo
 - Modo directo: $2^{16} = 64K = 65536$ ports
 - Modo indirecto: $2^{16} = 64 K = 65536$ ports
 - Sol
 - $128K=131072$ puertos
- 7.5 A system is based on an 8-bit microprocessor and has two I/O devices. The I/O controllers for this system use separate control and status registers. Both devices handle data on a 1-byte-at-a-time basis. The first device has two status lines and three control lines. The second device has three status lines and four control lines.
 - a. How many 8-bit I/O control module registers do we need for status reading and control of each device?
 - b. What is the total number of needed control module registers given that the first device is an output-only device?
 - c. How many distinct addresses are needed to control the two devices?
 - modelo: El controlador i/o de los perifericos tiene implementados los puertos que son la interfaz con el periférico. Los puertos son direccionables y estan formados por un banco de registros: registro de datos, registro de control, registro de estado.
- Desarrollo:
 - buffer data de 8 bits: 2 puertos de datos (in,out) por cada periférico.
 - 1 buffer status de lectura (registro de estado) y 1 buffer control de escritura (registro de control) por cada periférico
 - las líneas de estado y control no son líneas de direccionamiento,sino que serán líneas conectadas a sus respectivos puertos. Las líneas de estado a 1 registro de estado y las líneas de control a 1 registro de control.
 - a. : 1 registro de control y 1 registro de estado por cada periférico

- b. : periférico A (1Data+1Status+1Control) y periférico B (2Data+1Status+1Control) = 7 registros
- c. : tantas direcciones como registros = 7 direcciones
- Sol:
 - a. 1 reg control y 1 reg estado
 - b. 7 registros
 - c. 7 direcciones
- 7.6 For programmed I/O, Figure 7.5 indicates that the processor is stuck in a wait loop doing status checking of an I/O device. To increase efficiency, the I/O software could be written so that the processor periodically checks the status of the device. If the device is not ready, the processor can jump to other tasks. After some timed interval, the processor comes back to check status again.



| ADDRESS | INSTRUCTION | OPERAND | COMMENT |
|---------|--------------------|---------|------------------------|
| 200 | Load AC | "1" | Load accumulator |
| | Store AC | 517 | Initiate keyboard read |
| 202 | Load AC | 517 | Get status byte |
| | Branch if Sign = 0 | 202 | Loop until ready |
| | Load AC | 516 | Load data byte |

(a) Memory-mapped I/O

| ADDRESS | INSTRUCTION | OPERAND | COMMENT |
|---------|------------------|---------|------------------------|
| 200 | Load I/O | 5 | Initiate keyboard read |
| 201 | Test I/O | 5 | Check for completion |
| | Branch Not Ready | 201 | Loop until complete |
| | In | 5 | Load data byte |

(b) Isolated I/O

Figure 7.5 Memory-Mapped and Isolated I/O

- a. Consider the above scheme for outputting data one character at a time to a printer that operates at 10 characters per second (cps). What will happen if its status is scanned every 200 ms?
- b. Next consider a keyboard with a single character buffer. On average, characters are entered at a rate of 10 cps. However, the time interval between two consecutive key depressions can be as short as 60 ms. At what frequency should the keyboard be scanned by the I/O program?
- Desarrollo:
 - a. 10cps → El periférico necesita transmitir 1 carácter cada 100ms y escribe en el puerto dicho dato. Si la CPU no salva el dato escrito por el periférico antes de cada escritura, los datos se pierden. Si la CPU consulta cada 200ms y el periférico escribe cada 100ms, cada dos datos uno se pierde. La solución sería aumentar el buffer de datos a dos caracteres o aumentar la frecuencia de consulta a períodos de 100ms.
 - b. La velocidad media es de 10cps pero la frecuencia máxima es de 60ms. La frecuencia de escaneo de la CPU tiene que ser como mínimo de 1/60ms → 16.66Hz
- 7.10 Consider a system employing interrupt-driven I/O for a particular device that transfers data at an average of 8 KB/s on a continuous basis.

- Assume that interrupt processing takes about 100 us (i.e., the time to jump to the interrupt service routine (ISR), execute it, and return to the main program). Determine what fraction of processor time is consumed by this I/O device if it interrupts for every byte.
 - Now assume that the device has two 16-byte buffers and interrupts the processor when one of the buffers is full. Naturally, interrupt processing takes longer, because the ISR must transfer 16 bytes. While executing the ISR, the processor takes about 8 us for the transfer of each byte. Determine what fraction of processor time is consumed by this I/O device in this case.
 - Now assume that the processor is equipped with a block transfer I/O instruction such as that found on the Z8000. This permits the associated ISR to transfer each byte of a block in only 2 us. Determine what fraction of processor time is consumed by this I/O device in this case.
- Desarrollo:
 - $8\text{ kB/s} \rightarrow T_{\text{int_rq}} = 1/8\text{ KB/s} = 125\text{ us} \rightarrow \text{fracción} = 100\text{ us}/125\text{ us} = 80\%$
 - $T_{\text{interrupt_service}} = 100\text{ us}(\text{ISR} + 1\text{ byte}) + 15\text{ bytes} \times 8\text{ us} = 220\text{ us} \rightarrow T_{16} = 16 \times 125\text{ us} = 2\text{ ms} \rightarrow \text{fracción} = 220\text{ us}/2000\text{ us} = 0.11 = 11\%$
 - $T_{\text{int_serv}} = 100\text{ us}(\text{ISR} + 1\text{ byte}) + 15\text{ bytes} \times 2\text{ us} = 130\text{ us} \rightarrow \text{fracción} = 130\text{ us}/2000\text{ us} = 6.5\%$
 - 7.11 In virtually all systems that include DMA modules, DMA access to main memory is given higher priority than CPU access to main memory. Why?
 - Si el buffer de datos del DMAC se llena y no es leído, se perderían los datos.
 - 7.12 A DMA module is transferring characters to memory using cycle stealing, from a device transmitting at 9600 bps. The processor is fetching instructions at the rate of 1 million instructions per second (1 MIPS). Suponer que la CPU está continuamente capturando instrucciones (no captura datos). By how much will the processor be slowed down due to the DMA activity?
 - 1MIPS \rightarrow 1 instrucción cada microsegundo. Como la CPU está continuamente capturando instrucciones tendrá ocupado el bus durante 1 microsegundo para captar cada instrucción y completar el ciclo de instrucción, por lo que el ciclo del bus del sistema es 1us.
 - 1 character = 8 bits
 - 9600 bps \rightarrow 1200 bytes/s \rightarrow 1/1200 seg/byte = 833us/byte \rightarrow cada byte se transfiere por robo de ciclo. Se roba el bus del sistema cada 833us, es decir, cada 833 ciclos del bus del sistema.
 - El bus del sistema lo tiene el DMAC durante un ciclo, es decir, 1 us.
 - Cada 833 ciclos el DMAC roba 1 \rightarrow 1/833 \rightarrow 0.12%
 - $1\text{ MIPS} \times (1 - 0.0012) = 998800$ instrucciones por segundo.
 - 7.13 Consider a system in which bus cycles takes 500 ns. Transfer of bus control in either direction, from processor to I/O device or viceversa, takes 250 ns. One of the I/O devices has a data transfer rate of 50 KB/s and employs DMA. Data are transferred one byte at a time.
 - Suppose we employ DMA in a burst mode. That is, the DMA interface gains bus mastership prior to the start of a block transfer and maintains control of the bus until the whole block is transferred. For how long would the device tie up the bus when transferring a block of 128 bytes?
 - Repeat the calculation for cycle-stealing mode.

- Desarrollo:

- 500ns dura el ciclo del bus del sistema
- $T_{\text{tx}} = 1/50\text{ KB} = 20\text{ us/B}$. El DMA según recibe el dato del periférico lo transfiere a la memoria principal, transfiriendo datos a través del bus del sistema a la misma velocidad del periférico. Sólo tiene sentido en periféricos de alta velocidad.
 - Modo ráfaga: $T = t_{\text{acceso_bus}} + t_{\text{bus_io_transferencia_bloque}} + t_{\text{liberar_bus}} = 250\text{ ns} + 128 \times 20\text{ us} + 250\text{ ns} = 2560\text{ us}$
 - Robo de ciclo $T = 128 \times (t_{\text{acceso_bus}} + t_{\text{bus_io_transferencia_byte}} + t_{\text{liberar_bus}}) = 128 \times (250\text{ ns} + 20\text{ us} + 250\text{ ns}) = 128 \times 20.5\text{ us} = 2624\text{ us}$

- 7.16 A DMA controller serves four receive-only telecommunication links (one per DMA channel) having a speed of 64 Kbps each.
 1. Would you operate the controller in burst mode or in cycle-stealing mode?
 2. What priority scheme would you employ for service of the DMA channels?
- Desarrollo:
 - a. Ahora el DMAC tendrá cuatro buffers de datos y podría acceder al bus de la misma forma que con uno. Debido a que los enlaces de telecomunicaciones ocupan el canal de forma continua (voz o datos), todo el tiempo que dura la comunicación, el modo ráfaga ocuparía el bus el 100% del tiempo. Por lo que seleccionamos el robo de ciclo.
 - b. Prioridad entre 4 clientes: misma prioridad ya que tienen la misma velocidad. Si tuviesen diferentes velocidades, tendría mayor velocidad el más rápido, el de mayor tráfico.
- 7.17 A 32-bit computer has two selector channels and one multiplexor channel. Each selector channel supports two magnetic disk and two magnetic tape units. The multiplexor channel has two line printers, two card readers, and 10 VDT terminals connected to it. Assume the following transfer rates:
 - Disk drive 800 KBytes/s
 - Magnetic tape drive 200 KBytes/s
 - Line printer 6.6 KBytes/s
 - Card reader 1.2 KBytes/s
 - VDT 1 KBytes/s
 - Estimate the maximum aggregate I/O transfer rate in this system.
 - a. Los dos canales selector tienen los mismos periféricos. Un canal selector está permanentemente asignado a sus periféricos y sólo puede dar servicio a uno de los periféricos asignados. El multiplexor en cambio da servicio a todos → Rate=800+800+2x6.6+2x1.2+10x1=1625.6KB/s
- 7.18 A computer consists of a processor and an *I/O device D* connected to *main memory M* via a shared bus with a data bus width of one word. The processor can execute a maximum of 10^6 instructions per second. An average instruction requires five machine cycles, three of which use the memory bus. A memory read or write operation uses one machine cycle. Suppose that the processor is continuously executing “background” programs that require 95% of its instruction execution rate but not any I/O instructions, es decir, el 5% son instrucciones I/O si utiliza mecanismo e/s por programa. Assume that one processor cycle equals one bus cycle. Now suppose the I/O device is to be used to transfer very large blocks of data between M and D.
 1. If programmed I/O is used and each one-word I/O transfer requires the processor to execute two instructions, estimate the maximum I/O data-transfer rate, in words per second, possible through D.
 2. Estimate the same rate if DMA is used.
- Desarrollo:
 - a. Mecanismo E/S por programa
 - i. La transferencia se realiza por programa y lo realiza la CPU. La transferencia de 1 palabra requiere la ejecución de dos instrucciones.
 - ii. 1 instrucción=3 ciclos máquina con el memory bus
 - iii. Como el ciclo de bus equivale a un ciclo máquina → 3 ciclos de bus con el memory bus → La transferencia de una palabra requiere 3 ciclos de bus→ En cada ciclo de bus se transfiere un tercio de la palabra.
 - iv. Los programas en background requieren el 95% de instrucciones a la CPU, dejando el 5% de instrucciones para I/O
 - v. Del 5% de instrucciones i/o el 2.5% son transferencias ya que hacen falta dos instrucciones i/o por transferencia.

vi. $T_{transfer}(1\text{word}) = 0.025 \times 10^6 \text{ instrucciones}_{\text{io}}/\text{seg} = 25000 \text{words}/\text{seg}$

b. DMA:

- i. Observamos el tiempo que la CPU no utiliza el bus del sistema= 5% de instrucciones (5 ciclos por instrucción) MÁS el 95% de instrucciones (2ciclos por instrucción).
- ii. El 5% de ejecución de CPU, la CPU esta libre : $10^6(\text{inst}/\text{seg}) \times 0.05 \times 5(\text{ciclos}/\text{instr}) = 250000$ ciclos/seg de procesador que utiliza el DMA= 250000 ciclos/seg de i/o que utiliza el DMA
- iii. El 95% de ejecución de CPU, el DMA comparte bus del sistema= $10^6 \times 0.95 \times 2$ ciclos libres de los 5 ciclos= 1900000 ciclos/seg de cpu= 1900000 ciclos/seg i/o
- iv. Total= $1900000 + 250000 = 2.150.000$ ciclos/seg bus i/o
- v. Si en cada ciclo se puede realizar una transferencia, esa sería la velocidad máxima. La CPU no realiza la operación de acceso a memoria, la realiza el controlador de memoria.

10.12. Capítulo 8: Operating System

- 8.3 A program computes the row sums $C_i = \sum[a_{ij}]$ para $j=1,n$ of an array A that is 100 by 100. Assume that the computer uses demand paging with a page size of 1000 words, and that the amount of main memory allotted for data is five page frames. Is there any difference in the page fault rate if A were stored in virtual memory by rows or columns? Explain.
 - Matriz A = 100x100 palabras = 10000 palabras
 - Memoria: 5 marcos de páginas : 5000 palabras
 - Proceso: 10000 palabras se dividirá en $10000/1000 = 10$ páginas
 - Almacenamiento por *filas*
 - 1^a página: a1_1,a1_2,..,a1_100,a2_1,..,a2_100,..,..,a10_1,..,a10_100 → diez filas
 - 5^a página: a41_1,..,..,a50_100
 - 10^a página: a91_1 ,...,..,a100_100
 - x^a página: desde $a10*(x-1)+1$ hasta $a10*x$ → diez filas
 - Ejecución primera fila: $C1 = \sum[a_{1j}] j=1,100$
 - Demand Paging:
 - La MP está vacía, ningún marco de página inicializado, todas las páginas en disco, sin copia en los marcos de la MPrincipal.
 - captura de a11 → FAULT (no está en MP, está en disco) → copia 1^a página → obtiene C1
 - Ejecución C2 → a21 sí está en la primera página → obtiene C2
 - Ejecución C3,..,C10 → ningún fault ya que están en la primera página
 - Ejecución C11 → FAULT → copio la 2^a página
 - Ejecución C21 → FAULT → copio la 3^a página
 - FAULTS: C1,C11,C21,..,C91
 - Cada vez que se ejecutan las 100 filas Ci se producen 10 Fallos
 - El resultado hubiese sido el mismo si en lugar de 5 páginas hubiese tenido una página si la política de reemplazo es la FIFO
 - se podrían utilizar 4 marcos de página con los mismos datos y realizar los reemplazos en el mismo marco.
 - Ejecución primera fila: $C1 = \sum[a_{1j}] j=1,100$
 - Demand Paging:
 - La MP está vacía, ningún marco de página inicializado, todas las páginas en disco, sin copia en los marcos de la MPrincipal.
 - captura de a11 → FAULT (no está en MP, está en disco) → copia 1^a página → obtiene C1
 - Ejecución C2 → a21 sí está en la primera página → obtiene C2
 - Ejecución C3,..,C10 → ningún fault ya que están en la primera página
 - Ejecución C11 → FAULT → copio la 2^a página
 - Ejecución C21 → FAULT → copio la 3^a página
 - FAULTS: C1,C11,C21,..,C91
 - Cada vez que se ejecutan las 100 filas Ci se producen 10 Fallos
 - El resultado hubiese sido el mismo si en lugar de 5 páginas hubiese tenido una página si la política de reemplazo es la FIFO
 - se podrían utilizar 4 marcos de página con los mismos datos y realizar los reemplazos en el mismo marco.
- Almacenamiento por *columnas* :
 - 1^a página: a1_1,a2_1,..,a100_1,a1_2,..,a100_2,..,..,a1_10,..,a100_10 → diez columnas

- x^a página: desde a1_10*(x-1)+1 hasta a100_10*x → diez columnas
 - Ejecución 1^a fila: C1=SUM[a1j] j=1,100
 - necesito cargar las 100 columnas de la fila 1 → necesito 10 páginas con diez columnas por página → 10 FAULTS
 - Ejecución n^a fila: se necesitan 100 columnas que están distribuidas por páginas de 10 en 10 columnas. → hacen falta 10 páginas → 10 FAULTS
- Cada vez que se ejecutan las 100 filas Ci : 10 Faults por fila → 1000 FAULTS
- 8.4 Consider a fixed partitioning scheme with equal-size partitions of 2^{16} bytes and a total main memory size of 2^{24} bytes. A process table is maintained that includes a pointer to a partition for each resident process. How many bits are required for the pointer?
 - Las tablas de descriptores están formadas por el índice y el contenido que en este caso es un puntero.
 - $2^{24}/2^{16}=2^8$ particiones de la memoria
 - Tamaño de 2^{16} → direcciones que terminan en hexadecimal en 0000 → direcciones $k \cdot 2^{16} \rightarrow 0xnn0000$
 - puntero: solo es necesario guardar los dos dígitos nn de mayor peso → 8 bits
 - luego se desplazan 16 bits a la izda para tener la dirección base
- 8.6 Suppose the page table for the process currently executing on the processor looks like the following. All numbers are decimal, everything is numbered starting from zero, and all addresses are memory byte addresses. The page size is 1024 bytes.

| Virtual page number | Valid bit | Reference bit | Modify bit | Page frame number |
|----------------------------|------------------|----------------------|-------------------|--------------------------|
| 0 | 1 | 1 | 0 | 4 |
| 1 | 1 | 1 | 1 | 7 |
| 2 | 0 | 0 | 0 | — |
| 3 | 1 | 0 | 0 | 2 |
| 4 | 0 | 0 | 0 | — |
| 5 | 1 | 0 | 1 | 0 |

Figure 76. VM

- 1. Describe exactly how, in general, a virtual address generated by the CPU is translated into a physical main memory address.
 - La dirección virtual esta formada por los campos (VPN,VPO) → (base,offset). Mediante la tabla de páginas virtuales traducimos VPN en PPN. La dirección física es el par (PPN,PPO) donde el offset PPO=VPO
 - Para qué este cacheada la página virtual en la tabla de páginas virtuales el bit de validación tiene que valer 1.
 2. What physical address, if any, would each of the following virtual addresses correspond to? (Do not try to handle any page faults, if any.)
 - a. 1052
 - VPN=Mod{1052/1024}=1 → Valid Bit=1 → PPN=7
 - VPO=Rest{1052/1024}=28
 - Dirección física = $7 \cdot 1024 + 28 = 7196$
 - b. 2221
 - VPN=Mod{2221/1024}=2 → Valid Bit=0

- No hay copia de esa página por lo que no se puede realizar la traducción
- c. 5499
- $\text{VPN} = \text{Mod}\{5499/1024\} = 5 \rightarrow \text{Valid Bit} = 1 \rightarrow \text{PPN} = 0$
 - $\text{VPO} = \text{Rest}\{5499/1024\} = 379$
 - Dirección física = $0 + 379 = 379$
- 8.8 A process references five pages, A, B, C, D, and E, in the following order: A; B; C; D; A; B; E; A; B; C; D; E . Assume that the replacement algorithm is first-in-first-out and find the number of page transfers during this sequence of references starting with an empty main memory with three page frames. Repeat for four page frames.
1. MP → 3 marcos de página ; política FIFO
 - v/v/v; A→ A/v/v ; B→ A/B/v ; C→ A/B/C; D→ D/B/C; A→ D/A/C; B→ D/A/B; E→ E/A/B ; A→ E/A/B; B→ E/A/B; C→ E/C/B; D→ E/C/D; E→ E/C/D
 - 10 Fallos
 2. MP → 4 marcos de página ; política FIFO
 - v/v/v/v; A→ A/v/v/v ; B→ A/B/v/v ; C→ A/B/C/v; D→ A/B/C/D; A→ A/B/C/D; B→ A/B/C/D; E→ E/B/C/D; A→ E/A/C/D; B→ E/A/B/D; C→ E/A/B/C; D→ D/A/B/C; E→ D/E/B/C
- 8.9 The following sequence of virtual page numbers is encountered in the course of execution on a computer with virtual memory: 3 4 2 6 4 7 1 3 2 6 3 5 1 2 3 Assume that a least recently used page replacement policy is adopted. Plot a graph of page hit ratio (fraction of page references in which the page is in main memory) as a function of main-memory page capacity n for $1 \leq n \leq 8$. Assume that main memory is initially empty.

| Nº marcos | Fracción Aciertos | 3 4 2 6 4 7 1 3 2 6 3 5 1 2 3 |
|-----------|-------------------|-------------------------------|
| 1 | 0 | |
| 2 | 0 | |
| 3 | 2/15 | 4 |
| 4 | 3/15 | 4 4 |
| 5 | 4/15 | 3 3 |
| 6 | 7/15 | 4 3 2 3 |
| 7 | 8/15 | 1 2 3 |
| 8 | 8/15 | 2 3 |

- 8.11 Suppose the program statement

```
for (i = 1; i <= n; i++)
    a[i] = b[i] + c[i];
```

- is executed in a memory with page size of 1000 words. Let $n = 1000$. Using a machine that has a full range of register-to-register instructions and employs index registers, write a hypothetical program to implement the foregoing statement. Then show the sequence of page references during execution.
 - En la memoria virtual estará tanto el código como los datos
 - Marcos de 1000 palabras.
 - Programa arquitectura load/store (espacio de direcciones virtual)

SECCION CODIGO
Ri <- 1

```

Ra <- n
loop_start: R1 <- b[Ri]
            R2 <- c[Ri]
            R3 <- R1+R2
            a[Ri] <- R3
            Flags <- Ri<Ra
            Flags:PC <- loop_start
            CPU <- halt
SECCION DATOS INICIALIZADOS
uno:      1
n:        1000
a:        array a[1000]
b:        array b[1000]
c:        array c[1000]

```

- Asignación del espacio virtual
 - Código en la página PV1
 - array A ocupa una página → PV2
 - array B ocupa una página → PV3
 - array C ocupa una página → PV4
 - uno y n en una página de tipo datos → PV5
- Ejecución
 - $1515(131411211)^{100011}$
- 8.13 Consider a computer system with both segmentation and paging. When a segment is in memory, some words are wasted on the last page. In addition, for a segment size s and a page size p, there are s/p page table entries. The smaller the page size, the less waste in the last page of the segment, but the larger the page table. What page size minimizes the total overhead?
 - Desarrollo:
 1. Número de páginas por segmento: tamaño del segmento/ tamaño de página = s/p
 2. Cada segmento tiene su propia tabla de páginas
 3. Si reducimos el tamaño de página se reduce la fragmentación interna pero se incrementa el número de entradas de la tabla de páginas.
 4. El Total de palabras desperdiciadas (w) es el desperdicio debido a las últimas páginas de cada segmento más el tamaño de la tabla de páginas. El valor medio de la fragmentación interna de todos los segmentos es $p/2$ y el tamaño de la tabla de páginas es proporcional al número de entradas de la tabla s/p → $w=p/2+s/p \rightarrow dw/dp=1/2-s/p^2=0 \rightarrow p^2=2s$
- 8.14 A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache, and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main-memory hit ratio is 0.6. What is the average time in ns required to access a referenced word on this system?
 - $T = \text{hit_cache} * t_{\text{acc_ca}} + (1 - \text{hit_cache}) \cdot \text{hit_main} \cdot (t_{\text{main_cache}} + t_{\text{acc_ca}}) + (1 - \text{hit_cache}) \cdot (1 - \text{hit_main}) \cdot (t_{\text{acc_disk_main}} + t_{\text{main_cache}} + t_{\text{acc_ca}}) = 0.9 * 20 + 0.1 * 0.6 * (60 + 20) + 0.1 * 0.4 * (12000000 + 60 + 20) = 480\mu\text{s}$
- 8.15 Assume a task is divided into four equal-sized segments and that the system builds an eight-entry page descriptor table for each segment. Thus, the system has a combination of segmentation and paging. Assume also that the page size is 2 Kbytes.

1. What is the maximum size of each segment?
 2. What is the maximum logical address space for the task?
 3. Assume that an element in physical location 00021ABC is accessed by this task. What is the format of the logical address that the task generates for it? What is the maximum physical address space for the system?
 - Desarrollo:
 1. Tabla de PAGINAS: 8 entradas : 8 paginas virtuales de 2KB → Segmento:8*2KB=16KB.
 - a. No dice nada pero ... La tabla de SEGMENTOS tendrá una entrada por segmento. Cada entrada de segmento apuntará a una tabla de página diferente. Una tabla de página por segmento.
 2. Proceso: 4 segmentos → $4 \times 16\text{KB} = 64\text{KB}$
 3. Dirección lógica → Formato (Segmento,Página,VPO) → (4seg,8pag,2KB) → (2bits,3bits,11bits) → Dirección lógica de 16 bits
 - a. Dirección física → 00021ABC → 8 dígitos hex → 32 bits → 4GB
 - b. Marcos de página → 4GB/2KB → 2^*2^{20} marcos
 - c. 00021ABC → 0000-0000-0000-0010-0001-1010-1011-1100 → marco/offset → 21/11 → 000000000000001000011 / 01010111100 → marco 67/ offset 700
 4. Traducción: El offset virtual y físico idénticos (11bits) → El segmento lógico (2bits) apunta a una tabla de páginas. La página virtual (3bits) es el offset de la tabla de páginas. Cada entrada de la tabla de páginas es un puntero a un marco de la memoria principal (una dirección base de 21 bits). Se añadir la pregunta de inventarse la tabla de descripción de segmentos y las cuatro tablas de páginas de cada segmento. En este ejercicio la dirección lógica tendrá el offset 01010111100 y los 5 bits del par seg/página no se pueden saber ya que haría falta saber en qué tabla y posición está el puntero al marco 67.
- 8.16 Assume a microprocessor capable of accessing up to 2^{32} bytes of physical main memory. It implements one segmented logical address space of maximum size 2^{31} bytes. Each instruction contains the whole two-part address. External memory management units (MMUs) are used, whose management scheme assigns contiguous blocks of physical memory of fixed size 2^{22} bytes to segments. The starting physical address of a segment is always divisible by 1024. Show the detailed interconnection of the external mapping mechanism that converts logical addresses to physical addresses using the appropriate number of MMUs, and show the detailed internal structure of an MMU (assuming that each MMU contains a 128-entry directly mapped segment descriptor cache) and how each MMU is selected.
 - un espacio virtual segmentado de 2^{31} bytes: no es el espacio virtual de cada segmento sino el de todos los segmentos.
 - dirección lógica con dos partes → (segmento,offset)
 - MP: Espacio de 2^{32} bytes con Bloques de 2^{22} bytes contiguos para cada segmento
 - offset de 22 bits
 - $2^{31}/2^{22} = 2^9$ segmentos en espacio virtual → 9 bits para el segmento en la dirección virtual y una tabla de segmentos con 512 entradas
 - Dirección lógica de 31 bits (9,22) → (seg,offset)
 - MP: segmentos alineados en múltiplos de 1K
 - segmento físico: los 10 bits de menor peso son cero y los 22 de mayor peso están en la *tabla de segmentos*.
 - Dirección física: segmento+offset
 - MMU: *tabla de segmentos* de 128 entradas (2^7)
 1. no tenemos una tabla con 512 entradas sino cuatro tablas de 128 cada una.

2. cantidad de MMUs: Si tenemos 2^9 segmentos en el espacio virtual y la MMU tiene una tabla de 2^7 necesitaremos 4 MMUs.
 3. Traducción: espacio lógico (9,22)(seg,offset) en una dirección segmento+offset de 32 bits.
 - a. De los 9 bits de segmento virtual, dos bits seleccionaran la MMU y otros siete bits la entrada de la tabla de segmentos. (2,7,22)
 - b. los 9 bits de segmento lógico son el índice de la tabla de segmentos que contiene los 22 bits altos de un segmento físico.
 - c. El offset físico también tiene un tamaño de 22 bits
 - d. dirección física: dirección base múltiplo de 1K más offset de 22bits.
- 8.17 Consider a paged logical address space (composed of 32 pages of 2 Kbytes each) mapped into a 1-Mbyte physical memory space.
 1. What is the format of the processor's logical address?
 2. What is the length and width of the page table (disregarding the "access rights" bits)?
 3. What is the effect on the page table if the physical memory space is reduced by half?
 - Desarrollo:
 - MP : 1MB con páginas de 2KB $\rightarrow 2^{20}/2^{11} = 2^9$ marcos de página
 1. VPN/OFFSET \rightarrow VPN:32 páginas supone 2^5 , 5 bits ; OFFSET:2KB supone 2^{11} , 11bits
 2. Tabla de páginas: longitud igual al número de páginas virtuales= 32 y anchura igual al puntero a uno de los 2^9 marcos, es decir, 9 bits.
 3. Si se reduce la MP a la mitad, se reduce el número de marcos a la mitad también $\rightarrow 2^8$ marcos de página \rightarrow anchura de 8 bits.
- Randal Capítulo 9: Figure 9.19 shows the formats of the virtual and physical addresses. Since each page is 64 bytes, the low-order 6 bits of the virtual and physical addresses serve as the VPO and PPO respectively. The high-order 8 bits of the virtual address serve as the VPN. The high-order 6 bits of the physical address serve as the PPN. Figure 9.20 shows a snapshot of our little memory system, including the TLB (Figure 9.20(a)), a portion of the page table (Figure 9.20(b)), and the L1 cache (Figure 9.20(c)). Above the figures of the TLB and cache, we have also shown how the bits of the virtual and physical addresses are partitioned by the hardware as it accesses these devices.

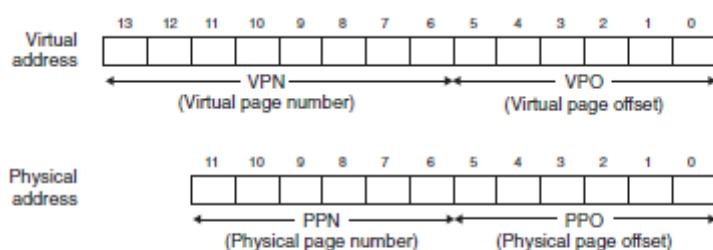
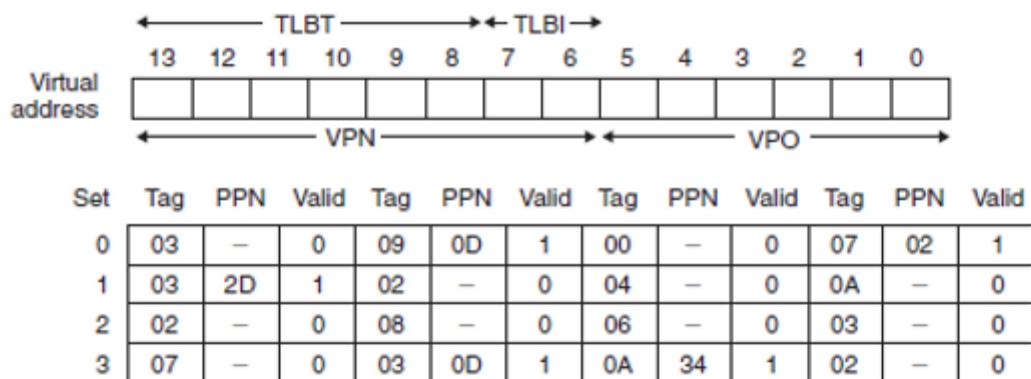


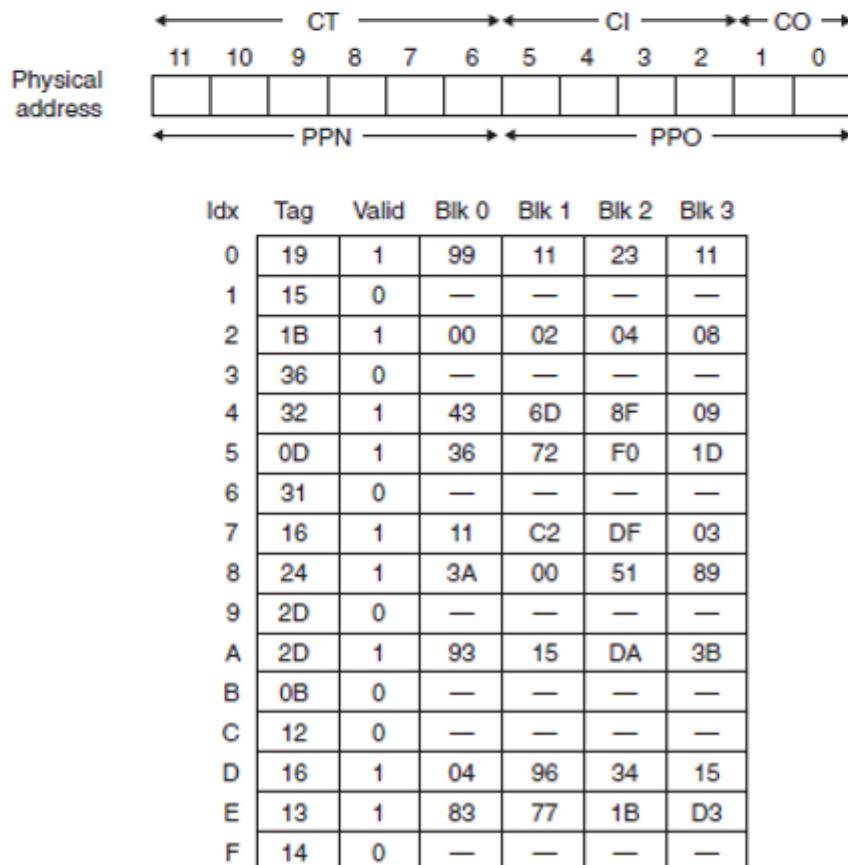
Figure 9.19 Addressing for small memory system. Assume 14-bit virtual addresses ($n = 14$), 12-bit physical addresses ($m = 12$), and 64-byte pages ($P = 64$).



(a) TLB: Four sets, 16 entries, four-way set associative

| VPN | PPN | Valid | VPN | PPN | Valid |
|-----|-----|-------|-----|-----|-------|
| 00 | 28 | 1 | 08 | 13 | 1 |
| 01 | - | 0 | 09 | 17 | 1 |
| 02 | 33 | 1 | 0A | 09 | 1 |
| 03 | 02 | 1 | 0B | - | 0 |
| 04 | - | 0 | 0C | - | 0 |
| 05 | 16 | 1 | 0D | 2D | 1 |
| 06 | - | 0 | 0E | 11 | 1 |
| 07 | - | 0 | 0F | 0D | 1 |

(b) Page table: Only the first 16 PTEs are shown



(c) Cache: Sixteen sets, 4-byte blocks, direct mapped

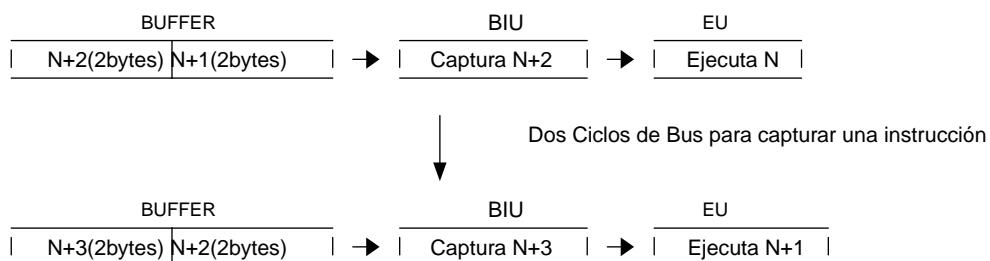
Figure 9.20 TLB, page table, and cache for small memory system. All values in the TLB, page table, and cache are in hexadecimal notation.

- Given this initial setup, let's see what happens when the CPU executes a load instruction that reads the byte at address 0x03d4
- Solución
 - TLBI:0x03
 - TLBT:0x3
 - VPN:0x0f
 - VPO:0x14
 - PPN=0x0D
 - physical address=0x354
 - CO=0x0
 - CI=0x5
 - CT=0x0D
 - Data=0x36

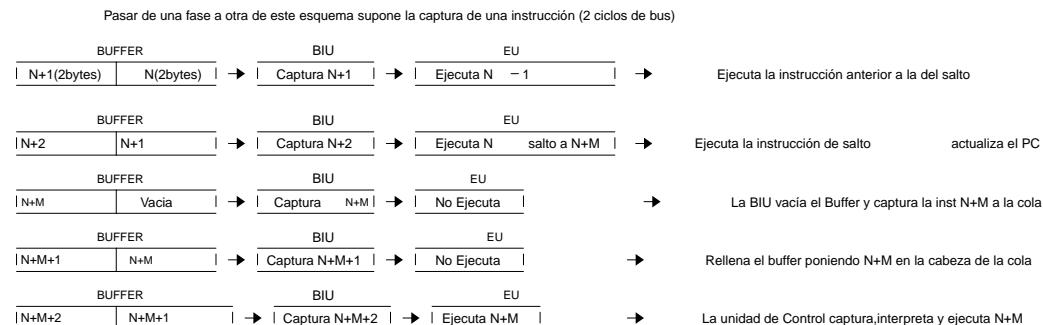
10.13. Capítulo 12: Processor Structure and Function (Capítulo 14 en 9^aEd)

- 12.1 a. If the last operation performed on a computer with an 8-bit word was an addition in which the two operands were 00000010 and 00000011, what would be the value of the following flags?
 - Carry
 - Zero
 - Overflow → Número con signo
 - Sign
 - Even Parity → Paridad PAR
 - Half-Carry
 - Desarrollo
 - 0010+0011=0101 → No hay llevada en el MSB, el resultado no es cero, no hay overflow, positivo, número de unos PAR, no hay llevada en el bit de posición 3. Por lo que todos los flags desactivados excepto el de paridad par . El flag parity estará a 1.
- 12.3 A microprocessor is clocked at a rate of 5 GHz.
 1. How long is a clock cycle?
 2. What is the duration of a particular type of machine instruction consisting of three clock cycles?
 - Desarrollo
 - a. $T = 1/f = 1/(5 \cdot 10^9) = 0.2\text{ns}$
 - b. $3T = 3 \cdot 0.2 = 0.6\text{ns}$
- 12.4 A microprocessor provides an instruction capable of moving a string of bytes from one area of memory to another. The fetching and initial decoding of the instruction takes 10 clock cycles. Thereafter, it takes 15 clock cycles to transfer each byte. The microprocessor is clocked at a rate of 10 GHz.
 1. Determine the length of the instruction cycle for the case of a string of 64 bytes.
 2. What is the worst-case delay for acknowledging an interrupt if the instruction is noninterruptible?
 3. Repeat part (b) assuming the instruction can be interrupted at the beginning of each byte transfer
 - Desarrollo

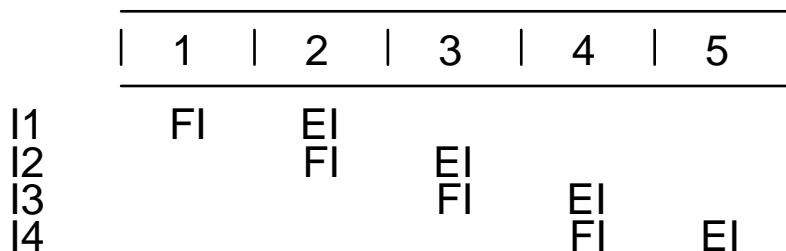
- a) $IC = \text{Instruction Cycle} = FI + DI + CO + FO + EI + WO$; $FI + DI = 10T$; $CO + FO = 0$; $EI = 15T/\text{byte}$; $WO = 0$; $T = 1/(10^9) = 0.1 \text{ ns}$; $IC = (10 + 15 * 64) * T = 970 * 0.1 = 97 \text{ ns}$
 - b) Justo nada más empezar la instrucción quedaría todo el ciclo para poder atender a la interrupción: 97 ns.
 - c) Si se interrumpe antes de la primera transfer tardaría $10T$ como mucho, y si se interrumpe durante las transferencias sería $15T$. Por lo que es caso peor sería $15T = 15 * 0.1 = 1.5 \text{ ns}$
- 12.5 The Intel 8088 consists of a bus interface unit (BIU) and an execution unit (EU), which form a 2-stage pipeline. The BIU fetches instructions into a 4-byte instruction queue. The BIU also participates in address calculations, fetches operands, and writes results in memory as requested by the EU. If no such requests are outstanding and the bus is free, the BIU fills any vacancies in the instruction queue. When the EU completes execution of an instruction, it passes any results to the BIU (destined for memory or I/O) and proceeds to the next instruction. [wikipedia](#): the 8088 had an **8-bit** external data bus
1. Suppose the tasks performed by the BIU and EU take about equal time. By what factor does pipelining improve the performance of the 8088? Ignore the effect of branch instructions.
 2. Repeat the calculation assuming that the EU takes twice as long as the BIU.
 - Desarrollo
 - 0. El micro 8088 tiene un bus de datos de 8bits. La unidad de ejecución comprende la ALU y los Registros. La BIU junto a la EU forman conjuntamente una CPU segmentada con dos unidades. La *prefetch instruction queue* es el buffer que almacena la siguiente instrucción a ejecutar.
 - a. Una etapa tarda x y la siguiente también x . La primera instrucción tarda en ejecutarse $2x$ y cada intervalo x sale una nueva por lo que la mejoría a partir de la segunda instrucción es de $x/2x \Rightarrow$ En un ciclo de instrucción (duración $2x$) salen instrucciones cada intervalo x , es decir, el doble.
 - b. $x+2x=3x$. A partir de la segunda instrucción tardan $2x$. En un ciclo de instrucción $3x$ salen instrucciones cada $2x \Rightarrow (3x \text{ time ciclo}) / (2x \text{ time/instrucción}) = 1.5$ veces más instrucciones por ciclo.
- 12.6 Assume an 8088 is executing a program in which the probability of a program jump is 0.1. For simplicity, assume that all instructions are 2 bytes long. If the prefetch instruction queue is empty, the EU waits for the next instruction byte to be fetched and shifted to top of the queue. When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions. Whenever this happens, the BIU automatically resets the queue and then begins to fetch instructions from this new location to refill the queue.
1. What fraction of instruction fetch bus cycles is wasted?
 2. Repeat if the instruction queue is 8 bytes long.
 - Buffer (de 4 bytes para dos instrucciones) \rightarrow BIU \rightarrow EU : Para leer una instrucción son necesarios **DOS ciclos de bus**, un bus de datos (1 byte) por ciclo de bus.
 - Desarrollo
 - 0. Si no hay salto durante la ejecución de la instrucción N se captura la instrucción N+1.



- 1. Si la ejecución de la instrucción N supone un salto de M instrucciones no se ejecutará la instrucción N+1 que espera en el buffer, sino que se debe ejecutar la instrucción N+M. Durante la ejecución de N NO se captura nada sino que se actualiza el Contador de Programa a N+M y en el siguiente ciclo de instrucción se capturará N+M. Por lo que si hay salto, el ciclo de captura estará infroutilizado y será necesario vaciar el buffer de instrucciones.
- Interpretación



- a. Cuando la BIU capta de la memoria principal N+1 y lo pone en cola detrás de N, se están desaprovechando los dos ciclos de bus que se necesitan para la captura de la instrucción M+1 que no se va a ejecutar. Lo mismo ocurre con la captura de N+2 de la memoria principal. Por lo tanto se malgastan los ciclos del bus del sistema de N+1 y N+2, es decir, 4 ciclos de bus.
 - El buffer de instrucción es de 4 bytes según el ejercicio anterior, por lo que es necesario "empujar" los 4 bytes de N+1 y N+2 para dejar pasar a la nueva instrucción N+M desde que es capturada de la memoria principal.
 - La captura de una instrucción no_jump supone 2 ciclos de bus bien utilizados, la instrucción estará en la cabeza del buffer cuando la vaya a ejecutar la CPU. La de una instrucción jump supone 2 ciclos bien utilizados en capturarla desde la memoria principal pero 4 ciclos mal utilizados en vaciar el buffer y desplazar la instrucción N+M desde la cola hasta la cabecera del buffer, mientras la cpu espera. De cada 100 instrucciones tendremos 100 instruccionesx2ciclos/instr bien utilizados y 10 instruccionesx4ciclos/instrucción mal utilizados→ en total 240 ciclos → fracción de infroutilización= $40/240 = 0.166 = 17\%$ del tiempo el bus no está siendo utilizado en operaciones fetch, la BIU está ocupado en vaciar el buffer.
 - b. Con una cola de 8 bytes → Total=100x2+10*8=280→ ineficiencia=80/280=0.285=28.5%
 - 12.7 Consider the timing diagram of Figures 12.10. Assume that there is only a two-stage pipeline (fetch, execute). Redraw the diagram to show how many time units are now needed for four instructions.
- Desarrollo



- Son necesarias 5 unidades de Tiempo
- 12.9 A pipelined processor has a clock rate of 2.5 GHz and executes a program with 1.5 million instructions. The pipeline has five stages, and instructions are issued at a rate of one per clock cycle. Ignore penalties due to branch instructions and out-of-sequence executions.
 1. What is the speedup of this processor for this program compared to a nonpipelined processor, making the same assumptions used in Section 12.4?
 2. What is throughput (in MIPS) of the pipelined processor?
 - Desarrollo
 - a. Duración Programa con N instrucciones, segmentación de k etapas de duración t cada una= 1^a instrucción más el resto = $k*t + (N-k)t = t(N+k-1)$ para $N > k = t^*(N-1)$. La relación sin_seg/con_seg = $N*k*t / t(N+k-1) = Nk/(N+k-1)$. Si N tiende a infinito $\rightarrow Nk/N=k=5$
 - b. Throughput = instrucciones del programa/duración del programa = $N/\{t^*(N+k-1)\}=1/t$
- 12.11 Consider an instruction sequence of length n that is streaming through the instruction pipeline. Let p be the probability of encountering a conditional or unconditional branch instruction, and let q be the probability that execution of a branch instruction I causes a jump to a nonconsecutive address. Assume that each such jump requires the pipeline to be cleared, destroying all ongoing instruction processing, when I emerges from the last stage. Revise Equations (12.1) and (12.2) to take these probabilities into account.
 - Desarrollo
 - Instrucciones cuya ejecución es un salto no consecutivo : pqn
 - Instrucciones cuya ejecución supone un no salto : (1-pq)n
 - $T_{programa} = T_{inst_salto} + T_{inst_nosalto} = \{pq*nkt\} + \{(1-pq)*(k+n-1)t\}$
- 12.13 Consider the state diagrams of Figure 12.28.
 1. Describe the behavior of each.
 2. Compare these with the branch prediction state diagram in Section 12.4. Discuss the relative merits of each of the three approaches to branch prediction.

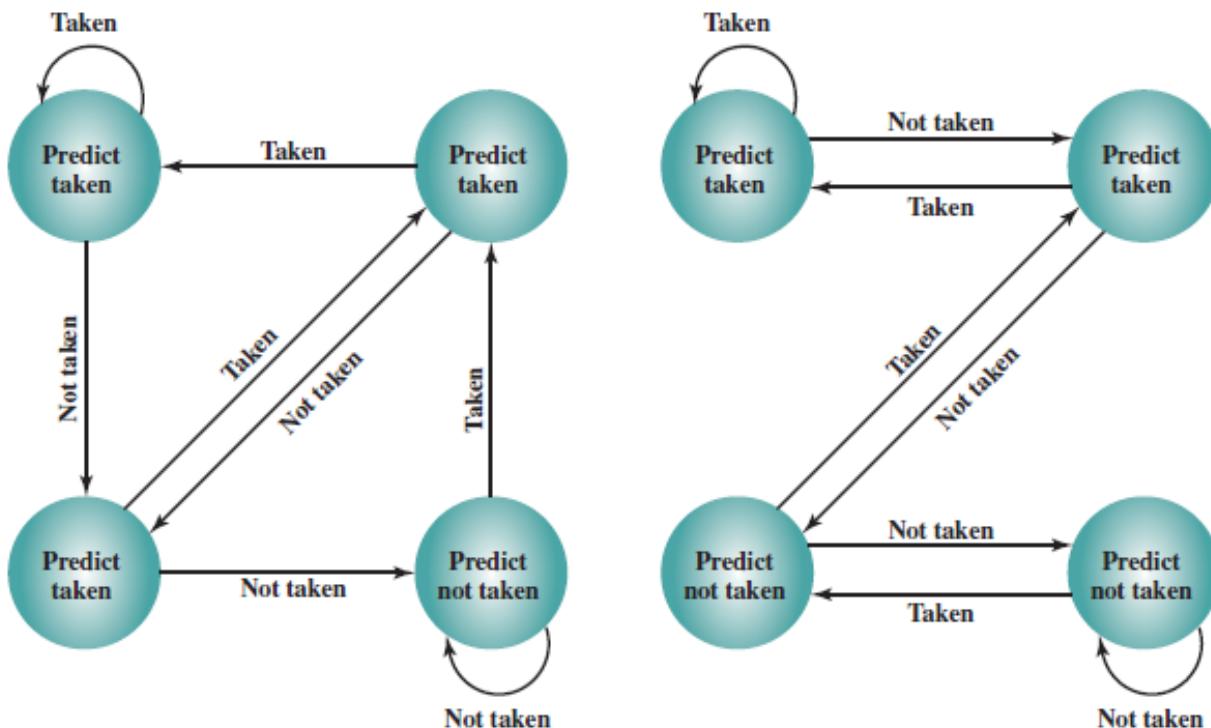


Figure 14.28 Two Branch Prediction State Diagrams

- Predict taken: predicción de SI salto.
- Diagrama A:
 - Cambio de predicción afirmativa a negativa:
 - Partiendo de predicción de salto SI → Dos "NO" consecutivos para cambiar la predicción a NO
 - Cambio de predicción negativa a afirmativa:
 - Partiendo de predicción de salto NO → Un "SI" para cambiar la predicción a SÍ
- Diagrama B:
 - Cambio de predicción afirmativa a negativa:
 - Partiendo de predicción de salto SI → Dos "NO" consecutivos para cambiar la predicción a NO
 - Cambio de predicción negativa a afirmativa:
 - Partiendo de predicción de salto NO → Un "SI" para cambiar la predicción a SÍ si previamente ha habido dos "NO SALTO" consecutivos
 - Partiendo de predicción de salto NO → Dos "SI" para cambiar la predicción a SÍ si ha habido más de dos "NO SALTO" consecutivos
- 12.14 The Motorola 680x0 machines include the instruction *Decrement and Branch According to Condition*, which has the following form:

DBcc Dn, displacement

where cc is one of the testable conditions, Dn is a general-purpose register, and displacement specifies the target address relative to the current address.

The instruction can be defined as follows:

```
if (cc = False)
  then begin
    Dn := (Dn) - 1;
    if Dn != -1 then PC := (PC) + displacement end
  else PC := (PC) + 2;
```

- When the instruction is executed, the condition is first tested to determine whether the termination condition for the loop is satisfied. If so, no operation is performed and execution continues at the next instruction in sequence. If the condition is false, the specified data register is decremented and checked to see if it is less than zero. If it is less than zero, the loop is terminated and execution continues at the next instruction in sequence. Otherwise, the program branches to the specified location. Now consider the following assembly-language program fragment:

```
AGAIN CMPM.L (A0)+, (A1)+  
DBNE D1, AGAIN  
NOP
```

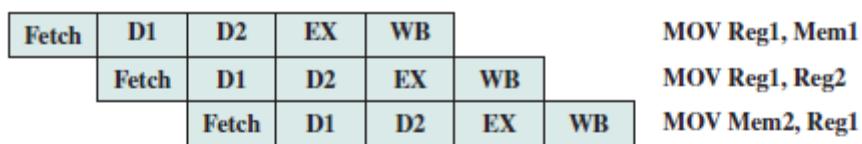
- Two strings addressed by A0 and A1 are compared for equality; the string pointers are incremented with each reference. D1 initially contains the number of longwords (4 bytes) to be compared.
1. The initial contents of the registers are A0 = \$00004000, A1 = \$00005000 and D1 = \$000000FF (the \$ indicates hexadecimal notation). Memory between \$4000 and \$6000 is loaded with words AAAA. If the foregoing program is run, specify the number of times the DBNE loop is executed and the contents of the three registers when the NOP instruction is reached.

2. Repeat (a), but now assume that memory between \$4000 and \$4FEE is loaded with \$0000 and between \$5000 and \$6000 is loaded with \$AAA.

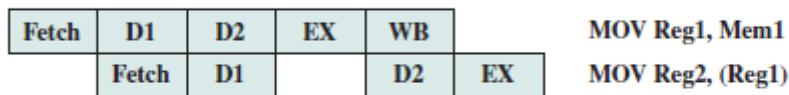
- Desarrollo:

- La instrucción DBcc se emplea como control de los bucles una vez finalizada cada iteración. La condición cc hace referencia a la última operación antes de la instrucción DBcc, en este caso CMPM.L. Si la condición es verdadera → Fin de bucle y sigue la secuencia del programa. Si la condición es falsa decrementa el contador de iteraciones y salta al comienzo del bucle. Palabras tipo .L (Large) de 4 bytes. D1=0xFF. D1-1=0xFFFFFFFF. A0 puntero a string → (A0) indirección → (A0)+ incrementa el puntero en una palabra en cada ejecución.
- Los dos punteros apuntan a memoria cuyo contenido es \$AAAA por lo tanto la comparación da como resultado EQUAL. La condición de la instrucción DBcc es NE, not equal, por lo tanto es FALSE y sí se ejecuta el bucle. Se ejecuta 0xFF+1 veces hasta llegar el contador D1=-1. Ultima dirección del puntero A0 → 0x4000+0xFFpalabras+1palabra=0x4000+4x(0xFF)bytes+4bytes → $2^2 \times (0xFF)$ equivale a desplazar 0xFF dos bits a la izda = 0x3FC → 0x4000+0x3FC+4=0x4400. Puntero A1 → 0x5000+0x3FC+4=0x5400.
- Todas las comparaciones dan como resultado distinto de cero → NE → por lo tanto TRUE → Unicamente se ejecuta una iteración. D1=0xFF-1=0xFE; A0=A0+1palabra=0x4004 y A1=0x5004

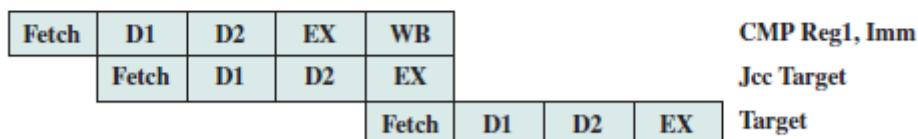
- 12.15 Redraw Figures 12.19c (14.21c), assuming that the conditional branch is not taken



(a) No data load delay in the pipeline



(b) Pointer load delay



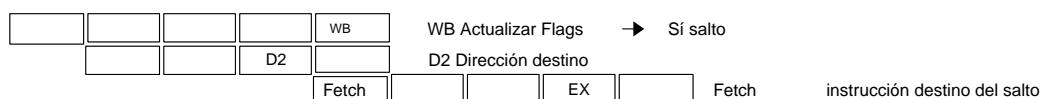
(c) Branch instruction timing

Figure 14.21 80486 Instruction Pipeline Examples

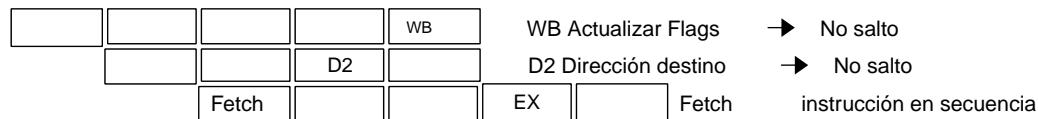
- Desarrollo:

- 80486: 32 bits
- Etapas del cauce (pipeline) de instrucciones: FE-D1-D2-Ex-WB.
- Fetch: Captura de la instrucción
- D1: Decodifico Cod.Op y Modo Direccionalamiento
- D2: Operaciones para el cálculo de la Dirección Efectiva del Operando
- EX: Operaciones ALU y acceso a operandos
- WB: EFLAGS, Resultados en Reg y Mem(Caché y MP)
- Figura 12.19 b) La 1^a ins. en EX lee el operando de la memoria y la 2^a en D2 accede a memoria para leer del puntero la dirección del operando.

- Figura 12.19 c) La instrucción CMP actualiza reg flags. La instrucción Jcc en D2 ya tiene la dirección de salto aunque actualiza el Contador de Programa en EX. La inst 3^a después de D2 de Jcc ya puede ser capturada.
- Sí salto:
 - La cpu realiza la captura de la 3^a instrucción (Fetch) inmediatamente después de la captura de la segunda pero dicha captura es errónea ya que ha capturado la siguiente en secuencia a la 2^a y no la instrucción target. La captura de la instrucción destino se ha de realizar cuando se conozca la dirección dónde se encuentra dicha instrucción.



- No salto:
 - La CPU captura las 3 instrucciones en secuencia y no se equivoca en la 3^a ya que no hay salto.



- 12.16 Table 14.5 summarizes statistics from [MACD84] concerning branch behavior for various classes of applications. With the exception of type 1 branch behavior, there is no noticeable difference among the application classes. Determine the fraction of all branches that go to the branch target address for the scientific environment. Repeat for commercial and systems environments.

Table 14.5 Branch Behavior in Sample Applications

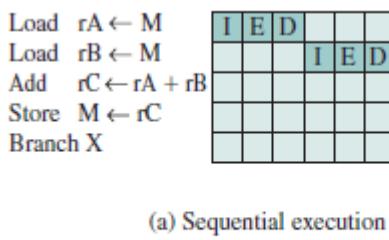
| Occurrence of branch classes: | | | |
|---|-------|-------|-------|
| Type 1: Branch | 72.5% | | |
| Type 2: Loop control | 9.8% | | |
| Type 3: Procedure call, return | 17.7% | | |
| Type 1 branch: where it goes | | | |
| Unconditional—100% go to target | 20% | 40% | 35% |
| Conditional—went to target | 43.2% | 24.3% | 32.5% |
| Conditional—did not go to target (inline) | 36.8% | 35.7% | 32.5% |
| Type 2 branch (all environments) | | | |
| That go to target | 91% | | |
| That go inline | 9% | | |
| Type 3 branch | | | |
| 100% go to target | | | |

- Desarrollo:
 - tipo1=72.5 ; tipo2=9.8 ; tipo3=17.7
 - tipo1: hay 3 casos dentro del tipo1 (1/3 salta incondicional, 1/3 salta condicional, 1/3 no salta)
 - tipo2: 91% salta, el 9% no salta

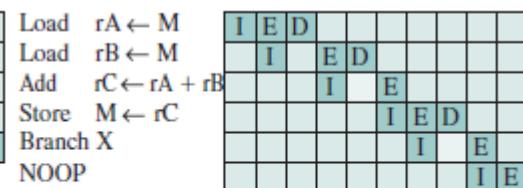
- tipo3: saltan todas
- Salto a destino = tipo1x[(0.2+0.4+0.35)x100/100+(43.2+24.3+32.5)x1/3]+ tipo2x0.91+tipo3x100/100 =
- Saltos to taget por aplicaciones
 - científica=tipo1x[(0.2)x100/100+(43.2)x1/3]+ tipo2x0.91+tipo3x100/100=0.724 → El 72% de los saltos de una aplicación científica son a destino.
 - comercial =tipo1x[(0.4)x100/100+(24.3)x1/3]+ tipo2x0.91+tipo3x100/100=0.732
 - sistema =tipo1x[(0.35)x100/100+(32.5)x1/3]+ tipo2x0.91+tipo3x100/100=0.756

10.14. Capítulo 13: Reduces Instruction Set Computer (Capítulo 15 en 9^aEd)

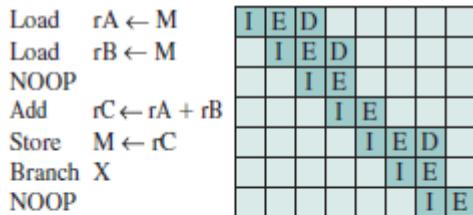
- 13.3 We wish to determine the execution time for a given program using the various pipelining schemes discussed in Section 13.5. Let N = number of executed instructions, J = number of jump instructions, D = number of memory accesses. For the simple sequential scheme (Figure 13.6a) for a RISC architecture, the execution time is $2N+D$ stages. Derive formulas for two-stage, three-stage, and four-stage pipelining.



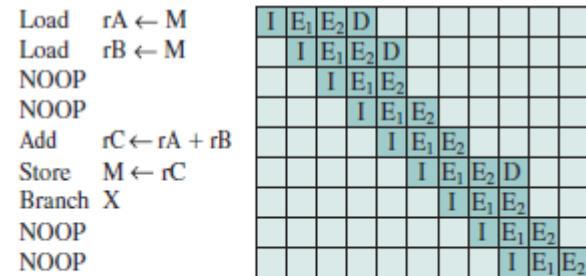
(a) Sequential execution



(b) Two-stage pipelined timing



(c) Three-stage pipelined timing

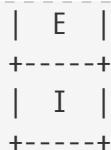


(d) Four-stage pipelined timing

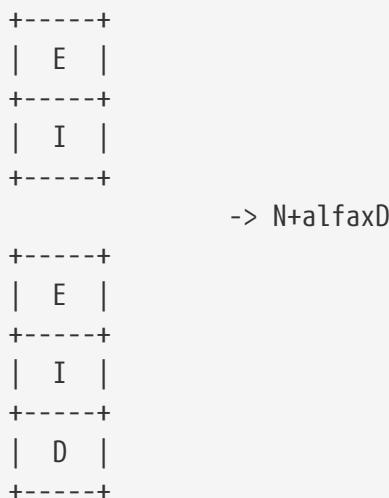
Figure 15.6 The Effects of Pipelining

- Desarrollo:
 - CAUCE SEGMENTADO: I → captación de la instrucción. E → operaciones ALU con Reg. u obtención de la dirección efectiva. D → Transferencia Mem <→ Reg.
 - Cada instrucción tiene por lo menos dos etapas: E e I. En cambio la etapa D no la tienen todas las instrucciones (sólo load y store entre reg y mem)
- 1. Figura apartado a → $T=Nx(t_e+t_i)+t_dxD$; si $t_i=t_e=t_d=t \rightarrow T=[2N+D]t$
- 2. Figura apartado b → Cauce segmentado → $k=2 \rightarrow$ Sin instrucciones de salto → $T_{k,n}=[k+(n-1)]t \rightarrow T_{2,n}=[2+(N-1)]t$.
 - Sólo es posible un acceso a memoria en cada etapa.
 - I es una etapa, E y D forman una única etapa.
 - E e I se solapan → N etapas E|I

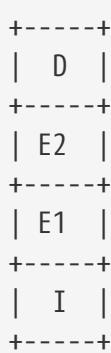
-----+



- d. D no se solapa → D etapas
- e. En la fase E de la instrucción Branch se calcula la dirección de salto por lo que la fase I de la instrucción destino no puede coincidir con dicha fase E. Se soluciona con un instrucción de no operación NOOP.
- f. Los saltos originan un NOOP → una etapa de retardo más que añadir
- g. $T=(N+D+J)t$
3. Figura apartado c → k=3 etapas
- En una etapa son posibles dos accesos a memoria.
 - La 2^a instrucción load carga el dato en el registro en la etapa D por lo que no puede coincidir con la ejecución de la instrucción suma.
 - D,E e I se solapan si no hay dependencia de datos
 - Si hay dependencias D no se solapa por lo que hay una fracción de las D instrucciones que hay que sumar.

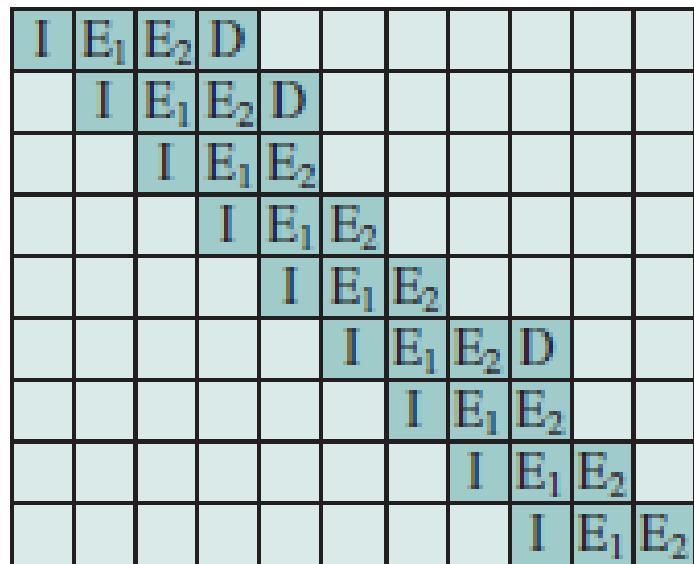


- e. $T=(N+alfa*D+J)t \rightarrow J$ noops
4. Figura apartado d → k=4 etapas
- Dividimos E en E1 (lectura RPG) y E2 (ALU y escritura RPG)



- b. El solape de D con dependencia de datos introduce un retardo y J otros dos según la figura.
- c. $T = (N + \alpha * D + 2J)t$
- 13.4 Reorganize the code sequence in Figure 13.6d to reduce the number of NOOPs. Figura del ejercicio 13.3 d).

Load $rA \leftarrow M$
 Load $rB \leftarrow M$
 NOOP
 NOOP
 Add $rC \leftarrow rA + rB$
 Store $M \leftarrow rC$
 Branch X
 NOOP
 NOOP



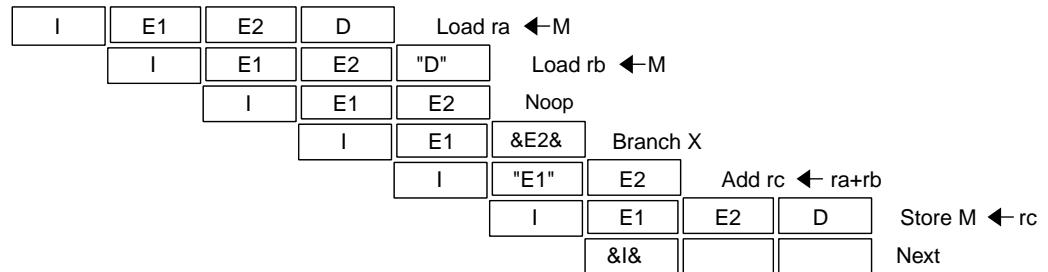
(d) Four-stage pipelined timing

- La instrucción Branch la ejecutamos en la posición del 2º NOOP
- Las dos instrucciones anteriores al salto (Add y Store) en lugar de los 2 NOOP después del salto Branch.

```

Load ra<-M
Load rb<-M
Noop
Branch X
Add rc<-ra+rb
Store M<-rc
Next
  
```

- Mientras se ejecutan Add y Store se calcula la dirección X



- Dependencias: Una vez que se carga rb en la fase "D" ya se puede leer rb en "E1"

- Dependencias: Una vez que se calcula X en &E2& ya se puede capturar la instrucción Next durante &I&
- 13.5 Consider the following code fragment in a high-level language:

```
for I in 1...100 loop
    S ← S + Q(I).VAL
end loop;
```

- Assume that Q is an array of 32-byte records and the VAL field is in the first 4 bytes of each record. Using x86 code, we can compile this program fragment as follows:

```
MOV ECX, 1 ;use register ECX to hold I
LP: IMUL EAX, ECX, 32 ;get offset in EAX
    MOV EBX, Q[EAX] ;load VAL field
    ADD S, EBX ;add to S
    INC ECX ;increment I
    CMP ECX, 101 ;compare to 101
    JNE LP ;loop until I = 100
```

- This program makes use of the IMUL instruction, which multiplies the second operand by the immediate value in the third operand and places the result in the first operand (see Problem 10.13). A RISC advocate would like to demonstrate that a clever compiler can eliminate unnecessarily complex instructions such as IMUL. Provide the demonstration by rewriting the above x86 program without using the IMUL instruction.
 - Array Q: 100 registros (estructuras) de 32 bytes cada uno.
 - VAL field: primeros 4 bytes del registro.

```
typedef struct {int VAL;....} Data;
Data Q[100];
```

- c. Q(i).VAL : El campo VAL de cada registro Q(i)

```
## El bucle suma los campos VAL de los 100 registros
MOV ECX, 1 ;use register ECX to hold I #Indice de registro del array Q
LP: IMUL EAX, ECX, 32 ;get offset in EAX #EAX: dirección relativa del campo VAL
    del registro al que apunta el índice ECX del array Q
    #Cada 32 bytes un registro
    MOV EBX, Q[EAX] ;load VAL field #Q en ensamblador se estructura en
    bytes. 100registrosx32bytes/registro=3200registros
    ADD S, EBX ;add to S #Suma de los 4 bytes del campo VAL
    INC ECX ;increment I #siguiente registro
    CMP ECX, 101 ;compare to 101 #ultimo+1 registro?
    JNE LP ;loop until I = 100 #Siguiente interacción si no ultimo
```

- d. Multiplicar por 2^x equivale a un desplazamiento de x bits a la izda
 - Multiplicar por 32 → $x2^5 \rightarrow$ desplazar 5 bits a la izda : shl \$5,%ecx
- 13.6 Consider the following loop:

```

S := 0;
for K := 1 to 100 do
  S := S - K;

```

- A straightforward translation of this into a *generic* assembly language would look something like this:

```

LD R1, 0          ;keep value of S in R1
LD R2, 1          ;keep value of K in R2
LP SUB R1, R1, R2 ;S := S - K
BEQ R2, 100, EXIT ;done if K = 100      #Branch Equal
ADD R2, R2, 1     ;else increment K
JMP LP            ;back to start of loop

```

- A compiler for a *RISC* machine will *introduce* delay slots into this code so that the processor can employ the *delayed branch mechanism*. The JMP instruction is easy to deal with, because this instruction is always followed by the SUB instruction; therefore, we can simply place a copy of the SUB instruction in the delay slot after the JMP. The BEQ presents a difficulty. We can't leave the code as is, because the ADD instruction would then be executed one too many times. Therefore, a NOP instruction is needed. Show the resulting code.

- Desarrollo:

- Delayed Branch

Time →

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----------------|---|---|---|---|---|---|---|
| 100 LOAD X, rA | I | E | D | | | | |
| 101 ADD 1, rA | | I | E | | | | |
| 102 JUMP 105 | | | I | E | | | |
| 103 ADD rA, rB | | | | I | E | | |
| 105 STORE rA, Z | | | | | I | E | D |

(a) Traditional pipeline

| | I | E | D | | | | |
|-----------------|---|---|---|---|---|---|---|
| 100 LOAD X, rA | I | E | D | | | | |
| 101 ADD 1, rA | | I | E | | | | |
| 102 JUMP 106 | | | I | E | | | |
| 103 NOOP | | | | I | E | | |
| 106 STORE rA, Z | | | | | I | E | D |

(b) RISC pipeline with inserted NOOP

| | I | E | D | | | | |
|-----------------|---|---|---|---|---|---|--|
| 100 LOAD X, Ar | I | E | D | | | | |
| 101 JUMP 105 | | I | E | | | | |
| 102 ADD 1, rA | | | I | E | | | |
| 105 STORE rA, Z | | | | I | E | D | |

(c) Reversed instructions

Figure 15.7 Use of the Delayed Branch

- La primera gráfica es un salto normal y las otras dos retardado. La última gráfica supone una instrucción menos
- b. El programa presenta dos instrucciones de salto con retardo: BEQ y JMP
- c. JMP LP
 - Salto incondicional. En el programa original se ejecutará el salto después de SUB R1, R1, R2. Solución:

```
JMP LP          ;back to start of loop
ADD R2, R2, 1   ;else increment K
```

- De esta manera el salto se ejecuta después del ADD R2, R2, 1
- d. BEQ R2, 100, EXIT
 - Salto condicional
 - Si lo dejamos como está el salto será después del ADD R2, R2, 1. Solución:

```
BEQ R2, 100, EXIT ;done if K = 100      #Branch EQual
LP SUB R1, R1, R2 ;S := S - K
```

- Ahora el salto condicional se realizará después de la resta SUB
- e. Solución 1^a:

```
LD R1, 0          ;keep value of S in R1
LD R2, 1          ;keep value of K in R2
LP BEQ R2, 100, EXIT ;done if K = 100      #Branch EEqual
      SUB R1, R1, R2 ;S := S - K
      JMP LP          ;back to start of loop
      ADD R2, R2, 1   ;else increment K
```

- Tiene el defecto de que si R2=100 se ejecuta también SUB modificando el valor final de R1 y R2
- f. Solución Definitiva:

```
LD R1, 0          ;keep value of S in R1
LD R2, 1          ;keep value of K in R2
LP BEQ R2, 100, EXIT ;done if K = 100      #Branch EEqual
      NOP
      ADD R2, R2, 1   ;else increment K
      JMP LP          ;back to start of loop
      SUB R1, R1, R2 ;S := S - K
```

- 13.7 A RISC machine may do both a mapping of symbolic registers to actual registers and a *rearrangement* of instructions for pipeline efficiency. An interesting question arises as to the order in which these two operations should be done. Consider the following program fragment:

```
LD SR1,A          ;load A into symbolic register 1
LD SR2,B          ;load B into symbolic register 2
```

```

ADD SR3, SR1, SR2 ;add contents of SR1 and SR2 and store in SR3
LD SR4, C
LD SR5,D
ADD SR6, SR4, SR5

```

1. First do the register mapping and then any possible instruction reordering. How many machine registers are used? Has there been any pipeline improvement?
 2. Starting with the original program, now do instruction reordering and then any possible mapping. How many machine registers are used? Has there been any pipeline improvement?
- Desarrollo: .
 - 13.9 In many cases, common machine instructions that are not listed as part of the MIPS instruction set can be synthesized with a single MIPS instruction. Show this for the following:
 1. Register-to-register move
 2. Increment, decrement
 3. Complement
 4. Negate
 5. Clear
 - 13.11 SPARC is lacking a number of instructions commonly found on CISC machines. Some of these are easily simulated using either register R0, which is always set to 0, or a constant operand. These simulated instructions are called pseudoinstructions and are recognized by the SPARC compiler. Show how to simulate the following pseudoinstructions, each with a single SPARC instruction. In all of these, src and dst refer to registers. (Hint: A store to R0 has no effect.)
 1. MOV src, dst
 2. COMPARE src1, src2
 3. TEST src1
 4. NOT dst
 5. NEG dst
 6. INC dst
 7. DEC dst
 8. CLR dst
 9. NOP
 - 13.12 Consider the following code fragment:

```

if K > 10
  L := K + 1
else
  L := K - 1;

```

- A straightforward translation of this statement into SPARC assembler could take the following form:

```

sethi %hi(K), %r8          ;load high-order 22 bits of address of location
                                ;K into register r8
ld [%r8 + %lo(K)], %r8    ;load contents of location K into r8
                                ;compare contents of r8 with 10

```

```
ble L1           ;branch if (r8) <= 10
nop
sethi %hi(K), %r9
ld [%r9 + %lo(K)], %r9 ;load contents of location K into r9
inc %r9          ;add 1 to (r9)
sethi %hi(L), %r10
st %r9, [%r10 + %lo(L)] ;store (r9) into location L
b L2
nop
L1: sethi %hi(K), %r11
ld [%r11 + %lo(K)], %r12 ;load contents of location K into r12
dec %r12         ;subtract 1 from (r12)
sethi %hi(L), %r13
st %r12, [%r13 + %lo(L)] ;store (r12) into location L
L2:
```

- The code contains a nop after each branch instruction to permit delayed branch operation.
 1. Standard compiler optimizations that have nothing to do with RISC machines are generally effective in being able to perform two transformations on the foregoing code. Notice that two of the loads are unnecessary and that the two stores can be merged if the store is moved to a different place in the code. Show the program after making these two changes.
 2. It is now possible to perform some optimizations peculiar to SPARC. The nop after the ble can be replaced by moving another instruction into that delay slot and setting the annul bit on the ble instruction (expressed as ble,a L1). Show the program after this change.
 3. There are now two unnecessary instructions. Remove these and show the resulting program

IV Autoevaluación Teoría

Chapter 11. Teoría: Cuestionario

11.1. Arquitectura von Neumann

1. Qué función tiene el programa ensamblador
2. Qué tipo de lenguaje es el lenguaje máquina
3. En qué año se desarrolló la máquina IAS
4. Cuál es el formato de instrucción de la máquina IAS
5. Indica dos instrucciones de la máquina IAS en dos lenguajes diferentes.
6. Cuál es el programa de demostración en lenguaje ensamblador del emulador, de la máquina IAS, IASSim.
7. Cuáles son las dos primeras instrucciones en lenguaje máquina del programa demo del emulador IASSim.
8. En qué consiste la arquitectura ISA
9. Emplea la palabra abstracción en algún concepto de la asignatura
10. Indica dos microórdenes
11. Que unidad genera las microórdenes
12. Hacia qué elementos van dirigidas las microórdenes
13. Qué significa 1MB
14. Cuáles son los registros que utiliza la UC de la IAS
15. Cuáles son los registros que utiliza la ALU de la IAS
16. Cuáles son las fases del ciclo de instrucción
17. Qué unidad o unidades implementan el ciclo de instrucción.
18. De qué categoría de memoria lee los programas la CPU
19. Qué es un proceso
20. Define la arquitectura de la CPU utilizando por lo menos los términos siguientes:
 - ciclo de instrucción
 - instrucción máquina
 - almacenamiento
 - interpretación
 - ruta de datos
 - registro
 - microórdenes

V Guiones de Prácticas: Programación Ensamblador x86

Chapter 12. Introducción a la Programación en Lenguaje Ensamblador AT&T x86-32

12.1. Introducción

12.1.1. Objetivos

- Introducción a la **programación de bajo nivel** mediante los lenguajes *C* y *ensamblador AT&T* para la arquitectura **x86 de 32 bits** de Intel.
- Utilización de herramientas de desarrollo de sw de bajo nivel como el toolchain (compilador, ensamblador, linker) y el depurador GDB, libres de la fundación GNU.
- Desarrollo de una workstation mediante la instalación de herramientas de desarrollo en un entorno GNU/linux/x86 en una computadora personal.
- Comprender el funcionamiento de la computadora desde el punto de vista de un programador de bajo nivel.
- Relacionar las propiedades de un lenguaje de alto nivel con un lenguaje de bajo nivel.
- El lenguaje ensamblador puede ser utilizado como una herramienta para analizar la arquitectura y el funcionamiento de la computadora. No es el objetivo central de esta asignatura ser un experto en lenguajes de programación de bajo nivel ni en el desarrollo de algoritmos, aunque sí un nivel muy básico.

12.1.2. Requisitos

Teóricos

- Conocimientos muy básicos de una arquitectura ISA: Arquitectura modelo Von Neumann(Microarquitectura CPU, arquitectura Memoria Principal, ciclo de instrucción), representación de datos y operaciones aritméticas, formato de instrucciones y programación básica en lenguaje ensamblador y lenguaje máquina con la máquina IAS de John von Neumann, la sintaxis del lenguaje ensamblador AT&T x86 y la arquitectura básica del procesador x86-64 y x86-32 de intel.

Prácticos

- Tener configurada la "Plataforma de Desarrollo" GNU/linux(AMD64)/x86_64(Intel ó AMD) con las herramientas apropiadas.
- Conocimientos de programación imperativa en lenguaje C y del "Lenguaje de Transferencia entre Registros" RTL, manejo básico del entorno GNU/linux y una herramienta de edición.
- Estar dado de alta en el sitio de la asignatura en servidor miaulario.
- Haber realizado la simulación de ejecución de un programa en lenguaje ensamblador y lenguaje máquina. Por ejemplo en la máquina IAS de John von Neumann mediante el emulador Web IASSim



Haber analizado la sintaxis y estructura de un programa sencillo en lenguaje ensamblador AT&T x86, compilado, ejecutado y analizada la ejecución del programa paso a paso mediante el depurador GDB mediante la información de [Apéndice: Practicando la Programación desde el principio](#) y el pequeño tutorial [Apéndice: empezando ASM](#) y [Apéndice: Empezando C](#)

12.2. LEEME

- Lectura del guión de prácticas y de los capítulos 1 y 2 del Libro Programming from the Ground-Up.

- Apuntes y Libro de Texto



La memoria se realiza en tiempo real durante la realización de la práctica y es la única documentación impresa que se utilizará el día del examen, por lo que debe de estar bien documentada. La memoria se entrega unos días después de la realización de la memoria a través de MiAulario/TAREAS siguiendo las indicaciones especificadas en [Apéndice: Contenido y Formato de la Memoria](#) y en la fecha indicada en MiAulario/TAREAS.

- Abrir en el Escritorio de la computadora un documento texto que será la **MEMORIA** donde se irá añadiendo toda la información obtenida durante toda la sesión de prácticas
- **Evaluación:** sistema de evaluación
- **Programación :** metodología

12.3. Cuestiones

- "Autoevaluación de Prácticas" opcional: Prácticas: Cuestionario

12.4. Estación de Trabajo

- Anotar las características de la [Plataforma de Desarrollo](#) en el Documento Memoria.

12.5. Programación sum1toN.c

12.5.1. Algoritmo

- Desarrollar un programa en lenguaje C que realice la suma $\sum_{i=1}^N i$ cuyo resultado es $N(N + 1)/2$ con la arquitectura *i386* utilizando el [método de programación](#) de descripción inicial en lenguaje pseudocódigo y organigrama.

12.5.2. Edición del Módulo fuente: sum1toN.c

- Editar el programa descargando el módulo fuente "sum1toN.c" de miaulario y añadiendo los comentarios apropiados.
- Cabecera con información complementaria:

```
/*
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
y equivalente al programa sum1toN.s en lenguaje ensamblador AT&T
Lenguaje: C99
Descripción: Suma de los primeros 5 números naturales
Entrada: Definida en una variable
Salida: Sin salida periférica<<<s
Compilación: gcc -m32 -g -o sum1toN sum1toN.c
S.O: GNU/linux 5.4.0-128-generic ubuntu 20.04 x86-64
Librería: /usr/lib/x86_64-linux-gnu/libc.so
PC: ThinkPad L560 product: 20F1S0H400 serial: MP15YSW7
```

```
CPU:           Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz width:64 bits
Compilador:    gcc version 9.4.0
Ensamblador:   GNU assembler version 2.34
Linker/Loader: GNU ld (GNU Binutils for Ubuntu) 2.34
Asignatura:    Estructura de Computadores
Fecha:         25/09/2022
Autor:         Cándido Aramburu
*/
```

- Programa:

```
// Módulo Principal
void main (void) {
    //Declaración de variables locales e inicialización de los parámetros del bucle
    int sum=0,n=5;
    //Bucle que genera los sumandos y realiza la suma
    while(n>0){      //Condición de salida del bucle cuando el sumando es negativo
        sum+=n;
        n--;          //Actualización del sumando
    }
}      //Si las instrucciones se han ejecutado sin interrupción ni fallo main()
devuelve el valor cero al sistema operativo.
```

12.5.3. Compilación

- Seguir los pasos del proceso de compilación común a todas las sesiones.
 - `gcc -m32 -g -o sum1toN sum1toN.c`
 - listar los ficheros: `ls -l sum1toN*`. ¿Qué representa cada uno de ellos?
 - `gcc --save-temps -m32 -g -o sum1toN sum1toN.c`
 - listar los ficheros: `ls -l sum1toN*`. ¿Qué representan los nuevos ficheros?

12.5.4. Análisis de los módulos

- Análisis para comprobar los distintos módulos:

- `file sum1toN.c`
- `file sum1toN.i`
- `file sum1toN.s`
- `file sum1toN.o`
- `file sum1toN`

12.5.5. Ejecución

- `./sum1toN` : llamada al módulo binario ejecutable
- `echo $?` : visualización del valor devuelto por el programa *sum1toN* al sistema operativo linux.
 - El valor cero se utiliza como indicador de que el programa se ha ejecutado sin ningún tipo de contratiempo.

12.5.6. Depuración

introducción

- La ejecución del programa paso a paso, instrucción a instrucción, permite un análisis minucioso de bajo nivel de la ejecución del programa pudiendo detener el programa y volcar el valor de las variables en la memoria principal, estado de los registros de la cpu, etc.
- El Depurador GDB (GNU DeBugger) permite la ejecución a paso a paso y análisis de la memoria mediante un repertorio de comandos propios del depurador.

Generación de la tabla de símbolos

- `gcc -m32 -g -o sum1toN sum1toN.c`
 - opción **-g**: Inserta la "Tabla de Símbolos" en el módulo binario ejecutable.
- En la línea de comandos emplear el TABULADOR TAB para Completar los nombres: `gcc -m32 -g -o sum1TAB suTAB`

gdb

- Abrir el depurador: `gdb` para comenzar la sesión de depuración.
 - Ventana con el prompt (**gdb**): línea de comandos propios del debugger que serán interpretados por el GDB.

abrir una nueva ventana

- (gdb) `layout src` ó `Control-x Control-a`
- navegar entre las dos ventanas: `Control-x` o y sino también `focus src` y `focus cmd`
- `help focus`

Logging



El logging hay que configurarlo con las dos ventanas abiertas: la ventana de comandos y la ventana del código source. Si se hace teniendo solamente la ventana de comandos, luego al abrir más ventanas deja de loggear.

- Salvar toda la sesión de depuración en el fichero `sum1toN_gdb_c.txt`
 - Entradas → (gdb) `set trace-commands on`,
 - Salidas → (gdb) `set logging file sum1toN_gdb_c.txt`
 - Activación → `set logging on`

```
Copying output to sum1toN_gdb_c.txt.  
Copying debug output to sum1toN_gdb_c.txt.
```

- Cada vez que salga y entre en el debugger en la misma sesión de prácticas y utilice el mismo fichero histórico `sum1toN_gdb_c.txt` hay que realizar la configuración anterior con las dos ventanas abiertas.

Comandos linux desde la línea de comandos gdb

- `shell ls -l sum1toN_gdb_c.txt`
 - `shell date`

- `shell pwd`
- `shell ls`
- `shTAB` : Emplear el TABULADOR TAB para Completar los nombres.
- `shell daTAB`

COMPROBAR el histórico de la sesión de depuración

- Abrir una consola y en la carpeta de trabajo comprobar que existe el fichero `sum1toN_gdb_c.txt` y que contiene los comandos gdb ejecutados anteriormente.



Es muy frustrante darse cuenta después de dos horas de trabajo que no se ha guardado todo el trabajo

Ventanas

- Layout: `C-x a` → por defecto dos ventanas: módulo fuente y línea de comandos.
- Navegador ventanas: `C-x o`
- Navegar por el histórico de comandos
 - Logging histórico de comandos: Activar la ventana de comandos del GDB. Navegar con las teclas flecha arriba/abajo.

Ayuda

- `help shell` ó `h shell`

Cargar módulo objeto ejecutable

- Cargar el módulo objeto binario que contiene la Tabla de Símbolos: `file sum1toN`
- módulo fuente con los símbolos asociados a la Tabla de Símbolos: `info sources`



Observar que el depurador confirma la existencia de la tabla de símbolos, imprescindible para la depuración.

Ejecución paso a paso

- punto ruptura en la entrada al programa: `break main`
- Ejecución: `run` hasta el punto de ruptura cuya línea NO se ejecuta → Aparece el código fuente.
- Next source line: `next, n`
- Next 5 source lines: `n 5, print sum, p sum`
- Comenzar desde el principio nuevamente: `run` ó `start`
- Continuar hasta el próximo punto de ruptura: `continue, c`
- `run, n, RETURN, RET, RET, RET..hasta salir del bucle.., p sum, c`

Bucle

- ¿cómo salir de un bucle de cientos o miles de iteracciones hasta la siguiente instrucción fuera del bucle?
- `run, until, RET,RET,RET..hasta salir del bucle.., p sum, c`

Análisis de la memoria principal DRAM

- imprimir el contenido de variables y sus direcciones en la memoria principal
- `print n, p n, p /t n, p /x n, ptype n, whatis n,p &n`
- `print symbol` : symbol es el nombre de la variable, no su dirección.
- `p $eax`
- `p $ebx`
- `p $ecx`
- `info registers`

Desensamblar

- Desensamblar: ingeniería inversa . Convierte el código binario en código ensamblador
 - `layout split`
 - Next machine instruction: `ni`, RET, RET, RET, RET, `until`, RET,..hasta salir del bucle
 - Ejecuta instrucciones máquina (observar ventana con el código ensamblador)

Salir

- `exit`



Comprobar que el contenido del fichero `sum1toN_gdb_C.txt` es correcto.

12.5.7. Recordatorio: Documento Memoria

- Ir salvando el trabajo y comentándolo según se va realizando.
- En la consola abrimos el fichero `sum1toN_gdb_C.txt` que contiene todos los comandos utilizados con sus volcados.
- Guardar el contenido de `sum1toN_gdb_C.txt` en el Documento Memoria añadiendo los comentarios necesarios.

12.5.8. Continuamos con más ejercicios

Jugar con el módulo fuente

- Cambios en el Módulo fuente en lenguaje C.
 - Cambiar el tamaño de los datos con alguno de los siguientes tipos:
 - char,short,int,long
 - Cambiar el formato de los números con alguno de los siguientes bases:
 - decimal, hexadecimal, octal, binario → prefijos 0x, 0, 0b → 0x5, 05, 0b5
 - Compilar y ejecutarlos. Indicar en la memoria si da o no algún error

Jugar con el depurador

- GDB
 - Cambiar en el módulo fuente el tamaño de las variables a `char` y la sentencia `sum+=n` por la sentencia `sum-=n`.
 - Compilar el módulo fuente con la opción de inserción de la tabla de símbolos

- Abrir el depurador y cargar el módulo binario
- Ejecutar en modo paso a paso observando las sumas parciales con lo siguientes comandos:
 - `x /1db $sum, x /1tb $sum, x /1ob $sum, x /1xb $sum`
 - indicar para comando el resultado
 - Con la ayuda de `help x` explica el significado de `1db, 1tb, 1ob, 1xb.`

12.6. Programación sum1toN.s

12.6.1. Algoritmo

- Desarrollar un programa en lenguaje ensamblador AT&T con la arquitectura *i386* que realice la suma $\sum_{i=1}^N i$ cuyo resultado es $N(N + 1)/2$ utilizando el [método de programación](#) de descripción inicial en lenguaje pseudocódigo y organigrama.

12.6.2. Edición del Módulo fuente: sum1toN.s

- Descargar el módulo fuente "sum1toN.s" de miaulario y añadir los comentarios apropiados.
- x86 es la arquitectura de Intel de 32 bits
- i386 significa en linux: arquitectura x86-32
- Lenguaje ensamblador AT&T de GNU para la arquitectura i386 → lenguaje GNU as → lenguaje gas

```

#### 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 i386 -> GNU as ->
gas -> AT&T
#### Es el programa en lenguaje AT&T i386 equivalente a sum.ias de la máquina IAS de
von Neumann
#### gcc -m32 -g -nostartfiles -o sum1toN sum1toN.s
#### Ensamblaje as --32 --gstabs sum1toN.s -o sum1toN.o
#### linker -> ld -melf_i386 -o sum1toN sum1toN.o

      ## Declaración de variables
## SECCION DE DATOS
.section .data

n:    .int 5

.global _start

## Comienzo del código
## SECCION DE INSTRUCCIONES
.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 segúñ
convenio ABI i386

## salida
mov $1, %eax # código de la llamada al sistema operativo: subrutina exit
int $0x80 # llamada al sistema operativo para que ejecute la subrutina
segúñ el valor de EAX

.end

```

12.6.3. Compilación

- Seguir los pasos de la [compilación](#) de un módulo en lenguaje ensamblador.
 - `gcc -m32 -g -o sum1toN sum1toN.s`

12.6.4. Ejecución

- `./sum1toN`
- `echo $?`

12.6.5. Análisis del módulo Fuente

- Leer en las hojas de referencia rápida el [Programa Ejemplo Minimalista](#)

12.6.6. Depuración

Inicio

- Depurador GDB: GNU DeBugger.
- `gcc -m32 -g -o sum1toN sum1toN.s`
- `gdb`
- `C-x a`
- `C-x o`
- `set trace-commands on`
- `set logging file sum1toN_gdb_asm.txt`
- `set logging on`
- `shell ls -l sum1toN_gdb_asm.txt`

Arrancar el programa

- `file sum1toN`
- `info sources`
- `b _start`

- `run`

Analizar símbolos en memoria

- `ptype n`
- `p n`
 - `x address` : examine main memory address . Devuelve el contenido de la dirección de memoria address → `&symbol` donde symbol es el nombre de la variable.
- `x &n, x n, x /1bw &n, x /1xw &n, x /4xw &n`
- `b bucle`
- `c`
- `start`
- `c`
- `n`

Registros

- `p $ecx`
- `p $edx`
- `until`
- `p $ecx`
- `p $edx`
- `info registers`

Instrucciones máquina

- `layout split`
- `start`
- `c`

Fin

- `exit`
- En la consola abrimos el fichero `gedit sum1toN_gdb_asm.txt` que contiene todos los comandos utilizados con sus volcados.
- Guardar el contenido de `sum1toN_gdb_asm.txt` en el Documento Memoria añadiendo los comentarios necesarios.

12.7. Arquitectura amd64

- Ejemplo en el apéndice : [sum1toN.s](#)
 - Analizar el código, compilarlo, ejecutarlo y comprobar mediante el debugger que efectivamente los registros son de 64 bits.

Chapter 13. Representación de los Datos

13.1. Introducción

13.1.1. Objetivos

- Programación:
 - Desarrollar programas que almacenen y procesen distintos tipos de datos como enteros con signo, caracteres, arrays, strings de distintos tamaños como 1 byte, 2 bytes, 4 bytes, etc..
 - Empleo de los sufijos de los mnemónicos: analizar el tamaño de los operandos según el sufijo del mnemónico empleado, el tamaño del operando registro y el tipo de operando en memoria.
 - Emplear distintos tipos de modos de direccionamiento (inmediato, directo, indirecto, indexado) de acceso a los operandos
 - Empleo de Macros mediante directivas.
 - Concepto de llamada al sistema operativo.
- Análisis:
 - Comprobación del tipo de alineamiento *little endian* de los datos almacenados en memoria
 - Analizar el contenido de la memoria como números con signo, caracteres, arrays y strings: tipos y tamaños de los operandos numéricos y de los operandos alfanuméricos
 - Realizar la operación de desensamblaje para comprobar el lenguaje máquina del módulo ejecutable cargado en la memoria principal

13.1.2. Módulos fuente: características

- **datos_size.s**
 - Declaración del tamaño de los operandos:
 - Mediante las directivas (.byte, .2byte, .short, etc ..)
 - Declaración de **arrays** de datos numéricos mediante directivas (.short,.int, etc ..)
 - Declaración de datos **alfanuméricos** mediante:
 - Directivas del ensamblador (.ascii, .asciiz, .string, etc)
 - Empleo de Macros
- **datos_sufijos.s**
 - Acceso a los operandos mediante instrucciones con los **sufijos** (b,w,l,q)
- **datos_direccionamiento.s**
 - Diferentes Modos de direccionamiento de los operandos: inmediato, directo, indirecto, indexado

13.1.3. Requisitos

- Teoría: representación de datos, formato de instrucciones y repertorio ISA de la arquitectura X86.
 - Almacenamiento con **alineamiento interno** de bytes "little endian"
- Práctica previa: Introducción a la programación en lenguaje ensamblador AT&T x86-32
- Conceptos del lenguaje de **programación C**:
 - Punteros, array, string y operación de casting.

13.2. LEEME

- Lectura del guión de prácticas y del capítulo 3 del Libro **Programming from the Ground-Up**.
- [Apuntes y Libro de Texto](#)
- [Documentación Memoria](#): Contenido y Formato de la Memoria
- [Evaluación](#): sistema de evaluación
- [Plataforma de Desarrollo](#) : configuración de la computadora personal
- [Programación](#) : metodología

13.3. Cuestiones Opcionales

- "Autoevaluación de Prácticas" opcional: [Prácticas: Cuestionario](#)
 - Los ejerciciosopcionales suman 4 puntos en la calificación de las memorias.

13.4. Registros internos de la CPU

- La arquitectura amd64 dispone de:
 - 16 registros de propósito general (RPG) de 64 bits cada uno: rax,rbx,rcx,rdx,rsi,rdi,rsp, etc
 - 1 registros de estado de 64 bits: rflags
- El acceso a los registros de propósito general puede ser *parcial*:
 - Registro RAX: es un registro de 64 bits
 - Registro EAX: son los 32 bits de menor peso de RAX
 - Registro AX: son los 16 bits de menor peso de RAX
 - Registro AL: son los 8 bits de menor peso de RAX
 - Registro AH: es el byte con los bits de las posiciones 8:15 de RAX
- En las "[Hojas de Referencia Rápida](#)" están representados todos los nombres de los diferentes grupos de bits de cada registro de propósito general.

13.5. Tamaño de los datos y variables

13.5.1. Algoritmo

- La sección de instrucciones comprende un algoritmo que inicializa dos punteros.

13.5.2. Edición del Módulo fuente: `datos_size.s`

- Descargar el módulo fuente "datos_size.s" de miaulario y añadir los comentarios apropiados.

```
### Programa: datos_size.s
### Descripción: declarar y acceder a distintos tamaños de operandos
### Compilación: gcc -m32 -g -o datos_size datos_size.s
```

```
## MACROS
.equ SYS_EXIT, 1
.equ SUCCESS, 0
```

VARIABLES LOCALES

```
.data
```

```
da1: .byte 0x0A  
da2: .2byte 0xA0B  
da4: .4byte 0xA0B0C0D  
men1: .ascii "hola"  
lista: .int 1,2,3,4,5
```

INSTRUCCIONES

```
.global _start
```

```
.text
```

```
_start:
```

```
    mov $da4,%eax
```

```
    lea da4,%ebx
```

```
    mov (%eax),%ecx
```

```
    mov (%ebx),%edx
```

```
salida:
```

```
    mov $SYS_EXIT, %eax
```

```
    mov $SUCCESS, %ebx
```

```
    int $0x80
```

```
.end
```

13.5.3. Compilación

- Seguir los pasos del proceso de compilación común a todas las sesiones.
 - gcc -nostartfiles -m32 -g -o datos_size datos_size.s

13.5.4. Ejecución

- ./datos_size
- echo \$?

13.5.5. Análisis del módulo fuente

- Leer en las hojas de referencia rápida el Programa Ejemplo Minimalista

Estructura en secciones: ensamblaje

- La estructura del programa esta formada por los siguientes elementos:
 - Cabecera
 - Definición de Macros
 - Sección de Datos
 - Sección de Instrucciones

Definición de Macros

- Macro:
 - La construcción macro se utiliza en el programa fuente para sustituir datos utilizados en el programa

fuente por símbolos de texto que faciliten la lectura del código fuente.

- Para ello empleamos la directiva "EQU" cuya sintaxis es: **.EQU SÍMBOLO, dato**
- El preprocesador en la primera fase de la compilación sustituirá el texto SIMBOLO que aparece a lo largo de la sección de datos e instrucciones por el dato asociado.
- Macros empleadas
 - SYS_EXIT : código de la llamada al sistema para finalizar el programa y devolver el control al Sistema Operativo. En la arquitectura i386 su valor es 1.
 - SUCCESS : código empleado por los programas para indicar que su ejecución se ha realizado con normalidad. Su valor es 0.

Sección de Datos

- Interpretar las etiquetas y directivas de reserva de memoria e inicialización para los datos utilizando la [tabla de directivas](#): identificar las variables ordinarias, strings y arrays.



Si un objeto de memoria es inicializado con un número entero que es representado con menos dígitos que el tamaño del objeto, los dígitos de mayor peso tendrán de valor cero. Por ejemplo: **.4byte 0xFF** equivale a **.4byte 0x000000FF**

Sección de Instrucciones

- Determinar la instrucción de entrada al programa.
- Determinar el bloque de salida del programa.

13.5.6. GDB: Observaciones

- El depurador al visualizar el contenido de los registros:
 - únicamente visualiza el número de bytes del tamaño de los operandos..aunque los registros "r-x" son de 64 bits (rax,rbx,etc..)
 - con números enteros con signo, no visualiza los ceros de mayor peso, es decir, ni el signo ni la extensión de signo de los números positivos.

13.5.7. GDB:Ejecución paso a paso

Inicialización

- Compilar el programa con la opción de generación de la tabla de símbolos requerida por el depurador y generar el módulo binario ejecutable:
 - **gcc -nostartfiles -m32 -g -o datos_size datos_size.s**
- Abrir el depurador GDB, cargar el módulo binario ejecutable y comprobar que se carga la tabla de símbolos junto al módulo binario ejecutable.
 - **gdb**
 - **file datos_size**
 - **info sources**
- Abrir la ventana para el módulo fuente
 - **layout src ó Control-x Control-a**
- Configurar el fichero para el logging histórico de los comandos.
 - **set trace-commands on**

- `set logging file datos_size_gdb_asm.txt`
- `set logging on`
- `shell ls -l datos_size_gdb_asm.txt`
- Activar un punto de ruptura en la instrucción de entrada al programa.
 - `b _start`
- Ejecutar el programa deteniéndolo en la primera instrucción del programa.
 - `run`

Comandos y operadores: x, p, disas, casting, &, *, @

- comando eXaminar **x**: vuelca el contenido de una **dirección** de memoria
 - `x /nvt address`
 - formato `/nvt` : "t" es el *tamaño* de la variable en memoria , "v" la codificación del *valor* del contenido de memoria a visualizar y "n" el *número* de veces que hay que volcar secuencialmente grupos de bytes en memoria de tamaño "t" comenzando en la dirección `address`
 - `help x` : formatos d (decimal) ,x (hexadecimal),t (binario) ,o (octal) ,c (character) ,a (address),i (instruction),etc
 - ejemplos:
 - `x /1d4 address` (ejecutar 1 vez el comando examinar en código decimal volcando un objeto de 4bytes ubicado en la dirección `address`)
 - `x /2t4 address` (ejecutar 2 veces el comando examinar en código binario: volcando la primera vez un objeto de 4 bytes ubicado en la dirección `address` y volcando la segunda vez un objeto de 4 bytes ubicado en la dirección `address+4`)
 - `x /100x1 address`: vuelca 100 datos de 1 byte en código hexadecimal a partir de la dirección `address`.
 - La sintaxis del argumento del comando examinar es la misma que en lenguaje de **programación de C**.
 - `work language: show language` → indica que el lenguaje de las expresiones GDB son ASM (pej `$eax`) pero en cambio también admite el lenguaje C (&variable)
 - `work languages supported: set language`
- operador **&** : se utiliza como prefijo de una etiqueta para evaluar la dirección de memoria a la que hace referencia una etiqueta
- operador ***** : se utiliza para evaluar el contenido de una posición de memoria mediante la indirección de un puntero
- operación de **casting**:
 - [Apéndice Programación Lenguaje C](#)
 - El casting consiste en definir o redifinir el tipo de variable. Se utiliza como prefijo de la variable a redefinir y va entre paréntesis.
 - la etiqueta "lista" está definida en la sección de datos mediante la directiva ".int". Esta directiva reserva memoria para inicializar los datos a partir de la dirección &lista pero NO es una declaración de tipo por lo que el depurador NO tiene información sobre el tipo de elementos del array lista y por ello es necesario realizar declaraciones en modo casting.
 - Ej. (`char *`): el tipo `char *` es un puntero a un entero de 1 byte.
- comando Print **p**: Evalua el argumento del comando y el valor resultante lo imprime en pantalla
 - La sintaxis del argumento del comando examinar es la misma que en lenguaje de programación de C.

- Ej. p /a &lista : evalua &lista cuyo valor resultante se imprime con formato tipo "a" (address)
- formatos de impresión: los mismos que eXaminar: **help x**
- operador @: **direccion@n**: array artificial: evalua la expresión "direccion" (a la izda de @) y debe ser una dirección de memoria. Crea una array artificial de longitud "n" (el valor del parámetro a la derecha del operador @) bytes.
- comando **disas** : desensambla el código binario traduciéndolo a código ensamblador.

Análisis

- Análisis del contenido de la memoria principal mediante el depurador GDB:

```
//Alineamiento interno de los bytes de un dato
x /tb &da1
x /xh &da2
x /xw &da4
x /5xb &da4 -> Alineamiento little endian

//Alineamiento de los bytes de un string
x /5cb &men1      -> Alineamiento en secuencia
x /5xb &men1

//Volcado de un string
p /s (char *)&men1 -> imprime una cadena de caracteres desde la primera dirección
hasta encontrar el carácter NULL (0x00).

//Volcado de un array
x /5xw &lista          -> contenido de 5 elementos de lista
p /a &lista            -> dirección del array lista
p /a &lista+1          -> el depurador informa que es necesario realizar algún tipo de
                           casting (declaración dinámica)
p /a (void *)&lista+1   -> se incrementa en 1 byte
p /a (int *)&lista+1    -> escalado: se incrementa en 1*4 bytes apuntando al
                           segundo elemento del array
p lista                -> el depurador informa que es necesario realizar un
                           casting
p (int)lista           -> primer elemento del array
p (int *)&lista         -> dirección del array lista
p *((int *)&lista+1)    -> segundo elemento de lista
x /dw (int *)&lista+1   -> segundo elemento de lista
p (int [5])lista        -> contenido de cinco elementos de lista
p *(int *)&lista@5       -> array artificial de 5 elementos de tipo int a partir de
                           la dirección &lista.

//volcado de una instrucción
p &_start
x /i &_start           -> desensambla: convierte el código máquina en código
                           ensamblador.

//Desensamblar: Conversión del código máquina en ensamblador
disas /r _start
```

```

layout split

//Análisis de los punteros
b salida
c
p /a &da4
x /x4 &da4
p /x (int)da4
p /x $eax
p /x *(int *)$eax

```

13.6. Tamaño de los Operandos

13.6.1. Edición del Módulo fuente: datos_sufijos.s

- Descargar el módulo fuente "datos_sufijos.s" de miaulario y añadir los comentarios apropiados.

```

### Programa: datos_sufijos.s
### Descripción: utilizar distintos sufijos para los mnemónicos indicado distintos
tamaños de operandos
### Compilación: gcc -nostartfiles -m32 -g -o datos_sufijos datos_sufijos.s

## MACROS
.equ SYS_EXIT, 1
.equ SUCCESS, 0

## VARIABLES LOCALES
.data

da1: .byte 0xA
da2: .2byte 0xA0B
da4: .4byte 0xA0B0C0D
saludo: .ascii "hola"
lista: .int 1,2,3,4,5

## INSTRUCCIONES
.global _start
.text
_start:

## Reset de Registros
xor %eax,%eax
xor %ebx,%ebx
xor %ecx,%ecx
xor %edx,%edx

## Carga de datos
## mov da1,da4      ERROR: por referenciar las dos direcciones efectivas de los
dos operandos a la memoria principal
mov da4,%eax

```

```

movl da4,%ebx
movw da4,%cx
movb da4,%dl

## Reset de Registros
xor %eax,%eax
xor %ebx,%ebx
xor %ecx,%ecx
xor %edx,%edx

## Carga de datos
mov da4,%al      #aplica el tamaño de AL
## movw da4,%al  ERROR: incoherencia entre -w y AL
movb da4,%ebx    #AVISO, NO error: incoherencia entre el registro BL y el
sufijo

mov da1,%ecx
mov da4,%dx

## Reset de Registros
xor %eax,%eax
xor %ebx,%ebx
xor %ecx,%ecx
xor %edx,%edx

## Carga de datos

mov da1,%al

## inc da1      ERROR: por ser la dirección efectiva del operando una
referencia a la memoria principal no restringe el tamaño del operando. Al no
especificar tampoco sufijo el ensamblador no reconoce el tamaño del operando.
incb da1
incw da2
incl da4

## salida
mov $SYS_EXIT, %eax
mov $SUCCESS, %ebx
int $0x80

.end

```

13.6.2. Compilación

- Seguir los pasos de la compilación de un módulo en lenguaje ensamblador.
 - `gcc -nostartfiles -m32 -g -o datos_sufijos datos_sufijos.s`
 - WARNING: **Aviso:** empleando `%b` en lugar de `%ebx` debido a la utilización de `b` como sufijo

- Es un aviso de la sintaxis de la instrucción `movb da4,%ebx`, NO es un error.

13.6.3. Ejecución

- `./datos_sufijos`
- `echo $?`

13.6.4. Análisis del módulo fuente asm

- Sufijos de los mnemónicos indicando distintos tamaños de los operandos: b,w,l
 - `movw da4,%cx` : el sufijo "w" de 2 bytes y el registro destino CX de dos bytes.
 - `movw da4,%al` : el sufijo "w" impone una transferencia de 2 bytes a un registro destino AL de 1 byte → error en el ensamblaje.
 - `movb da4,%ebx` : el sufijo "b" no es coherente con el registro destino EBX de 4 bytes y el ensamblaje se produce con BL.
- Sin sufijo:
 - `xor %eax,%eax` : operandos fuente y destino EAX de 4 bytes
 - `mov da4,%al` : el registro destino AL limita la transferencia a 1 byte y no hay contradicción con el sufijo ya que éste no existe.
 - `mov da1,%ecx` : de los dos operandos, registro y memoria, es el registro quien prioriza el tamaño de la transferencia.
 - `inc da1` : Al ser la dirección efectiva del operando una referencia a la memoria principal no restringe el tamaño del operando. Al no especificar tampoco un sufijo el ensamblador no reconoce el tamaño del operando → error en el ensamblaje

13.6.5. Deducción del tamaño del operando en una instrucción asm

1. Diferencia entre la referencia a un operando en memoria o registro
 - a. Un operando referenciado mediante una dirección de memoria no tiene un tamaño específico para el assembler.
 - b. En cambio el nombre de un registro si es asociado a un tamaño de operando por el assembler.
2. En una instrucción con un único operando en memoria el tamaño es deducido por el assembler gracias al sufijo del mnemónico, por lo tanto en este caso si el mnemónico no tiene sufijo el assembler no traducirá la instrucción.
3. En una instrucción con dos operandos, uno en memoria y otro en un registro, es el operando en el registro o el sufijo quienes especifican el tamaño de los dos operandos fuente y destino:
 - a. Si el mnemónico tiene sufijo, es dicho sufijo quien especifica el tamaño de los operandos fuente y destino.
 - b. Si el mnemónico no tiene sufijo, es el tamaño del registro quien especifica el tamaño de los operandos fuente y destino.
4. Casos de error
 - a. En el caso de que el mnemónico tenga un sufijo mayor que el tamaño del registro destino.
 - b. En el caso de que el mnemónico no tenga sufijo y el tamaño del registro fuente sea mayor que el registro destino.

13.6.6. GDB:Ejecución paso a paso

Inicialización

- Compilar el programa con la opción de generación de la tabla de símbolos requerida por el depurador y generar el módulo binario ejecutable:
 - `gcc -nostartfiles -m32 -g -o datos_sufijo datos_sufijo.s` donde modulo_fuente se sustituye por el nombre del archivo que se desea compilar.
- Abrir el depurador GDB, cargar el módulo binario ejecutable y comprobar que se carga la tabla de símbolos junto al módulo binario ejecutable.
 - `gdb`
 - `file modulo_ejecutable`
 - `info sources`
- Abrir la ventana para el módulo fuente
 - `layout src` ó `Control-x Control-a`
- Configurar el fichero para el logging histórico de los comandos.
 - `set trace-commands on`
 - `set logging file datos_sufijo_gdb_asm.txt`
 - `set logging on`
 - `shell ls -l datos_sufijo_gdb_asm.txt`
- Activar un punto de ruptura en la instrucción de entrada al programa.
 - `b _start`
- Ejecutar el programa deteniéndolo en la primera instrucción del programa.
 - `run`
- abrir la ventana de visualización de los registros
 - `layout regs`
- Análisis del contenido de la memoria principal mediante el depurador GDB.
 - Ejecutar el programa paso a paso analizando el resultado de la ejecución de cada instrucción
 - `n`
 - RET, RET, RET,

13.7. Modos de Direcciónamiento

13.7.1. Edición del Módulo fuente: `datos_direccionamiento.s`

- Descargar el módulo fuente "datos_direccionamiento.s" de miaulario y añadir los comentarios apropiados.

```
### Program:      datos_direccionamiento.s
### Descripción: Emplear estructuras de datos con diferentes direccionamientos
### Compilación: gcc -m32 -g -o datos_direccionamiento datos_direccionamiento.s
###           sin la opción startfiles al utilizar el punto de entrada referenciado con
###           la etiqueta "main"
```

```
## MACROS
.equ SYS_EXIT, 1
.equ SUCCESS, 0
```

```

## VARIABLES LOCALES
.data

.align 4          # Alineamiento con direcciones de MP múltiplos de
4
da2: .2byte 0xA0B,0b000111101011100,-21,0xFFFF # Array da2 de elementos de 2
bytes
.align 4
lista: .word 1,2,3,4,5 # Array lista de elementos de 2 bytes
.align 8
buffer: .space 100     # Array buffer de 100 bytes
.align 2
saludo:
.string "Hola"      # Array saludo de elementos de 1 byte por ser caracteres

## INSTRUCCIONES
.global main
.text
main:

## RESET

xor %eax,%eax
xor %ebx,%ebx
xor %ecx,%ecx
xor %edx,%edx
xor %esi,%esi
xor %edi,%edi

## ALGORITMO sum1toN

## Direccionamiento inmediato
mov $4,%si
## Direccionamiento indexado
bucle: add lista(%esi,2),%di
## Direccionamiento a registro
dec %si
## Direccionamiento relativo al PC
jns bucle

## EJERCICIOS SOBRE DIRECCIONAMIENTO

## Direccionamiento indirecto
lea buffer,%eax    #inicializo el puntero EAX
## mov da2,(%eax) ERROR: la dirección efectiva de los dos operandos hacen
referencia a la memoria principal
mov da2,%bx
mov %bx, (%eax)
## Direccionamiento directo

```

```

incw da2
## Direccionamiento indexado
lea da2,%ebx
## inc 2(%ebx) ERROR: dirección efectiva a memoria y no hay sufijo
incw 2(%ebx)

mov $3,%esi
mov da2(,%esi,2),%ebx

## SALIDA

mov $SYS_EXIT, %eax
mov $SUCCESS, %ebx
int $0x80

.end

```

13.7.2. Compilación

- Seguir los pasos de la compilación de un módulo en lenguaje ensamblador.
 - El punto de entrada no es "_start".
 - `gcc -m32 -g -o datos_direccionamiento datos_direccionamiento.s`

13.7.3. Ejecución

- `./datos_direccionamiento`
- `echo $?`

13.7.4. Análisis del módulo fuente asm

- Alineación de datos mediante la directiva `.align n` asigna una dirección de memoria múltiplo de n al siguiente dato declarado.



NO está permitido que en el caso de una instrucción con dos operandos, ambos estén en la memoria principal. Uno o los dos operandos han de estar en los registros de propósito general.

13.7.5. GDB: Ejecución paso a paso

inicialización

- Compilar el programa con la opción de generación de la tabla de símbolos requerida por el depurador y generar el módulo binario ejecutable:
 - `gcc -m32 -g -o datos_direccionamiento datos_direccionamiento.s` donde modulo_fuente se sustituye por el nombre del archivo que se desea compilar.
- Abrir el depurador GDB, cargar el módulo binario ejecutable y comprobar que se carga la tabla de símbolos junto al módulo binario ejecutable.
 - `gdb`
 - `file datos_direccionamiento`

- `info sources`
- Abrir la ventana para el módulo fuente
 - `layout src` ó `Control-x Control-a`
- Configurar el fichero para el logging histórico de los comandos.
 - `set trace-commands on`
 - `set logging file datos_direccionamiento_gdb_asm.txt`
 - `set logging on`
 - `shell ls -l datos_direccionamiento_gdb_asm.txt`
- Activar un punto de ruptura en la instrucción de entrada al programa.
 - `b _start`
- Ejecutar el programa deteniéndolo en la primera instrucción del programa.
 - `run`

Análisis

- Array *da2*
 - Imprimir la dirección de memoria del array *da2* y el contenido del primer elemento: `x /xh &da2`
 - 4 elementos de 2bytes del array *da2*: `x /4xh &da2`
 - `p /x (short[4])da2`
- Array *lista*
 - `ptype lista`
 - `p (short[5])lista`
- Array *buffer*
 - `ptype buffer`
 - Imprimir la dirección de memoria del array *buffer* y comprobar su alineamiento: `p &buffer`
- String
 - `ptype saludo`: no debug info → no admite referencia elemento array expresión *saludo[n]*
 - `p /c (char[5])saludo`: casting array
 - `x /5c (char *)&saludo`: casting puntero
 - `p /c *(char *)&saludo`: casting puntero e indirección
 - `p /s (char *)&saludo`: casting puntero y formato string

Chapter 14. Operaciones Aritméticas y Lógicas

14.1. Introducción

14.1.1. Objetivos

- Programación:
 - Realizar operaciones aritméticas (suma,resta,multiplicación y división) con números enteros.
- Análisis:
 - Comprobar cómo afectan las operaciones lógicas y aritméticas a los flags del registro de estado EFLAGS
 - Analizar el contenido de la memoria como números con signo, caracteres, arrays y strings: tipos y tamaños de los operandos numéricos y de los operandos alfanuméricos
 - Realizar la operación de desensamblaje para comprobar el lenguaje máquina del módulo ejecutable cargado en la memoria principal

14.1.2. Conceptos de Arquitectura

- La Unidad Aritmético-Lógica ALU sólo opera con números enteros almacenados en los registros de propósito general. Para operar con números reales es necesaria la unidad Float Process Unit FPU con los operandos almacenados en los registros específicos para números en coma flotante.

14.1.3. Módulos fuente

- [op_arit_log.s](#)

14.1.4. Requisitos

- Teoría: representación de datos, operaciones aritméticas y lógicas, formato de instrucciones y repertorio ISA de la arquitectura X86.
- Prácticas previas:
 - Introducción a la programación en lenguaje ensamblador AT&T x86-32
 - Representación de los Datos

14.2. LEEME

- Lectura del guión de prácticas y del capítulo 3 del Libro Programming from the Ground-Up.
- [Apuntes y Libro de Texto](#)
- [Documentación Memoria](#): Contenido y Formato de la Memoria
- [Evaluación](#): sistema de evaluación
- [Plataforma de Desarrollo](#) : configuración de la computadora personal
- [Programación](#) : metodología

14.3. Cuestiones

- "Autoevaluación de Prácticas" opcional: [Prácticas: Cuestionario](#)

14.4. Registros internos de la CPU

- La arquitectura amd64 dispone de:
 - 16 registros de propósito general (RPG) de 64 bits cada uno: rax,rbx,rcx,rdx,rsi,rdi,rsp, etc
 - 1 registros de estado de 64 bits: rflags
- El acceso a los registros de propósito general puede ser *parcial*:
 - Registro RAX: es un registro de 64 bits
 - Registro EAX: son los 32 bits de menor peso de RAX
 - Registro AX: son los 16 bits de menor peso de RAX
 - Registro AL: son los 8 bits de menor peso de RAX
 - Registro AH: es el byte con los bits de las posiciones 8:15 de RAX
- En las "[Hojas de Referencia Rápida](#)" están representados todos los nombres de los diferentes grupos de bits de cada registro de propósito general.

14.5. Operaciones Aritméticas y Lógicas con Números Enteros con Signo

14.5.1. Edición del Módulo fuente: op_arit_log.s

- Descargar el módulo fuente "op_arit_log.s" de miaulario y añadir los comentarios apropiados.

```
### Programa:      op_arit_log.s
### Descripción: Emplear estructuras de datos con diferentes operaciones lógicas y
aritméticas.
### Compilación: gcc -m32 -g -o op_arit_log op_arit_log.s

## MACROS
.equ SYS_EXIT, 1
.equ SUCCESS, 0
.equ N,      5

## VARIABLES LOCALES

## INSTRUCCIONES
.global main
.text
main:

## RESET

xor %eax,%eax
xor %ebx,%ebx
xor %ecx,%ecx
xor %edx,%edx
xor %esi,%esi
xor %edi,%edi
```

OPERACIONES ARITMETICAS con NUMEROS ENTEROS

```

## add: suma
mov $5,%eax
mov $10,%ebx
add %ebx,%eax

## sub: resta ①
mov $5,%eax
mov $10,%ebx
sub %ebx,%eax

## imul: multiplicación entera "con signo": AX<- BL*AL ②
    movb $-3,%bl
    movb $5,%al
    imulb %bl

## idiv: división "con signo" . (AL=Cociente, AH=Resto) <- AX/(byte en
registro o memoria) ③
    movw $5,%ax      #dividendo
    movb $3,%bl      #divisor
    idivb %bl       # 5/3 = 1*3 + 2

## complemento a 2: equivalente a cambiar de signo negación
negb %bl

## Expresión N*(N+1)/2
movw $N,%bx
movw $(N+1),%ax
imulw %bx        #imulw Op ; Op=word ; DX:AX<- AX*Op
movw $2,%bx
## El resultado queda en AX y el resto DX=0 ④
idivw %bx        #idivw Op ; Op=word ; AX<-(DX:AX)/Op ; DX:=Resto

```

OPERACIONES LOGICAS

```

mov $0xFFFF1F, %eax
    mov $0x0000F1, %ebx
not %eax      # inversión
and %ebx,%eax  # producto lógico
or %ebx,%eax   # suma lógica

## Complemento a 2 mediante operación lógica not() +1
mov %ebx,%eax
not %eax
inc %eax

```

```

## Desplazamiento de bits ⑤
    shr $4,%eax      #desplazamiento lógico: bits a introducir -> 0..
    sar $4,%eax      #desplazamiento aritmético: bits a introducir -> extensión del
signo

## SALIDA

    mov $SYS_EXIT, %eax
    mov $SUCCESS,  %ebx
    int $0x80

.end

```

① Instrucciones referenciadas en las cuestiones de autoevaluación

② " " "

③ " " "

④ " " "

⑤ " " "

14.5.2. Compilación

- Seguir los pasos de la [compilación](#) de un módulo en lenguaje ensamblador.
 - El punto de entrada no es "_start".
 - `gcc -m32 -g -o op_arit_log op_arit_log.s`

14.5.3. Ejecución

- `./op_arit_log`
- `echo $?`

14.5.4. Análisis del módulo fuente

- Para la interpretación de las instrucciones add, sub, imul (integer multiplication), idiv (integer division) , neg, not, and, or, xor, shr, sar, consultar la tabla de operaciones de las [hojas de referencia rápida](#)
- Las operaciones únicamente procesan el número de bits que indica el sufijo del mnemónico...aunque los registros "r-x" son de 64 bits.

14.5.5. Ejecución paso a paso

Observaciones



El depurador al visualizar el contenido de los registros: Unicamente visualiza el número de bytes del tamaño de los operandos..aunque los registros "r-x" son de 64 bits. Con números enteros con signo no visualiza los ceros de mayor peso, es decir, ni el signo ni la extensión de signo de los números positivos.

Operaciones

- Compilar el programa con la opción de generación de la tabla de símbolos requerida por el depurador y

generar el módulo binario ejecutable:

- `gcc -m32 -g -o op_arit_log op_arit_log.s` donde modulo_fuente se sustituye por el nombre del archivo que se desea compilar.
- Abrir el depurador GDB, cargar el módulo binario ejecutable y comprobar que se carga la tabla de símbolos junto al módulo binario ejecutable.
 - `gdb`
 - `file op_arit_log`
 - `info sources`
- Configurar el fichero para el logging histórico de los comandos.
 - `set trace-commands on`
 - `set logging file op_arit_log_gdb_asm.txt`
 - `set logging on`
 - `shell ls -l op_arit_log_gdb_asm.txt`
- Activar un punto de ruptura en la instrucción de entrada al programa.
 - `b _start`
- Ejecutar el programa deteniéndolo en la primera instrucción del programa.
 - `run`

Operaciones aritméticas

- Comprobar los resultados de las operaciones aritméticas de suma, resta, multiplicación, división y negación de números enteros con signo

Operaciones lógicas

- Comprobar los resultados de las operaciones lógicas bitwise de negación, multiplicación, suma, or-exclusiva y desplazamiento.

Chapter 15. Instrucciones de Saltos Condicionales

15.1. Introducción

15.1.1. Objetivos

- Manejo del registro de flags, instrucciones de comparación y saltos condicionales para su aplicación en sentencias de lenguajes de alto nivel tipo if, for, while, switch-case.
- Depurador GDB
 - Uso del comando `watch`

15.1.2. Requisitos

- Teoría: representación de datos, operaciones aritméticas y lógicas, formato de instrucciones y repertorio ISA de la arquitectura X86.
- Práctica anterior: Introducción a la programación en lenguaje ensamblador AT&T x86-32
- Conceptos del lenguaje de programación C: if, for, while, switch-case

15.2. LEEME

- Lectura del guión de prácticas y del capítulo 3 del Libro Programming from the Ground-Up.
- [Apuntes y Libro de Texto](#)
- [Documentación Memoria](#): Contenido y Formato de la Memoria
- [Evaluación](#): sistema de evaluación
- [Plataforma de Desarrollo](#) : configuración de la computadora personal
- [Programación](#) : metodología

15.3. Cuestiones

- "Autoevaluación de Prácticas" opcional: [Prácticas: Cuestionario](#)

15.4. Saltos Condicionales

15.4.1. Algoritmo

- No se desarrolla ningún algoritmo. Son instrucciones para la práctica de los saltos condicionales.

15.4.2. Edición del Módulo fuente: `saltos.s`

- Descargar el módulo fuente "saltos.s" de miaulario y añadir los comentarios apropiados.

```
/*
Programa:      datos_saltos.s
Descripción: Emplear estructuras de datos con diferentes direccionamientos
### Compilación: gcc -m32 -g -o datos_saltos datos_saltos.s
```

```
*/\n\n## MACROS\n.equ SYS_EXIT, 1\n.equ SUCCESS, 0\n\n## VARIABLES LOCALES\n.data\n\n## INSTRUCCIONES\n.global main\n.text\nmain:\n\n## RESET\n\n    xor %eax,%eax\n    xor %ebx,%ebx\n    xor %ecx,%ecx\n    xor %edx,%edx\n    xor %esi,%esi\n    xor %edi,%edi\n\n## SALTOS INCONDICIONALES\n\n## Direccionamiento relativo\njmp salto1      #salto relativo al contador de programa pc -> eip\nxor %esi,%esi\n\n## FLAGS DEL REGISTRO DE BANDERINES EFLAGS\n/*\n   los flags se activan al realizar operaciones aritméticas, lógicas, etc\n   dependiendo del resultado de dicha operación\nCF: El resultado de la operación tiene llevada del bit MSB del destino\nOF: El resultado de la operación con signo se desborda, su tamaño supera el\npermido.\nZF: el resultado de la operación tiene valor cero\nSF: el resultado de la operación tiene valor negativo\nPF: el resultado de la operación tiene el byte LSB con un número par de bits\n */\nsalto1:\n    xor %eax,%eax          # resultado cero -> activa ZF y PF pero desactiva\n    CF,OF,SF\n    inc %eax              # desactiva ZF y PF\n    neg %eax              # activa SF,PF y CF : realiza la resta de la definición de\n    complemento a 2 :(0-N)\n    shr $1,%eax           # SHift Right : desplazamiento lógico: desplaza n bits el\n    operando destino.\n    /* Salen bits por la dcha y entran ceros por la izda.\n       El último bit salido queda en CF.
```

SF=0 ya que ha entrado un cero en el MSB
MANUAL INTEL: <http://www.cs.nyu.edu/~mwalfish/classes/ut/s13-cs439/ref/i386/SAL.htm>

For SHR, OF is set to the high-order bit of the original operand.
OF=MSB=1
The OF flag is affected only on 1-bit shifts.
Equivale a dividir 2^n si desplaza a la dcha y a multiplicar 2^n hacia la izda (possible overflow).
*/

```

shl $1,%eax
clc          # clear CF -> CF=0
xor %eax,%eax      # resultado cero -> activa ZF y PF pero desactiva
CF,OF,SF
movw $0xFFFF,%ax      # MOV NO afecta a ningún flag
addw $0xFFFF,%ax      # activa SF y CF pero no OF
clc
movw $0x7FFF,%ax
addw $1,%ax      # activa OF pero no CF, OF avisa del error en la suma y se puede
ver que SF se ha activado

```

INSTRUCCIONES COMPARATIVAS: TEST,CMP

Comprobar si el bit de la posicion 5 es cero con la mascara 0x0010 que aisla dicha posicion
test realiza la operación AND afectando a los flags de EFLAGS pero no guarda el resultado en el operando destino

```

movw $0xABFF, %ax
movw $0x0BCF, %bx
test $0x0010, %ax    # AX^0x0010=0x0010=positivo -> SF=0, low byte=0x10 impar ->
PF=0,
        # El manual dice -> The OF and CF flags are cleared
test $0xFFFF, %ax    # SF=1 porque AX^0xFFFF=AX= negativo, low byte=AL= par
-> PF=1
test $0b0000000000001000, %bx    # SF=0 porque AX^0x0010=positivo ,ZF=1 porque
BX[5] es cero, PF=0

```

Comprobar si el valor de una variable es mayor, menor o igual al valor 0x00FF
cmp realiza la operacion SUB afectando a los flags de EFLAGS pero no guarda el resultado en el operando destino

SUB: It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags

to indicate an overflow in the signed or unsigned result, respectively

```

movw $0x01FF, %ax
movw $0x0001, %bx
movw $0x00FF, %cx
cmp $0x00FF, %ax    # AX-0x00FF=0x0100 > 0 -> ZF=0 y SF=0, low byte=00 ->
PF=1
cmp $0x00FF, %bx    # BX-0x00FF=0x0001+0xFF01=0xFF02 < 0 -> SF=1, 0x02 impar
-> PF=0,

```

```

# unsigned overflow -> CF=1, signed not overflow OF=0
cmp $0x00FF, %cx # CX-0x00FF=0 -> ZF=1, SF=0, 0xFF par PF=1, CF=0, OF=0

## SALTOS CONDICIONALES

movw $0x01FF, %ax
movw $0x0001, %bx
movw $0x00FF, %cx
cmp $0x00FF, %ax # AX-0x00FF=0x0100 > 0, luego ZF=0 y SF=0, 0x00 para ->
PF=1
jg salto4 # great jump -> resta de numeros con signo -> SF=0 y salta
nop
salto4: cmp $0x00FF, %bx # BX-0x00FF=0x0001+0xFF01=0xFF02 < 0, luego ZF=0 ,
SF=1,
# unsigned over CF=1 y not signed over OF=0
jl salto5 # less jump -> resta de numeros con signo -> SF=1 y salta
nop
salto5: movw $0x8000, %ax # 0x8000 vale -32768 con signo y 32768 sin signo
cmp $0x0001, %ax # Con signo ->0x8000 - 0x1 = 0x8000+0xFFFF=0x7FFF >0 ->
SF=0,
# OF=1 ya que la suma de dos negativos ha dado positivo
# CF=0 ya que en binario puro 0x01FF-
0x00001=0x01FE, no overflow
# 0xFF es par -> PF=1

ja salto6 # above jump -> resta de números sin signo -> 32768-1>0
nop
salto6: cmp $0x00FF, %cx # CX-0x00FF = 0, luego ZF=1 y SF=0
je salto7 # equal jump
nop

## SALIDA

salto7: mov $SYS_EXIT, %eax
        mov $SUCCESS, %ebx
        int $0x80

.end

```

15.4.3. Compilación

- Seguir los pasos del proceso de [compilación](#) común a todas las sesiones.
 - `gcc -m32 -g -o saltos saltos.s`

15.4.4. Ejecución

- `./saltos`
- `echo $?`

15.4.5. Análisis del módulo fuente

- Leer en las hojas de referencia rápida el Programa Ejemplo Minimalista

Estructura

- La estructura del programa esta formada por los siguientes elementos:
 - Cabecera
 - Definición de Macros
 - Sección de Datos
 - Sección de Instrucciones

15.4.6. Ejecución paso a paso

Operaciones Iniciales

- Compilar el programa con la opción de generación de la tabla de símbolos requerida por el depurador y generar el módulo binario ejecutable:
 - `gcc -m32 -g -o saltos saltos.s`
- Abrir el depurador GDB, cargar el módulo binario ejecutable y comprobar que se carga la tabla de símbolos junto al módulo binario ejecutable.
 - `gdb`
 - `file saltos`
 - `info sources`
- Configurar el fichero para el logging histórico de los comandos.
 - `set trace-commands on`
 - `set logging file saltos_gdb_asm.txt`
 - `set logging on`
 - `shell ls -l saltos_gdb_asm.txt`
- Activar un punto de ruptura en la instrucción de entrada al programa.
 - `b _start`
- Ejecutar el programa deteniéndolo en la primera instrucción del programa.
 - `run`

Registro EFLAGS

- `p $eflags` → imprime los nombres de los flags activos
- comprobar el estado de los banderines con las operaciones lógicas, aritméticas y las instrucciones TEST y CMP.
- Para el análisis de los banderines en modo paso a paso utilizar el comando `watch` del depurador GDB

```
(gdb) watch $eflags : interrumpe la ejecución y visualiza el contenido del registro  
EFLAGS cada vez que cambia su valor.  
(gdb) info watch   : visualiza los watchs definidos  
(gdb) delete breakpoints : elimina los breaks, watchs, etc
```

Saltos

- Comprobar la ejecución o no del salto con el estado e interpretación de los banderines del registro EFLAGS. Es un ejercicio de interpretación de FLAGS → Cuando se ejecuta la instrucción de salto, la CPU tiene que tomar la decisión de saltar o no interpretando los FLAGS. ¿ Sabríamos RELACIONAR la condición de **ja** (salto si ABOVE) leyendo los FLAGS y sin leer los valores de los operandos que se comparan en la condición ABOVE?

15.5. Mnemónicos Utilizados

- Ver capítulo "[Programación ensamblador : Mnemónicos Básicos \(Explicados\)](#)"

Chapter 16. LLamadas al Sistema Operativo (Kernel)

16.1. Introducción

16.1.1. Qué son las llamadas al sistema

- El HW está protegido por el Kernel del SO y por lo tanto el programador de ensamblador accede al HW indirectamente a través de las "LLamadas al Sistema" solicitando operaciones de entrada/salida al Sistema Operativo. Por lo tanto si queremos acceder al teclado y al monitor será necesario realizar llamadas al kernel.
- Definición de la interfaz entre el programador y el kernel del SO: *System V Application Binary Interface: SysV-ABI*
 - El lenguaje ensamblador sigue la norma ABI para el lenguaje C.
- En este guión se trabajan las llamadas al sistema de la arquitectura i386.
- LLamadas al sistema desde el código ensamblador:
 - directamente con la instrucción `int $0x80`
 - indirectamente a través de las funciones de la librería standard `/libc` con la instrucción `call`
 - En código ensamblador es necesario pasar los argumentos previamente a la ejecución de la llamada `call`

16.1.2. Manuales de las llamadas al sistema

- Listado con los nombres de las llamadas a al sistema: `man syscalls`
 - LLamada al sistema `exit`: `man 3 exit`
 - describe la función de llamada al sistema
 - especifica el nombre de la cabecera de la librería necesaria para compilar en lenguaje C.
 - especifica los parámetros que necesita la función y el orden en que son transferidos.
 - LLamada al sistema `write`: `man 2 write`

16.1.3. Códigos de las llamadas

- Códigos de las llamadas al sistema en la arquitectura x86-32:
 - `/usr/include/asm/unistd_32.h`
 - `/usr/include/x86-64-linux-gnu/asm/unistd_32.h`
 - Llamada `exit` → Código 1
 - `read` → 3
 - `write` → 4
- El código de la llamada se pasa a través del registro `EAX`.

16.1.4. Cómo pasar los argumentos directamente al Kernel

- A diferencia de los argumentos de las llamadas a subrutinas de usuario que se pasan a través de la pila, los argumentos de las llamadas a subrutinas del sistema operativo utilizan los registros como memoria para pasar los argumentos.

- Los parámetros del primero al sexto se corresponden con los registros : *EBX, ECX, EDX, ESI, EDI, EBP*
- Valor de retorno: *EAX*

16.1.5. Como pasar los argumentos indirectamente a través de funciones libc

- Desde un módulo fuente en ASM
- Los parámetros se pasan a las funciones libc través de la *pila* y por lo tanto también a los *wrappers* de la librería de C.
- Valor de retorno: *EAX*

16.2. LEEME

- Lectura del guión de prácticas y del capítulo 3 del Libro Programming from the Ground-Up.
- [Apuntes y Libro de Texto](#)
- [Documentación Memoria](#): Contenido y Formato de la Memoria
- [Evaluación](#): sistema de evaluación
- [Plataforma de Desarrollo](#) : configuración de la computadora personal
- [Programación](#) : metodología

16.3. Cuestiones

- "Autoevaluación de Prácticas" opcional: Prácticas: Cuestionario

16.4. Llamada Exit

16.4.1. Edición del Módulo fuente:salida.c / salida.s

- `gcc -m32 -g -o salida salida.c`

```
#include <stdlib.h>
void main (void)
{
    exit (0xFF);
}
```

- `gcc -m32 -o salida salida.c`

```
/* Llamada al sistema desde C
   Prototipo:    int syscall(int number, ...);
   man syscall
*/
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>

void main (void)
```

```
{
    syscall (__NR_exit, $0xFF);
}
```

- `gcc -m32 -g -nostartfiles -o salida salida.s`

```
.global _start
.section .text
_start:
    push    $0xFF      #return code
    call    exit        #libc library
    .end
```

- `gcc -m32 -g -nostartfiles -o salida salida.s`

```
.global _start
.section .text
_start:
    push    $0xFF      #return code
    push    $1          # exit syscall code
    call    syscall     #libc library
    .end
```

- `gcc -m32 -g -nostartfiles -o salida salida.s`

```
.global _start
.section .text
_start:
    mov    $1,%eax    #exit
    mov    $0xFF,%ebx    #argument
    int    $0x80        #system call
    .end
```

16.5. LLamar a la librería de C desde código ensamblador

16.5.1. imprimir.s: printf

- `imprimir.s`

```
.section .data
planet:
    .long 9                  # variable planet

    .section .rodata
mensaje:
    .asciz "El número de planetas es %d \n"      #string con formato de la
```

```

función printf

.global _start
.section .text
_start:
    ## imprimir en la pantalla
    push planet          # 2º argumento de la función printf
    push $mensaje         # 1º argumento de la función printf: dirección del string
    call printf
    ## salir al sistema
    push $0
    call exit

```

- Compilación con *gcc* : no es necesario indicar al linker el módulo objeto *libc* ya que lo enlaza por defecto.
 - gcc -m32 -g -nostartfiles -o imprimir imprimir.s*
- Compilación con *as* y *ld*
 - as --32 -gstabs -o imprimir.o imprimir.s*
 - ld -melf_i386 -dynamic-linker /lib32/ld-linux.so.2 -o imprimir imprimir.o -lc* : enlazar con el módulo objeto *libc*

16.6. Llamadas al Sistema en la Arquitectura AMD64

- LLamadas a las funciones de la librería standard *libc*.
 - El manual de los prototipos de las funciones *libc* son accesibles en GNU con el comando *man*. Ej *man write*
 - Es necesario pasar los argumentos previamente a la ejecución de la llamada mediante la instrucción *call*.
 - Los parámetros se pasan a través de los registros *%rdi*, *%rsi*, *%rdx*, *%rcx*, *%r8* and *%r9* que se asocia con los argumentos de la función de *libc* en sentido izda→dcha.
- Preservar los registros : *%rbp*, *%rbx* and *%r12* through *%r15*
- Valor de retorno: Uno de los dos registros libres de la secuencia *%rax*, *%rdx*.
- Ejemplo:

```

#include <stdlib.h>

exit (0xFF)

```

```

xor %rax          #resetear RAX
mov    $0xFF, %rdi   #return code
call   exit        #libc library

```

```

mov    $60,%rax   #exit
mov    $0xFF, %rdi   #return code

```

syscall

Chapter 17. Subrutinas

17.1. Introducción

17.1.1. Objetivos

Programación

- Concepto de Subrutina en el Lenguaje Ensamblador AT&T x86-32
- Instrucciones de llamada y retorno: call y ret
- Argumentos: Utilización de la pila
- Instrucciones de pila: push y pop
- Estructura de la pila: punteros al bottom y top de la pila: registros EBP y ESP.
- Anidamiento de llamadas: segmentación de la pila en segmentos "Frame".
- Convenio de llamada: Pase de los parámetros, Valor de retorno, Dirección de retorno, Pila, Frame de la pila, Punteros al stack Frame, Epílogo, Prólogo
- Directivas : .type sumMtoN, @function

Análisis

- Análisis de la pila mediante el depurador GDB
 - observar la generación de un nuevo frame
 - identificar los límites del frame a través de los registros puntero.
 - volcar los argumentos, dirección de retorno y valor de retorno de la subrutina en la pila.

17.2. Módulo Fuente

- El módulo fuente *sumMtoN.s* realiza una llamada desde la rutina principal *_start* a la subrutina *sumMtoN* pasándole dos argumentos y recibiendo el resultado de la suma.
- La subrutina *sumMtoN* realiza la suma desde el número entero M hasta el número entero N donde N>M.

17.3. Requisitos

- Conceptos básicos de estructura de computadores.
- Arquitectura básica intel x86-32.
- Programación en lenguaje ensamblador AT&T: práctica con datos, modos de direccionamiento e instrucciones básicas de transferencia, aritméticas y de saltos.

17.4. LEEME

- Lectura del guión de prácticas y del capítulo 3 del Libro Programming from the Ground-Up.
- [Apuntes y Libro de Texto](#)
- [Documentación Memoria](#): Contenido y Formato de la Memoria
- [Evaluación](#): sistema de evaluación
- [Plataforma de Desarrollo](#) : configuración de la computadora personal

- [Programación](#) : metodología

17.5. Cuestiones

- "Autoevaluación de Prácticas" opcional: [Prácticas: Cuestionario](#)

17.6. Tamaño de los datos y variables

17.6.1. Algoritmo

- Desarrollar un programa en lenguaje ensamblador de la arquitectura *i386* que realice la suma $\sum_{i=1}^N i$ cuyo resultado es $N(N + 1)/2$ utilizando el [método de programación](#) de descripción inicial en lenguaje pseudocódigo y organigrama. El programa debe de contener dos módulos: uno principal referenciado con el nombre *_start* y una subrutina denominada *sumMtoN* que realiza la suma. El programa principal pasa los parámetros *M* y *N* a la subrutina para una vez realizada la suma se devuelva el resultado de la suma como valor de retorno de la subrutina.

17.6.2. Edición del Módulo fuente: sumMtoN.s

- Descargar el módulo fuente "sumMtoN.s" de miaulario y añadir los comentarios apropiados.

```
/*
Programa: sumMtoN.s
Descripción: realiza la suma de números enteros de la serie M,M+1,M+2,M+3,...N
    función : sumMtoN(1º arg=M, 2º arg=N) donde M < N
Ejecución:   Editar los valores M y N y compilar el programa.
    Ejecutar $./sumMtoN
    El resultado de la suma se capture del sistema operativo con el comando linux:
echo $?

gcc -nostartfiles -m32 -g -o sumMtoN sumMtoN.s
Ensamblaje as --32 --gstabs sumMtoN.s -o sumMtoN.o
linker -> ld -melf_i386 -o sumMtoN sumMtoN.o
*/



## MACROS
.equ    SYS_EXIT,    1
## DATOS
.section .data

## INSTRUCCIONES
.section .text
.globl _start

_start:
## Paso los dos argumentos M y N a la subrutina a través de la pila
pushl $10 #push    second argument -> N
pushl $5  #push    first argument -> M

## Llamada a la subrutina sum1toN
call sumMtoN
```

```
## Paso la salida de sum11toN al argumento a la llamada al sistema exit()
mov %eax, %ebx # (%ebx is returned)
## Código de la llamada al sistema operativo
movl $SYS_EXIT, %eax # llamada exit
## Interrumpo al S.O.
int $0x80

/*
Subrutina: sumMtoN
Descripción: calcula la suma de números enteros en secuencia desde el 1º sumando
hasta el 2º sumando
    Argumentos de entrada: 1º sumando y 2º sumando
    los argumentos los pasa la rutina principal a través de la pila:
    1º se apila el último argumento y finalmente se apila el 1º argumento.
    Argumento de salida: es el resultado de la suma y se pasa a la rutina principal
    a través del registro EAX.
    Variables locales: se implementa una variable local en la pila pero no se
utiliza
*/
.type sumMtoN, @function # declara la etiqueta sumMtoN
sumMtoN:
    ## Prólogo: Crea el nuevo frame del stack
    pushl %ebp #salvar el frame pointer antiguo
    movl %esp, %ebp #actualizar el frame pointer nuevo
    ## Reserva una palabra en la pila como variable local
    ## Variable local en memoria externa: suma
    subl $4, %esp
    ## Captura de argumentos
    movl 8(%ebp), %ebx #1º argumento copiado en %ebx
    movl 12(%ebp), %ecx #2º argumento copiado en %ecx

    ## suma la secuencia entre el valor del 1ºarg y el valor del 2ºarg
    ## 1º arg < 2ºarg
    ## utilizo como variable local EDX en lugar de la reserva externa para variable
    local: optimiza velocidad
    ## Inicializo la variable local suma
    movl $0,%edx

    ## Número de iteraciones
    mov %ecx,%eax
    sub %ebx,%eax

bucle:
    add %ebx,%edx
    inc %ebx
    sub $1,%eax
    jns bucle

    ## Salvo el resultado de la suma como el valor de retorno
```

```

    movl %edx, %eax

    ## Epílogo: Recupera el frame antiguo
    movl %ebp, %esp      #restauro el stack pointer
    popl %ebp            #restauro el frame pointer

    ## Retorno a la rutina principal
    ret
    .end

```

17.6.3. Compilación

- Seguir los pasos del proceso de [compilación](#) común a todas las sesiones.

◦ `gcc -nostartfiles -m32 -g -o sumMtoN sumMtoN.s`

17.6.4. Ejecución

- `./sumMtoN`
- `echo $?`
- Comprobar que funciona correctamente cambiando los valores de los parámetros: 1º valor de la suma y 2º valor de la suma.

17.6.5. Análisis del módulo fuente

- Leer en las hojas de referencia rápida el [Programa Ejemplo Minimalista](#)

Estructura

- La estructura del programa esta formada por los siguientes elementos:
 - Cabecera
 - Definición de Macros
 - Sección de Datos
 - Sección de Instrucciones

Ejecución modo paso a paso mediante el depurador GDB

- Compilar el programa con la opción de generación de la tabla de símbolos requerida por el depurador y generar el módulo binario ejecutable:
 - `gcc -nostartfiles -m32 -g -o sumMtoN sumMtoN.s`
- Abrir el depurador GDB, cargar el módulo binario ejecutable y comprobar que se carga la tabla de símbolos junto al módulo binario ejecutable.
 - `gdb`
 - `file modulo_ejecutable`
 - `info sources`
- Configurar el fichero para el logging histórico de los comandos.
 - `set trace-commands on`
 - `set logging file sumMtoN_gdb_asm.txt`

- `set logging on`
- `shell ls -l sumMtoN_gdb_asm.txt`
- Activar un punto de ruptura en la instrucción de entrada al programa.
 - `b _start`
- Ejecutar el programa deteniéndolo en la primera instrucción del programa.
 - `r`
- Sin ejecutar ninguna instrucción del programa
 - Estado de la pila
 - Top del stack: `x $esp` ó `x $sp` : stack pointer
 - Bottom del frame: `x $ebp` ó `x $fp` : frame pointer
 - Contenido del top de la pila (dirección sp): argc: número de argumentos string de la línea de comandos en ejecución
 - `x /xw $sp`
 - Contenido una posición anterior al top de la pila (dirección sp+4): argv[0]: dirección del 1º string de la línea de comandos en ejecución
 - `p /s *(char **)($sp+4)`
- Ejecutar las líneas necesarias hasta entrar en la subrutina:
 - Comando step: `s` ya que el comando `n` no entra en la subrutina sino que la ejecuta completamente.
 - ¿A dónde apunta el stack pointer sp? ¿Qué información contiene a donde apunta el sp?
 - `x /i *(int *)$sp` : ¿qué instrucción es?
- Ejecutar el prólogo de la subrutina
 - Nuevo frame
 - Nuevo valor del frame pointer: `p $fp`
 - Valor del stack pointer: `p $sp`
 - Acceso a la dirección de retorno tomando como refefencia el nuevo frame pointer: `x /i *(int *)($fp+4)`
- Ejecutar la subrutina hasta obtener el valor de retorno
 - Imprimir el valor de retorno: `p $eax`
- Ejecutar el epílogo de la subrutina
 - Valor del frame pointer: `p $fp`
 - Valor del stack pointer: `p $sp`
 - Dirección de retorno: `x *(int *)$sp`
- Ejecutar la instrucción de retorno
 - Dirección del stack pointer: `p $sp`
 - ¿Por qué ha cambiado la dirección del stack pointer?

Chapter 18. Imágenes: Bit Map Portable

18.1. Introducción



Práctica introductoria al examen final por lo que es necesario realizarla de forma **individual** para obtener el mayor rendimiento. Las dudas se preguntan exclusivamente al profesor ya que son de interés general.

- El objetivo de la práctica es desarrollar una subrutina en lenguaje ensamblador equivalente a una función de C dentro de una aplicación de generación de imágenes con formato BMP.
- Los cuatro primeros ejercicios en lenguaje C se realizarán de forma guiada con el profesor y el resto de forma **individual**.

18.2. Aplicación

18.2.1. Ficheros incluidos

- Descargar el archivo *bmp_practica6.zip* y extraer los ficheros.
- Scripts:
 - **comp_ejec_vis.sh**: script que automatiza las tareas de compilar, ejecutar y visualizar llamando al script Makefile.
 - **Makefile_C**: script que automatiza la tarea de compilación del programa fuente C
 - **Makefile_pixels_as**: script que automatiza la tarea de ensamblaje, compilación y enlazado de los módulos fuentes C y asm.
 - **LEEME.txt**: instrucciones de como proceder para editar los distintos programas fuente en lenguaje C y ensamblador y copiar dichos programas con los nombres apropiados antes de ejecutar el script **comp_ejec_vis.sh**
- Módulos fuente:
 - *bitmap_gen_test.c*: Genera un imagen bitmap 512x512 en formato BMP y la guarda en el fichero *test.bmp*.
 - *cuadrado_128x128.c*: Genera un imagen bitmap DIMENSIONxDIMENSION en formato BMP y la guarda en el fichero *test.bmp*.
 - *cuadrados_4.c*: Genera cuatro rectángulos anidados bitmap en formato BMP y guarda la imagen en el fichero *test.bmp*.
 - *bmp_funcion.c*: Partiendo de *bitmap_gen_test.c* el bucle generador de pixels se define mediante la función *pixels_generator(xcoor,ycoor,top,buffer)*
 - *bmp_as.c*: Módulo no incluido a desarrollar.
 - *pixels.s*: Módulo no incluido a desarrollar.
- Fichero
 - *test.bmp*

18.2.2. Ejemplo

- Copiar **cp Makefile_C Makefile**
- Copiar **cp cuadrados_4.c bmp_imagen.c** y ejecutar **comp_ejec_vis.sh**
- Copiar **cp bitmap_gen_test.c bmp_imagen.c** y ejecutar **comp_ejec_vis.sh**

- Interpretar los scripts `comp_ejec_vis.sh` y `Makefile`

18.3. Formato BMP

18.3.1. Codificación

- El formato BitMapPortable (BPM) es un formato de imagen escalar, es decir, contiene los datos de cada pixel codificando la intensidad de los componentes RGB de color tal como se visualizará en la pantalla.
- La pantalla está formada por una matriz bidimensional de pixeles, donde cada pixel es un punto discreto de la pantalla programable. La matriz de la pantalla está vinculada a una estructura de datos tipo array bidimensional 2D de filas (eje horizontal) y columnas (eje vertical) almacenada en la memoria de la tarjeta de video. El origen de coordenadas del array es la esquina inferior izquierda. A cada par (x,y) del array 2D le corresponde el color de un pixel.
- True Color: cada elemento del array contiene un dato formada por 3 campos, donde cada campo representa un color (Blue-Green-Red) y ocupa un byte . Cada componente de color R-G-B está codificado con un byte que indica la intensidad del color. Ejemplos:
 - R-G-B:0xFF-0x00-0x00 → pixel 100% rojo e intensidad máxima.
 - R-G-B:0xFF-0x00-0xFF → pixel 50% rojo y 50% azul → color morado.
 - R-G-B:0x00-0x00-0x00 → ausencia de color → color negro
 - R-G-B:0xFF-0xFF-0xFF → misma proporción de colores primarios → color blanco
 - R-G-B:0x7F-0x7F-0x7F → misma proporción de colores primarios → escala de grises entre el negro (00-00-00) y el blanco (FF-FF-FF)
- Una imagen de tamaño en pixeles 512x512 dara lugar a un array de 512 pixeles x 512 pixeles x 3 bytes/pixel

18.3.2. Mapa de memoria

- Al escribir los colores del array2D MxN en la memoria lineal donde cada dirección es **un byte**, la estructura de datos o buffer se escribe de la siguiente forma:
 - F0C0BGR-F0C1BGR-...-F0C_(N-1)BGR-F1C0BGR-...-F1C_(N-1)BGR-...-F_(M-1)C0BGR-F_(M-1)C1BGR-...-F_(M-1)C_(N-1)BGR que se corresponden con las posiciones relativas 0-1-2-3-4-5-...-(MxNx3-1)
 - F0C0BGR: pixel de la Fila cero Columna cero
 - 3 bytes en el orden azul-verde-rojo.
 - longitud total del buffer: MxNx3 bytes
 - El byte azul ocupará dentro del buffer la posición relativa 0 , el verde la posición 1 y el rojo la posición 3.
 - F0C1BGR: byte azul → posición 3 dentro del buffer
 - F0C_(N-1)BGR: El byte azul está en la posición $3*(N-1)$, el verde en $3*(N-1)+1$ y el rojo en $3*(N-1)+2$.
 - F1C0BGR: byte azul → posición $3*N$
 - F1C_(N-1)BGR: byte azul → posición $3*N+3*(N-1)$
 - F_iC_jBGR:
 - byte azul → posición $3*N*i+3*j$ donde $0 < i < M$ y $0 < j < N$
 - byte verde → posición $(3*N*i+3*j)+1$ donde $0 < i < M$ y $0 < j < N$
 - byte rojo → posición $(3*N*i+3*j)+2$ donde $0 < i < M$ y $0 < j < N$

18.3.3. Fichero

- Las imágenes con formato BMP se guardan en ficheros con extensión "*.bmp" como "test.bmp"
- El fichero BMP además del buffer de datos contiene una cabecera con metainformación que no procede explicar en este contexto.

18.4. Módulo Fuente bitmap_gen_test.c

18.4.1. Descripción

- Genera un array de pixeles y lo salva en el fichero test.bmp

18.4.2. Funciones

main()

- Descripción de bloques

```
RGB_data buffer[512][512] : variable local donde se declara y genera el array 2D  
"buffer" de pixeles donde cada pixel es del tipo RGB_data
```

Tipo RGB_data: 3 bytes consecutivos donde el primero es la intensidad de azul, el segundo verde y el tercero rojo. Las intensidades son números enteros sin signo.

memset(buffer, 0, sizeof(buffer))

- Inicializa a cero el array 2D de pixeles "buffer"
- Esta función no se encuentra en ningún módulo fuente editado por el programador, por lo que debe ser una función definida en ... ¿?. Leer el prototipo de dicha función e interpretarlo.

bmp_generator("./test.bmp", 512, 512, (BYTE*)buffer)

- Genera el fichero "test.bmp" y escribe en dicho fichero el contenido del array 2D de pixeles con nombre buffer.

bucle doble

- bucle for :
 - la variable i es el índice de filas y la variable j el índice de columnas.
 - buffer[i][j].b : byte blue del pixel de la posición (i,j)
 - buffer[i][j].g : byte green del pixel de la posición (i,j)
 - buffer[i][j].r : byte red del pixel de la posición (i,j)

VI Hojas de Referencia Rápida

Chapter 19. Programación Ensamblador AT&T x86

19.1. Programas x86-32

19.1.1. Programa Minimalista

- minimalista.s

```
### Programa: minimalista.s
### gcc -m32 -g -o minimalista minimalista.s

.global main
.section .text
main:
    mov $1, %eax # código de la llamada al sistema operativo: subrutina exit
    int $0x80      # llamada al sistema operativo
.end
```

Estructura del programa

- Una Cabecera y dos Secciones:
 - Cabecera con comentarios.
 - Sección de Datos: Se realiza la reserva de para implementar las variables inicializadas
 - Directiva `.section .data` ó únicamente `.data`
 - Sección de Instrucciones: Secuencia de instrucciones en lenguaje ensamblador
 - Directiva `.section .text` ó únicamente `.text`

Cabecera

- Cabecera con comentarios sobre:
 - Nombre del programa, lenguaje de programación.
 - Descripción del programa: entradas al programa, salidas, función del programa.
 - Entorno de programación: sistema operativo, assembler utilizado, comandos de compilación, ensamblaje, linker.
 - Comentarios sobre el autor, fecha, etc

Sección de Datos

- Directiva `.section .data`: indica el comienzo de la sección de datos
- Etiqueta `n:` :reserva de memoria en la dirección simbólica `n`
- Directiva `.int` :reserva de 4 bytes a partir de la dirección `n`: direcciones `n,n+1,n+2,n+3`
- Literal `5` :valor de inicialización de la reserva de memoria

Sección de Instrucciones : punto de entrada y bloque de salida: llamada del sistema y llamada al sistema

- Directiva `.section .text` : indica el inicio de la sección de instrucciones.
- Sintaxis de las instrucciones en lenguaje AT&T :
 - `etiqueta: operación operando_fuente,operando_destino #comentario`
- Punto de entrada al programa desde el sistema operativo:
 - El SISTEMA Operativo llama al programa o aplicación.
 - Etiqueta `_start`: Apunta a la primera instrucción del programa.
 - Directiva `.global` : La etiqueta `_start` tiene que ser "visible" fuera del programa `sum1toN` para que el linker la enlace con el sistema operativo linux como punto de entrada, es decir, tiene que ser un símbolo *global* al resto de programas y no *local* al programa `sum1toN`. El linker `ld` por defecto presupone que el símbolo utilizado como etiqueta del punto de entrada es `_start`.
- Punto de salida del programa al sistema operativo:
 - Es necesario acabar con el proceso del programa `sum1toN` y liberar todos los recursos que este utilizando dicho proceso. Esta tarea de fin de proceso la tiene que realizar el sistema operativo o kernel linux.
 - El programa o aplicación llama al El SISTEMA Operativo.
 - El programa `sum1toN` llama al sistema operativo para realizar la operación de fin de proceso mediante la ejecución de la función `exit(argumento)` . El sistema operativo tiene un listado de posibles funciones que ejecuta si es llamado. Una de dichas funciones es `exit(argumento)`.
 - Llamada al Sistema Operativo en lenguaje ensamblador:
 - Registro EAX: almacena el código de la función a ejecutar por el Sistema Operativo. El código de la función `exit` es 1.
 - Registro EBX: almacena el código del argumento de la función `exit(argumento)`. El valor 0 se interpreta como ejecución del programa correcta.
 - Instrucción `int $0x80` : esta instrucción llama al sistema operativo, INTerrumpe al sistema operativo para que ejecute la función asociada al código almacenado en el registro EAX.
 - [Programming from the Ground Up](#)

Fin del ensamblaje

- Directiva `.end`

19.1.2. Ejemplo Básico

- Módulo fuente: `sum1toN.s`

```
### Programa: sum1toN.s
### Descripción: realiza la suma de la serie 1,2,3,...N
### Es el programa en lenguaje AT&T i386 equivalente a sum.ias de la máquina IAS de
von Neumann
### gcc -m32 -g -nostartfiles -o sum1toN sum1toN.s
### Ensamblaje as --32 --gstabs sum1toN.s -o sum1toN.o
### linker -> ld -melf_i386 -o sum1toN sum1toN.o
      ## 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

```

- Compilación: `gcc -nostartfiles -m32 -g -o sum1toN sum1toN.s`
- Ensamblaje: `as --32 --gstabs sum1toN.s -o sum1toN.o`
- linker: `ld -melf_i386 -o sum1toN sum1toN.o`
- Directivas del traductor ensamblador: `.section`, `.data`, `.text`, `.byte`, `.end`, etc... empiezan con un punto como prefijo

| | | | | | |
|---------------------|--------------------------|-----------------------------|----------------|----------------------------------|-----------------------|
| <code>label:</code> | <code>op_mnemonic</code> | <code>operand_source</code> | <code>,</code> | <code>operand_destination</code> | <code>;comment</code> |
|---------------------|--------------------------|-----------------------------|----------------|----------------------------------|-----------------------|

- Las etiquetas llevan el sufijo :
- La etiqueta `_start`: es el punto de entrada al programa. Obligatoria. La utiliza el linker.
- Sufijos de los mnemónicos
 - **b** → byte → 1Byte → Ej: `movb`
 - **w** → word → 2Bytes → Ej: `movw` . En este contexto word son 2 bytes por razones históricas.
 - **l** → long → 4Bytes → Ej: `movl` . Valor por defecto.
 - **q** → quad → 8Bytes → Ej: `movq`
- Direccionamientos de los operandos:
 - En la misma instrucción los operandos fuente y destino no pueden hacer ambos referencia a la memoria Principal.
 - inmediato: prefijo del operando **\$**
 - registro: prefijo del registro **%**
 - directo: el operando es una etiqueta que apunta a la memoria principal
 - indirecto: el operando es una etiqueta o un registro: utiliza paréntesis. (etiqueta) ó (%registro). Ver indexado.
 - La etiqueta referencia una posición de memoria que contiene a su vez una dirección de la memoria principal que apunta al operando.
 - El registro contiene la dirección de la memoria principal que apunta al operando.
 - indexado
 - dirección efectiva: `base + index*scale + disp` → la sintaxis es: `disp(base,índice,escala)`
 - `foo(%ebp,%esi,4)` → dirección efectiva= `EBP + 4*ESI + foo`

- $(\%edi) \rightarrow$ dirección efectiva= *EDI* → direccionamiento indirecto
- $-4(\%ebp) \rightarrow$ dirección efectiva= *EBP - 4*
- $\text{foo}(\%,\text{eax},4) \rightarrow$ dirección efectiva= $4*\text{EAX} + \text{foo}$
- $\text{foo}(,1) \rightarrow$ dirección efectiva= *foo*
- Cualquier instrucción que tiene una referencia a un operando en la memoria principal y no tiene una referencia a registro, debe especificar el tamaño del operando (byte, word, long, or quadruple) con una instrucción que lleve el sufijo ('b', 'w', 'l' or 'q', respectivamente).

19.2. Directivas Assembler AS

- Manual
 - <https://sourceware.org/binutils/docs/as/>

Table 23. Directivas básicas

| .global o .globl | 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 |
| .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 carácter 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 |

| | |
|-------------------------|---|
| .global o .globl | variables globales |
| .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 múltiplo de n bytes. |
| .end | fin del ensamblaje |

- Alineamiento **LittleEndian**: El byte de menor peso, LSB, se almacena en la posición de memoria más baja.
 - .int Ox AABBCCDD → 0xDD se almacena primero en la dirección más baja, el resto de bytes se almacenan en sentido ascendente en el orden 0xCC,0xBB,0xAA

19.3. Repertorio de Instrucciones Ensamblador

- [Manuales del Repertorio de Instrucciones](#)
- Lenguaje Ensamblador AT&T

19.3.1. 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 | | | | | | | | | |
| 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 | | | | | | | | | |

| Nombr e | Comentario | Código | Operación | O | D | I | T | S | Z | A | P | C |
|---------|-------------------------------|-----------------|---|---|---|---|---|---|---|---|---|---|
| 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

19.3.2. 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 | ? | ? | ? | ? | ? | ? | | | |

| Nombr e | Comentario | Código | Operación | O D I T S Z A P C |
|---------|---------------------------------|-----------------------|--|-------------------|
| MUL | Multiplicación (sin signo) | MUL Op | Op=byte: AX:=AL*Op si AH=0 # | ± ? ? ? ? ± |
| MUL | Multiplicación (sin signo) | MUL Op | Op=word: DX:AX:=AX*Op si DX=0 # | ± ? ? ? ? ± |
| MUL | 386 Multiplicación (sin signo) | MUL Op | Op=double: EDX:EAX:=EAX*Op si EDX=0 # | ± ? ? ? ? ± |
| IMUL | i Multiplic. entera con signo | IMUL Op | Op=byte: AX:=AL*Op si AL es suficiente # | ± ? ? ? ? ± |
| IMUL | Multiplic. entera con signo | IMUL Op | Op=word: DX:AX:=AX*Op si AX es suficiente # | ± ? ? ? ? ± |
| IMUL | 386 Multiplic. entera con signo | IMUL Op | Op=double: EDX:EAX:=EAX*Op si EAX es sufi. # | ± ? ? ? ? ± |
| INC | Incrementar | INC Op | Op:=Op+1 (El Carry no resulta afectado !) | ± ± ± ± ± |
| DEC | Decrementar | DEC Op | Op:=Op-1 (El Carry no resulta afectado !) | ± ± ± ± ± |
| CMP | Comparar | CMP 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

19.3.3. 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 |

| Nombre | Comentario | Código | Operación | O D I T S Z A P C |
|--------|-----------------------------|-----------------|-----------|-------------------|
| SHL | Desplazam. lógico a la izq. | SHL Op,Cantidad | | i ± ± ? ± ± |
| SHR | Desplazam. lógico a la der. | SHR Op,Cantidad | | i ± ± ? ± ± |

19.3.4. MISCELÁNEOS

| Nombre | 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 |

19.3.5. SALTOS (generales)

- [wiki x86 assembly](#)

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

19.3.6. 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 | J0 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 |
| JGE | Saltar si es mayor o igual | JGE Dest |
| JL | Saltar si es menor | JL Dest |
| JLE | Saltar si es menor o igual | JLE Dest |
| JNG | Saltar si no es mayor | JNG Dest |
| JNGE | Saltar si no es mayor o igual | JNGE Dest |
| JNL | Saltar si no es inferior | JNL Dest |
| JNLE | Saltar si no es menor o igual | JNLE Dest |

19.3.7. 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

- 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*

19.4. Registros

19.4.1. Visión completa

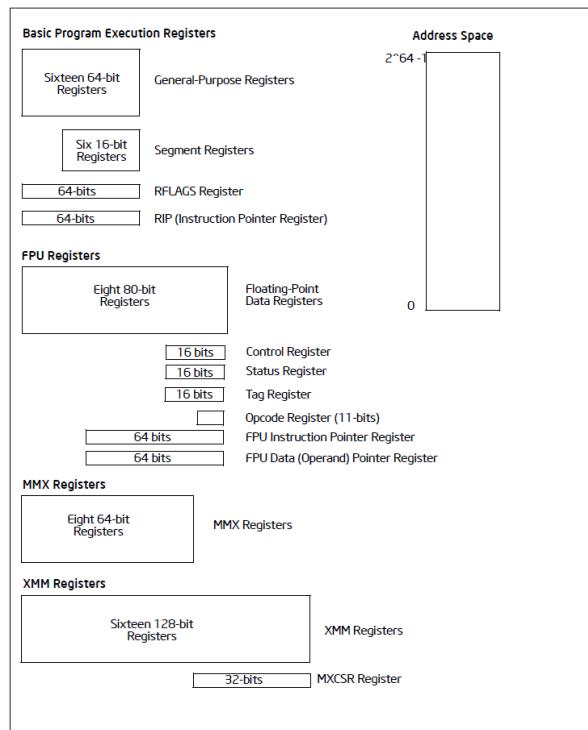


Figure 3-2. 64-Bit Mode Execution Environment

- Los registros de propósito general RPG de 32 bits son:
 - '%eax' (el acumulador), '%ebx', '%ecx', '%edx', '%edi', '%esi', '%ebp' (puntero frame), and '%esp' (puntero stack).

| register encoding | not modified for 8-bit operands | | | | low 8-bit | 16-bit | 32-bit | 64-bit |
|----------------------|--|--|-----|-------|--------------|--------|--------|--------|
| | not modified for 16-bit operands zero-extended for 32-bit operands | | | | | | | |
| 0 | | | AH* | AL | | AX | EAX | RAX |
| 3 | | | BH* | BL | | BX | EBX | RBX |
| 1 | | | CH* | CL | | CX | ECX | RCX |
| 2 | | | DH* | DL | | DX | EDX | RDX |
| 6 | | | | SIL** | | SI | ESI | RSI |
| 7 | | | | DIL** | | DI | EDI | RDI |
| 5 | | | | BPL** | | BP | EBP | RBP |
| 4 | | | | SPL** | | SP | ESP | RSP |
| 8 | | | | R8B | | R8W | R8D | R8 |
| 9 | | | | R9B | | R9W | R9D | R9 |
| 10 | | | | R10B | | R10W | R10D | R10 |
| 11 | | | | R11B | | R11W | R11D | R11 |
| 12 | | | | R12B | | R12W | R12D | R12 |
| 13 | | | | R13B | | R13W | R13D | R13 |
| 14 | | | | R14B | | R14W | R14D | R14 |
| 15 | | | | R15B | | R15W | R15D | R15 |

63 32 31 16 15 8 7 0

| | |
|---|--|
| 0 | |
| | |

63 32 31 0

RFLAGS 513-309.eps

RIP

* Not addressable when a REX prefix is used.

** Only addressable when a REX prefix is used.

Figure 2-3. General Registers in 64-Bit Mode

19.4.2. Registros visibles al programador

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

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

19.4.3. Compatibilidad 32-64

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

| | |
|---------|---------|
| 64 bits | 32 bits |
| RIP | EIP |
| RAX | EAX |
| RFLAG | EFLAG |
| | |

- Hay excepciones

19.4.4. Control Flag Register

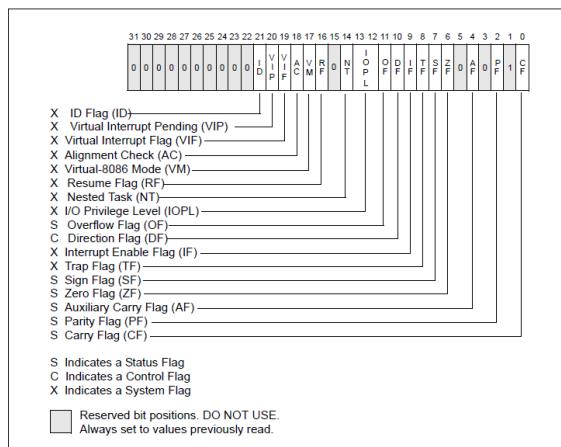


Figure 3-8. EFLAGS Register

- 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](#)
- Únicamente nos fijamos en los flags OSZAPC.

Table 24. 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 se sale del ancho de palabra 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 este fuera del tamaño) indica error en la operación aritmética con números enteros con signo.
- 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

GDB QUICK REFERENCE GDB Version 5**Essential Commands**

gdb [core] debug program [using coredump core]
 b [file:]line set breakpoints at function [in file]
 run [arglist] start your program [with arglist]
 bt backtrace: display program stack
 p expr display the value of an expression
 c continue running your program
 n next line, stepping over function calls
 s next line, stepping into function calls

Starting GDB

gdb start GDB, with no debugging files
 gdb program begin debugging program
 gdb program core debug coredump core produced by
 program
 gdb --help describe command line options

Stopping GDB

quit exit GDB; also q or EOF (eg C-d)
 INTERRUPT (eg C-c) terminate current command, or
 send to running process

Getting Help

help list classes of commands
 help class one-line descriptions for commands in
 class
 help command describe command

Executing your Program

run arglist start your program with arglist
 run start your program with current arguments
 run ... <inf>>outf start your program with input, output
 redirected
 kill kill running program
 tty dev use dev as stdin and stdout for next run
 set args arglist specify arglist for next run
 set args specify empty argument list
 show args display arguments list
 show env show all environment variables
 show env var show value of environment variable var
 set env var string set environment variable var
 unset env var remove var from environment

Shell Commands

cd dir change working directory to dir
 pwd Print working directory
 make ... call "make"
 shell cmd execute arbitrary shell command string

[] surround optional arguments ... show one or more arguments

©1998, 2000, 2010 Free Software Foundation, Inc. Permissions on back

Breakpoints and Watchpoints

break [file:]line set breakpoints at line number [in file]
 b [file:]line eg: break main.c:37
 break [file:]func set breakpoints at func [in file]
 break *offset set break at offset lines from current stop
 break *addr set breakpoint as address addr
 break set breakpoints at next instruction
 break ... if expr break conditionally on nonzero expr
 cond n [expr] new conditional expression on breakpoint
 n; make unconditional if no expr
 tbreak ... temporary break; disable when reached
 rbreak [file:]reger break on all functions matching reger [in
 file]
 watch expr set a watchpoints for expression expr
 catch event break at event, which may be catch,
 throw, exec, fork, vfork, load, or
 unload.
 info break show defined breakpoints
 info watch show defined watchpoints
 clear delete breakpoints at next instruction
 clear [file:]fun delete breakpoints at entry to fun()
 clear [file:]line delete breakpoints on source line
 delete [n] delete breakpoints [or breakpoint n]
 disable [n] disable breakpoints [or breakpoints n]
 enable [n] enable breakpoints [or breakpoints n];
 enable once [n] enable breakpoints [or breakpoints n];
 enable del [n] enable breakpoints [or breakpoints n];
 delete when reached
 ignore n count ignore breakpoint n, count times
 commands n execute GDB command-list every time
 [silent] breakpoint n is reached. [silent
 command-list suppresses default display]
 end end of command-list

Program Stack

backtrace [n] print trace of all frames in stack; or of n
 frames—innermost if n>0, outermost if
 n<0
 bt [n] select frame number n or frame at address
 frame [n] n; if no n, display current frame
 up n select frame n frames up
 down n select frame n frames down
 info frame [addr] describe selected frame, or frame at addr
 info args arguments of selected frame
 info locals local variables of selected frame
 info reg [rn]... register values [for reg, rn] in selected
 frames; all-reg includes floating point
 info all-reg [rn]

Execution Control

continue [count] continue running; if count specified, ignore
 this breakpoint next count times
 step [count] execute until another line reached; repeat
 count times if specified
 s [count] step by machine instructions rather than
 source lines
 next [count] execute next line, including any function
 calls
 n [count] next machine instruction rather than
 source line
 until [location] run until next instruction (or location)
 finish run until selected stack frame returns
 return [expr] pop selected stack frame without
 executing [setting return value]
 signal num resume execution with signal s (none if 0)
 jump line resume execution at specified line number
 jump *address or address
 set var=expr evaluate expr without displaying it; use
 for altering program variables

Display

| | |
|-------------------|--|
| print [/f] [expr] | show value of expr [or last value \$] according to formats f: |
| p [/f] [expr] | x hexadecimal d signed decimal u unsigned decimal o octal t binary a address, absolute and relative c character f floating point |
| call [/f] expr | like print but does not display void |
| x [/Nuf] expr | examine memory at address expr; optional format spec follows slash N count of how many units to display u unit size: one of b individual bytes h halfwords (two bytes) w words (four bytes) g giant words (eight bytes) |
| f | printing format. Any print format, or s null-terminated string i machine instructions |
| disassm [addr] | display memory as machine instructions |

Automatic Display

display [/f] expr show value of expr each time program
 stops [according to format f]
 display display all enabled expressions on list
 undisplay n remove number(s) n from list of
 automatically displayed expressions
 disable disp n disable display for expression(s) number n
 enable disp n enable display for expression(s) number n
 info display numbered list of display expressions

| | | |
|--|--|--|
| Expressions | Controlling GDB | Source Files |
| <code>expr</code> an expression in C, C++, or Modula-2 (including function calls), or: <code>addr@len</code> an array of <code>len</code> elements beginning at <code>addr</code> <code>file::nm</code> a variable or function <code>nm</code> defined in <code>file</code> <code>{type} addr</code> read memory at <code>addr</code> as specified <code>type</code> <code>\$</code> most recent displayed value <code>\$n</code> nth displayed value <code>\$\$</code> displayed value previous to \$ <code>\$\$n</code> nth displayed value back from \$ <code>\$_</code> last address examined with <code>x</code> <code>\$_var</code> value at address <code>\$_var</code> <code>\$var</code> convenience variable; assign any value | <code>set param value</code> set one of GDB's internal parameters <code>show param</code> display current setting of parameter Parameters understood by <code>set</code> and <code>show</code> : <code>complaint limit</code> number of messages on unusual symbols <code>confirm on/off</code> enable or disable cautionary queries <code>editing on/off</code> control readline command-line editing <code>height lpp</code> number of lines before pause in display <code>language lang</code> language for GDB expressions (auto, c or modula-2) <code>listsize n</code> number of lines shown by <code>list</code> <code>prompt str</code> use <code>str</code> as GDB prompt <code>radix base</code> octal, decimal, or hex number representation <code>verbose on/off</code> control messages when loading symbols <code>width cpl</code> number of characters before line folded <code>write on/off</code> Allow or forbid patching binary, core files (when reopened with <code>exec</code> or <code>core</code>) <code>history ...</code> groups with the following options: <code>b ...</code> <code>b exp off/on</code> disable/enable readline history expansion <code>b file filename</code> file for recording GDB command history <code>b size size</code> number of commands kept in history list <code>b save off/on</code> control use of external file for command history <code>print ...</code> groups with the following options: <code>p ...</code> <code>p address on/off</code> print memory addresses in stacks, values <code>p array off/on</code> compact or attractive format for arrays <code>p demangle on/off</code> source (demangled) or internal form for C++ symbols <code>p asm-dem on/off</code> demangle C++ symbols in machine-instruction output <code>p elements limit</code> number of array elements to display <code>p object on/off</code> print C++ derived types for objects <code>p pretty off/on</code> struct display: compact or indented <code>p union on/off</code> display of union members <code>p vtbl off/on</code> display of C++ virtual function tables <code>show commands</code> show last 10 commands <code>show commands n</code> show 10 commands around number <code>n</code> <code>show commands +</code> show next 10 commands | <code>dir names</code> add directory names to front of source path <code>dir</code> clear source path <code>show dir</code> show current source path <code>list</code> show next ten lines of source <code>list -</code> show previous ten lines <code>list lines</code> display source surrounding lines, specified as: <code>[file:]num</code> line number [in named file] <code>[file:]function</code> beginning of function [in named file] <code>+off</code> off lines after last printed <code>-off</code> off lines previous to last printed <code>*address</code> line containing address <code>list f,l</code> from line <code>f</code> to line <code>l</code> <code>info line num</code> show starting, ending addresses of compiled code for source line <code>num</code> <code>info source</code> show name of current source file <code>info sources</code> list all source files in use <code>forw regex</code> search following source lines for <code>regex</code> <code>rev regex</code> search preceding source lines for <code>regex</code> |
| Symbol Table | | GDB under GNU Emacs |
| <code>info address s</code> show where symbol <code>s</code> is stored <code>info func [regex]</code> show names, types of defined functions (all, or matching <code>regex</code>) <code>info var [regex]</code> show names, types of global variables (all, or matching <code>regex</code>) <code>whatis [expr]</code> show data type of <code>expr</code> [or \$] without evaluating; <code>ptype</code> gives more detail <code>ptype [expr]</code> describe type, struct, union, or enum <code>ptype type</code> | | <code>M-x gdb</code> run GDB under Emacs <code>C-h m</code> describe GDB mode <code>M-s</code> step one line (<code>step</code>) <code>M-n</code> next line (<code>next</code>) <code>M-i</code> step one instruction (<code>stopi</code>) <code>C-c C-f</code> finish current stack frame (<code>finish</code>) <code>M-c</code> continue (<code>cont</code>) <code>M-u</code> up arg frames (<code>up</code>) <code>M-d</code> down arg frames (<code>down</code>) <code>C-x &</code> copy number from point, insert as end <code>C-x SPC</code> (in source file) set break at point |
| GDB Scripts | | GDB License |
| <code>source script</code> read, execute GDB commands from file <code>script</code> <code>define cmd</code> create new GDB command <code>cmd</code> ; execute script defined by <code>command-list</code> <code>and</code> end of <code>command-list</code> <code>document cmd</code> create online documentation for new GDB command <code>cmd</code> <code>help-text</code> end of <code>help-text</code> <code>end</code> | | <code>show copying</code> Display GNU General Public License <code>show warranty</code> There is NO WARRANTY for GDB. Display full no-warranty statement. |
| Signals | | |
| <code>handle signal act</code> specify GDB actions for <code>signal</code> : <code>print</code> announce signal <code>noprint</code> be silent for signal <code>stop</code> halt execution on signal <code>nostop</code> do not halt execution <code>pass</code> allow your program to handle signal <code>nopass</code> do not allow your program to see signal <code>info signals</code> show table of signals, GDB action for each | | |
| Debugging Targets | | |
| <code>target type param</code> connect to target machine, process, or file <code>help target</code> display available targets <code>attach param</code> connects to another process <code>detach</code> release target from GDB control | | |

19.5. GDB

- Comandos básicos: ejecutar el comando `gdb`

```

shell date
shell pwd
shell ls
shTAB
shell daTAB
C-x a
C-x o
histórico comandos: navegar con las flechas
set trace-commands on
set logging file gdb_salida.txt
set logging on
shell ls -l gdb_salida.txt
file modulo_bin
info sources
info source
break main
b _start
info breakpoints
info reTAB
run

```

Copyright ©1991, 1992, 1993, 1998, 2000, 2010 Free Software Foundation, Inc. Author: Roland H. Pesch

The author assumes no responsibility for any errors on this card.

This card may be freely distributed under the terms of the GNU General Public License.

Please contribute to development of this card by annotating it.

Improvements can be sent to bug-gdb@gnu.org.

GDB itself is free software; you are welcome to distribute copies of it under the terms of the GNU General Public License. There is absolutely no warranty for GDB.

```
next , n , n 5
step , s
RETURN
continue, c
start
until, RETURN, RETURN ...
next instruction, ni, RET, RET, RET, RET, until, RET,..hasta salir del bucle
step ,s
si
ptype variable
whatis variable
print variable, p variable, p /t variable, p /x n
p &n
p $rax
p $eax
p $ax
p $ah
p $rflags
p $eflags
p /t $eflags
p $rip
x dirección
x &variable, x /1bw &variable, +x /1xw &variable, x /4xw &variable
pending? n
layout split
h layout
layout src
focus src : navegar
focus cmd : navegar
info registers
info reTAB
disas /r _start
disas /m _start
q
```

VII Autoevaluación Prácticas

Chapter 20. Prácticas: Cuestionario

20.1. Práctica 1^a: Introducción a la Programación en Lenguaje Ensamblador AT&T x86-32

20.1.1. Cuestiones teóricas

1. Cuál es la principal diferencia entre el lenguaje ensamblador AT&T y el propio de Intel.
2. Qué fases comprende el toolchain
3. Lista las herramientas de desarrollo a utilizar durante la realización de las prácticas mediante los dos procesadores utilizados.
4. Libro: Programming from the Ground-Up
 - a. Qué es GNU/Linux
 - b. Qué es GNU
 - c. Qué es gcc
 - d. Qué gestiona el kernel
 - e. ¿Se puede acceder simultáneamente a instrucciones y datos? ¿Por qué?
 - f. Cuál es la función del registro PC
 - g. Cuáles son los dos tipos de registros de la CPU
 - h. Qué significa Word Size
 - i. Qué es una variable puntero.
 - j. Lista cuatro modos diferentes de direccionar un operando.

20.1.2. Cuestiones prácticas

1. Comando de compilación del programa fuente ensamblador mediante el front-end *gcc* que incluya la tabla de símbolos para el depurador
2. Comando de enlace (linker) del módulo objeto reubicable.
3. Declaración en lenguaje C de la variable *n* tipo entero con signo de un byte.
4. Instrucción en lenguaje ensamblador del programa *sum1toN.s* que realiza una suma.
5. Comandos del depurador *gdb* para la impresión del contenido de la variable *n*
6. Ejecutar *sum1toN*, compilado de *sum1toN.s*, paso a paso mediante el depurador GDB ejecutando los comandos necesarios para:
 - a. imprimir el contenido de la variable *n* y su dirección en memoria principal
 - b. imprimir la dirección de la etiqueta bucle
 - c. imprimir el contenido del registro *ECX* al salir del bucle
7. Cambiar el tamaño de los operandos de la suma a 2 bytes
 - a. Cambiar el tamaño del operando *n* → `n: .word 5`
8. Cambiar la instrucción `add %edx,%ecx` por la instrucción `addw %dx,%ec`
9. En GDB qué comando hay que utilizar para ejecutar todas las iteracciones del bucle del programa de forma continuada.
10. Comparando las versiones en lenguajes C y ASM de los módulos fuente, por qué la instrucción *until* del depurador GDB en el caso del módulo fuente en lenguaje C se ejecuta durante la sentencia *while*.

11. Editando el módulo fuente provocar un error de ensamblaje al no haber coherencia entre la declaración del tamaño del operando referenciado por la etiqueta *n* y la declaración de tamaño de operandos de la instrucción *add*.

20.2. Práctica 2^a: Representación de los Datos

20.2.1. Módulo datos_size.s

- ¿En qué orden se guardan los caracteres del string "hola"?
- ¿Cuál es el código ASCII del carácter *o*?
- ¿Cuál es la dirección de memoria principal donde se almacena el string "hola"?
- ¿Cuál es la dirección memoria principal donde se almacena el array lista?
 - ¿Cuál es el contenido de los primeros 4 bytes a partir de la dirección anterior en sentido ascendente?

20.2.2. Módulo datos_sufijos.s

- ¿En qué orden se guardan los bytes del dato *da4*?
- ¿Cuál es el resultado de ejecutar `mov da1,%ecx`?

20.2.3. Módulo datos_direccionamiento.s

- Con el depurador ejecutar el programa en modo paso a paso realizando las siguientes operaciones.
- Array *da2*
 - Imprimir la dirección de memoria del array *da2* y el contenido del primer elemento: `x /xh &da2`
 - 4 elementos de 2bytes del array *da2*: `x /4xh &da2`
 - `ptype da2`: no debug info: al no tener información el debugger del tamaño de los elementos es necesario indicarlos explícitamente en los comandos posteriores.
 - Es necesario realizar un **casting** : Array de 4 elementos de tamaño 2bytes: `p /x (short[4])da2`
 - Fijarse con el comando *eXaminar* el resultado es independiente de si hacemos un **casting** (`short *`): `x /4xh (short *)&da2`
 - El tamaño y tipo de dato lo fija el argumento del comando: `/4xh`
 - Comprobar la norma de almacenamiento *little endian* identificando cada dirección de memoria a un byte con su contenido.
 - Acceder a la dirección de memoria del elemento de valor -21 del array *da2*:
 - el argumento elemento de array en `p da2[2]` no es válido ya que el debugger carece de información
- Desensamblar
 - `disas salto1`
 - `disas /r salto1`

20.3. Práctica 3^º: Operaciones Aritmetico-Lógicas e Instrucciones de Salto Condicionales

20.3.1. Módulo op_arit_log.s

- Módulo fuente `op_arit_log`

- Indicar cómo asociar el valor de los sumandos a las macros OPE1 y OPE2.
- Sin cambiar el valor de los operandos:
 - indicar el valor de la resta en la instrucción 1
 - indicar el valor de la multiplicación en la instrucción 2
 - indicar el valor de la división en la instrucción 3
 - indicar el valor de la división en la instrucción 4
 - indicar el valor de las operaciones lógicas en la instrucción 5

20.3.2. Módulo saltos.s

- Registro de Flags
 - Editar, compilar y ejecutar el siguiente bloque de instrucciones para indicar el contenido del registro EAX y el estado de los flags CF,ZF,SF,PF,OF después de la ejecución de cada instrucción :

```
mov $0xFFFFFFFF,%eax
shr $1,%eax
add %eax,%eax
testb $0xFF,%eax
cmpl $0xFFFFFFFF,%eax
```

- Saltos
 - Editar, compilar y ejecutar el siguiente bloque de instrucciones para indicar el estado de los flags CF,ZF,SF,PF,OF antes de la ejecución de la instrucción de salto e indicar si se produce o no el salto.

```
mov $0x00AA, %ax
mov $0xFF00, %bx
cmp %bx,%ax
ja salto1
jg salto2
salto1: mov $0xFF,%ebx
salto2: mov $1,%eax
int $0x80
```

20.4. Práctica 4: Llamadas al Sistema Operativo

20.4.1. Módulo syscall_write_puts.c

- Desarrollar un programa en lenguaje C que imprima en la pantalla el mensaje de bienvenida "Hola". Utilizar las funciones puts(), write() y syscall() de la librería standard de C.
 - `man 2 puts`
 - `man 2 write` : prototipo de la función

WRITE(2) Linux Programmer's Manual

NAME

`write - write to a file descriptor`

SYNOPSIS

```
#include <unistd.h>

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

- *fd*: file descriptor: el monitor es un fichero virtual con descriptor número 1.
- *void *buf*: buffer es un puntero que apunta a la cadena de caracteres a imprimir.
- *count*: tamaño máximo de la cadena de caracteres a imprimir
- Esta función llama indirectamente al sistema operativo a través de la llamada `syscall()`.

```
/*
Programa syscall_write_puts.c
Descripción: Realiza la llamada al sistema operativo para imprimir en la
pantalla
Realiza la llamada de tres formas diferentes: puts, write, syscall.
Compilación: gcc -m32 -g -o puts_gets puts_gets.c
*/



// Cabeceras de librerías
#include <stdio.h> // prototipo de la función puts()
#include <unistd.h> //declaración de las macros STDOUT_FILENO, STDIN_FILENO
#include <syscall.h> //declaración de la función syscall
#include <sys/syscall.h> // declaración de la macro __NR_write y __NR_exit
#include <stdlib.h> //declaración de exit()

// Macros
#define LON_BUF 5 // Tamaño del string


void main (void)
{
    char buffer[LON_BUF] = "Hola\n";

    puts("\n***** Práctica : LLAMADAS AL SISTEMA
*****\n"); // función puts() de la librería libc
    puts("\n***** Imprimo el mensaje de bienvenida mediante la
función write(): ");

    write(STDOUT_FILENO, buffer, LON_BUF); // wrapper de la llamada al sistema
    write.                                         // ya que write() incluye un
    syscall(), llama indirectamente al sistema

    puts("\n***** Imprimo el mensaje de bienvenida mediante la
```

```

llamada al sistema syscall(): ");
    syscall(__NR_write,STDOUT_FILENO,buffer,LON_BUF); // función syscall de
    llamada directa al sistema.
    exit(0xAA); //Salir al sistema enviando el código 0xAA. No es lo mismo que
    retornar.

}

```

20.4.2. Módulo `syscall_write_puts.s`

- Desarrollar un programa en lenguaje ensamblador x86-32 `syscall_write_puts.s` equivalente al programa `syscall_write_puts.c` llamando a las funciones `puts()` y `write()`. En lugar de utilizar la función `syscall()` realizar la llamada al sistema directamente con la instrucción `int 0x80`. Los argumentos de la llamada al sistema operativo se pasan a través de los registros:
 - 1º argumento: a través de EAX: tipo int: valor `__NR_write`: valor 4
 - 2º argumento: EBX: tipo `int fd`: valor `STDOUT_FILENO`: valor 1
 - 3º argumento: ECX: tipo `const void *buf`: puntero al string a imprimir , buffer
 - 4º argumento: EDX: tipo `size_t count`: valor `LON_BUF`: valor 5
 - Descripción RTL

```

# llamada a la función puts de la librería libc. Es necesario linkar con libc.
pila <-argumento
call puts
# llamada a la función write de la librería libc. Es necesario linkar con libc.
pila <- 3º argumento
pila <- 2º argumento
pila <- 1º argumento
call write
# Llamada al sistema operativo para ejecutar la operación write
EAX<-4
EBX<-1
ECX<-etiqueta que apunta al string a imprimir
EDX<-5
call sistema_operativo
# Llamada al sistema operativo para ejecutar la operación exit
EAX<-1
EBX<-0
call sistema_operativo

```

20.5. Práctica 5: LLamadas a una Subrutina

20.5.1. Módulo `sumMtoN_aviso.c`

- Desarrollar el programa `sumMtoN_aviso.c` equivalente al módulo en lenguaje asm `sumMtoN.s` y añadiendo un mensaje de aviso en caso de error indicando la relación correcta entre los parámetros 1º sumando y 2º sumando.

20.5.2. Módulo sumMtoN_aviso.s

- Añadir al programa fuente *sumMtoN.s* un mensaje de aviso en caso de error indicando la relación correcta entre los parámetros 1º sumando y 2º sumando.

20.6. Práctica 6: Imagen Bit Map Portable

20.6.1. Programación en C

- Leer el procedimiento de programación en el fichero **LEEME.txt**
- El objetivo es modificar la función principal **main()** del programa original **bitmap_gen_test.c** dando lugar a distintos programas independientes entre sí.
 1. - Compilar y ejecutar el program *bitmap_gen_test.c*
 2. - visualizar la imagen del fichero test.bmp: **\$display test.bmp**
 3. - Módulo **cuadrado_128x128.c** :Cambiar las dimensiones de la imagen a 128 pixeles x 128 pixeles definiendo la macro DIMENSION=128 y definiendo para cada pixel un color gris con una intensidad del 50% de su valor máximo.
 4. - Módulo **cuadrados_4.c**: Generar 4 cuadrados, uno dentro de otro simétricamente, donde el cuadrado mayor negro es 512x512 y el resto se reduce 1/8 cada uno. No utilizar ctes en las sentencias de C, utilizar las macros x_coor, y_coor, top para indicar el valor inicial del **for** y la posición máxima (top) de las filas y columnas. Colores de los cuadrados: background (00-00-00)/(FF-00-FF)/(00-FF-FF)/(FF-FF-00)/
 5. - Módulo **bmp_funcion.c**: El bloque de código que realiza el bucle para inicializar los pixeles del cuadrado convertirlo en la función:
 - prototipo: *void pixels_generator(unsigned int x, unsigned int y, unsigned int maximo, RGB_data reg_mem[][top])*
 - x e y son el origen de coordenadas del cuadrado
 - maximo es la coordenada mayor del cuadrado
 - llamada a la función: *pixels_generator(xcoor,ycoor,top,buffer);*
 - los argumentos xcoor=top/8, ycoor=top/8 y top=512 definirlos mediante macros

20.6.2. Programación en ASM

1. - Módulo **bmp_as.c**: Implementar la función *void pixels_generator(unsigned int maximo, RGB_data reg_mem[][top])* desarrollando en lenguaje ensamblador la subrutina *pixels_generator* en el nuevo fichero **array_pixel.s**. El fichero en lenguaje ensamblador únicamente contendrá la subrutina.
 - La subrutina implementa el doble bucle.
 - De forma implícita en la propia subrutina consideraremos los argumentos x=y=0.
 - Azul, rojo y verde son las intensidades de todos los pixeles del cuadrado.

20.6.3. GDB

1. En el programa en **bmp_funcion.c** indicar la posición de la pila donde se salva la dirección de retorno de la subrutina **pixels_generator**, así como el contenido del frame pointer y del stack pointer.
2. Lo mismo que en el apartado anterior con el programa **bmp_as.c** para la subrutina *pixels_generator*

VIII Apéndices

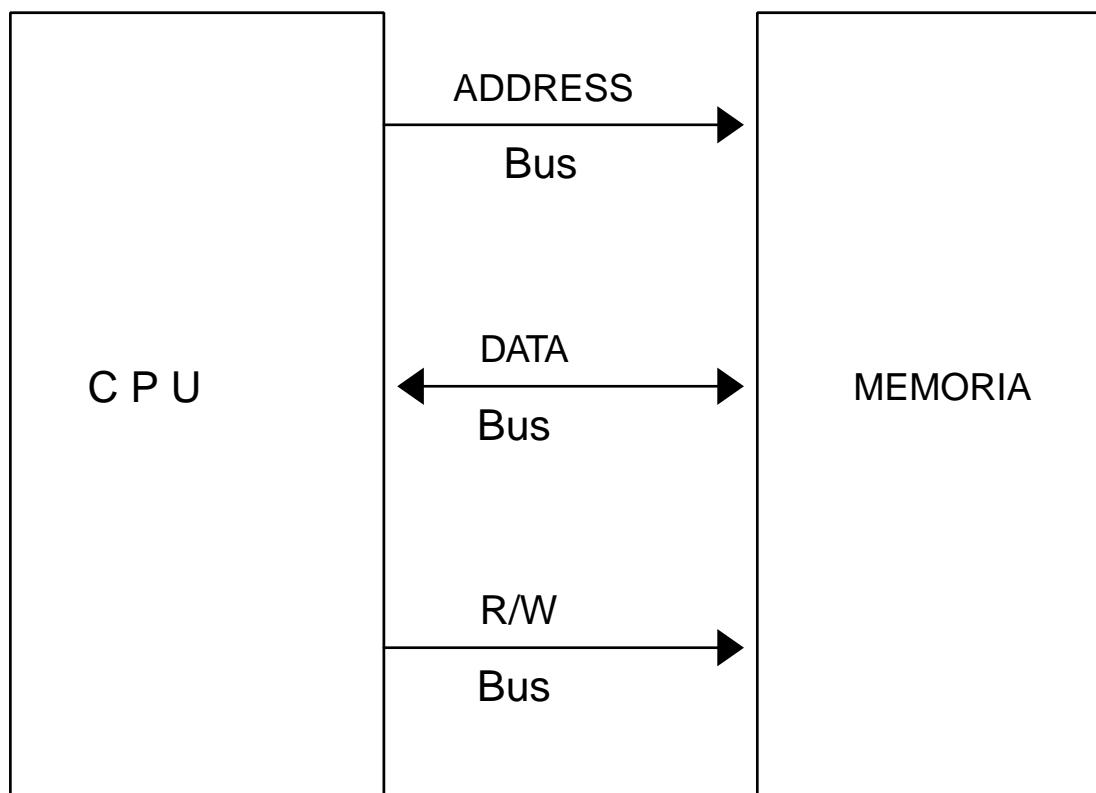
Chapter 21. Arquitectura de una Computadora

21.1. Estructura de la Computadora

21.1.1. Contexto

- El significado del término arquitectura en una computadora depende del contexto.
 - La primera gran división sería: Arquitectura Hardware y Arquitectura Software
 - 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 lenguaje máquina y por lo tanto tienen la misma arquitectura. En este contexto utilizaremos el término 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 microprocesador: es la organización interna de un procesador.

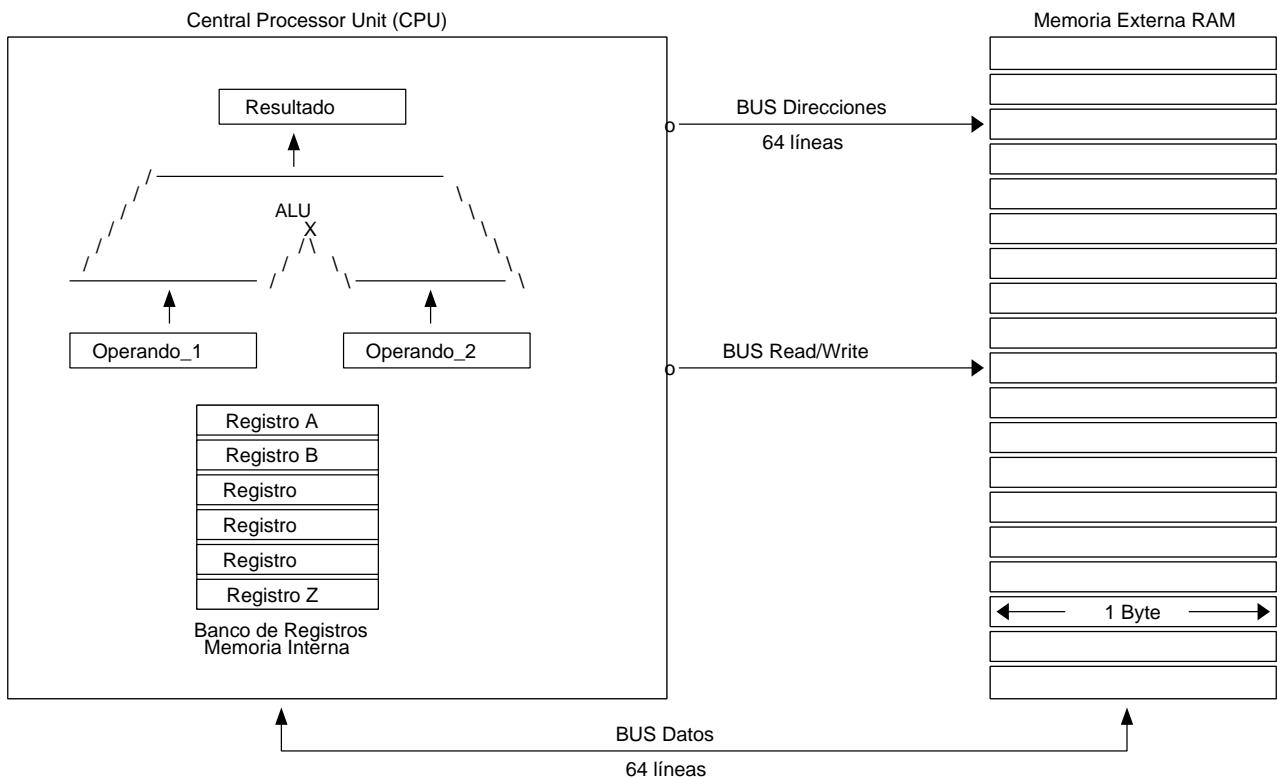
21.1.2. Arquitectura HW



21.1.3. 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



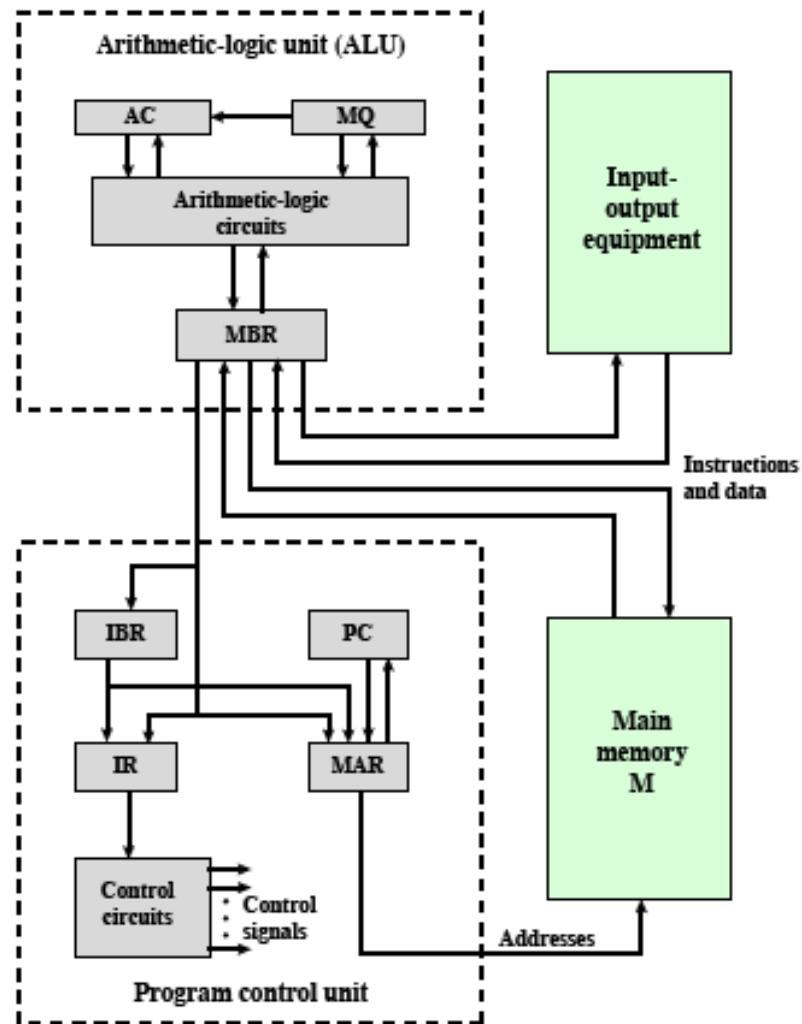


Figure 2.3 Expanded Structure of IAS Computer

Figure 77. IAS Structure

21.1.4. 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 hexadecimal
- 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.

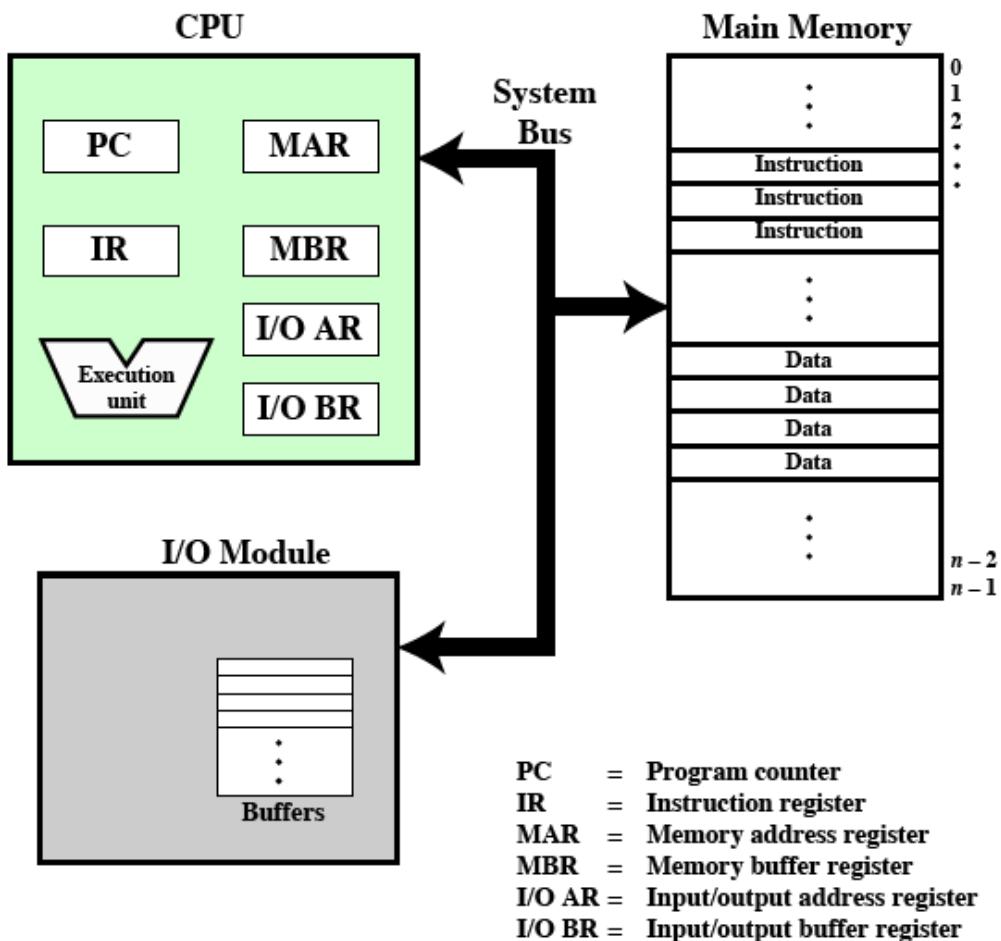
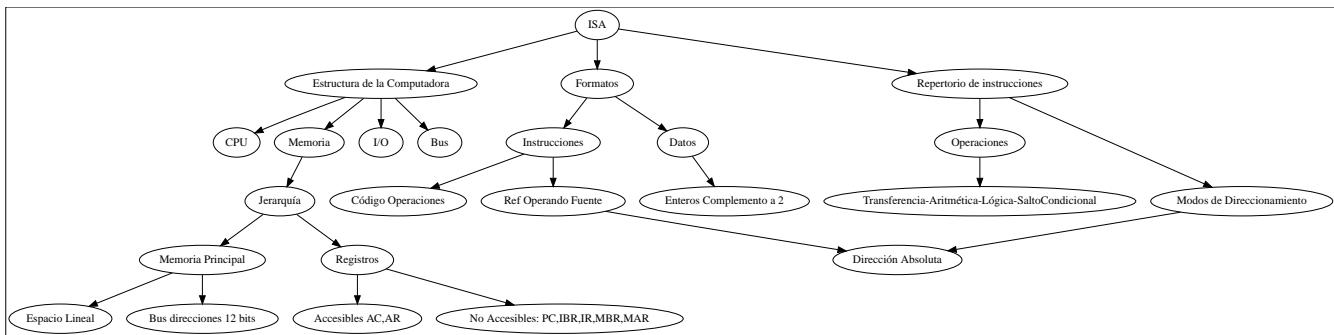


Figure 78. IAS_Architecture

21.2. 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.



21.2.1. Ejemplos: Intel x86, Motorola 68000, MIPS, ARM

- Ver Apéndice Lenguajes Ensamblador

21.3. Procesadores Intel con arquitectura x86

21.3.1. 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](#): 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
 - <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](#)

Chapter 22. RTL Register Transfer Language

22.1. Lenguaje RTL

22.1.1. 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.

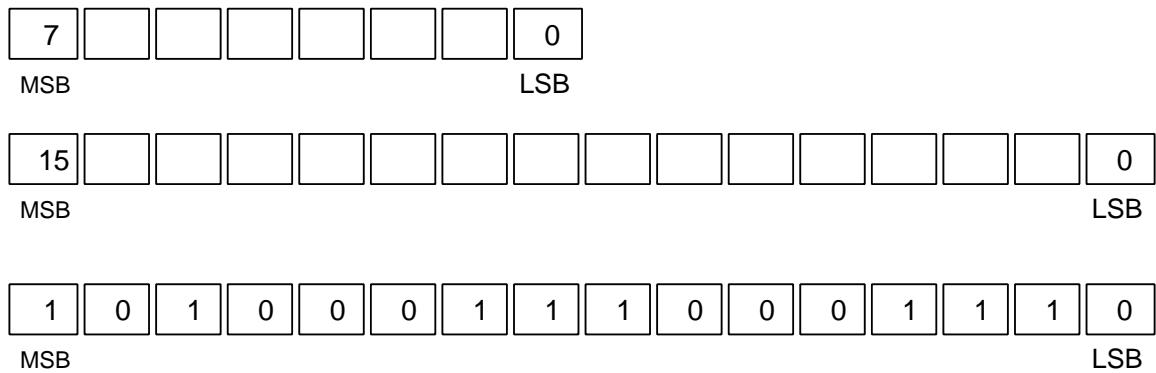


No confundir el RTL (Register Transfer Language) con el RTL (Register Transfer Level). El Register Transfer Level es utilizado por los lenguajes de descripción de HARDWARE (Hardware Description Language HDL)

22.1.2. Registros

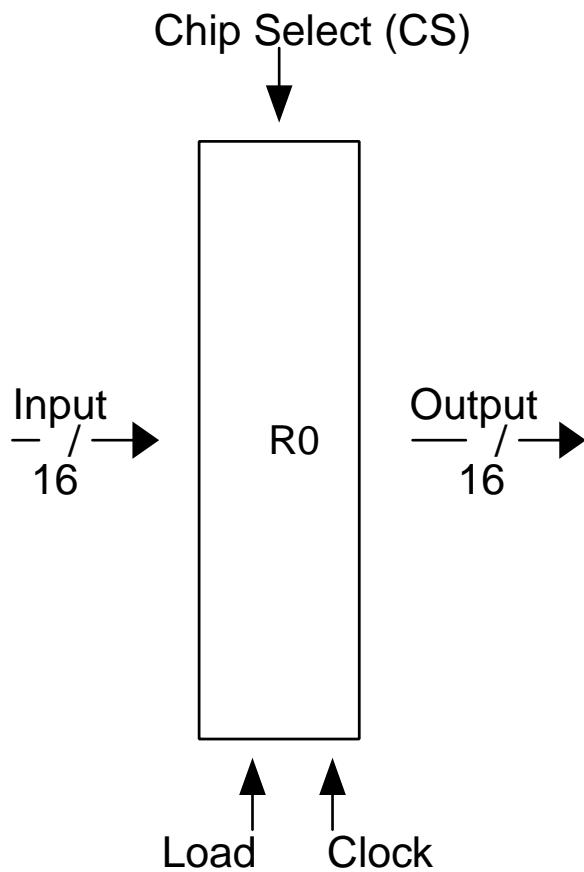
Arquitectura

- 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



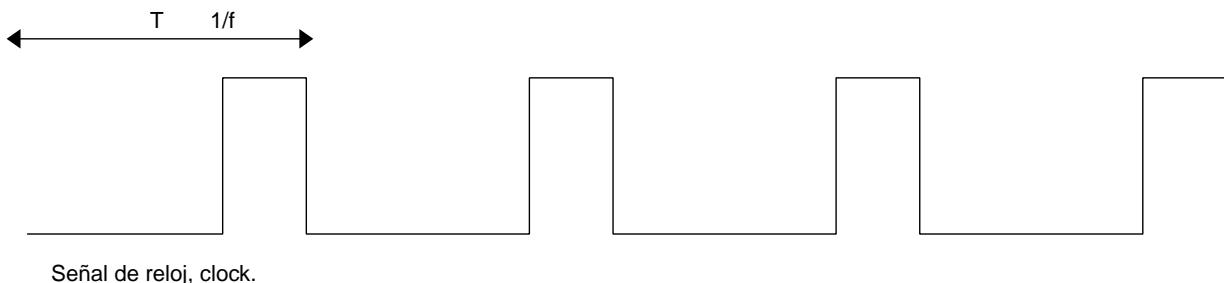
Estructura

- 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



22.1.3. 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.

22.1.4. 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←R1
 - K1:R2←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←R1,R3←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 ← M[0x80000] : copiar el contenido de memoria de la posición 0x8000 al registro AC
- AC ← M[AC] : copiar el contenido de la posición de memoria a la que **apunta** el registro AC en el registro AC
- M[0x8000] ← AC: copiar el contenido del registro AC en la posición de Memoria 0x8000
 - M[0x8000] ← 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] ← M[0x2000]
 - El contenido de la posición 0x2000 se copia en la posición 0x1000
 - Lo que hay a la derecha del operador ← se evalua y se obtiene un VALOR
 - Lo que hay a la izda del operador ← es una DIRECCION o REFERENCIA a Memoria (Principal o Registro)

22.1.5. Ejemplos RTL con expresiones aritmético-lógicas

- AC ← R1 v R2
 - Operación lógica OR
- (K1+K2):R1 ← R2+R3,R4←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 23. Programas ensamblador IASSim

23.1. Ejemplo 1: sum1toN.ias

- **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 ;

```

23.1.1. 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 de un número entero por 2 puede tener resto 1 ó 0 dependiendo de si el dividendo es par o impar
 - Si N es impar $\rightarrow (N+1)/2$ tiene el resto 0 $\rightarrow ((N + 1)/2) * N$ donde N+1 es par
 - Si N es par $\rightarrow (N+1)/2$ tiene el resto 1 $\rightarrow (N+1) = \text{Cociente}*2 + \text{Resto} \rightarrow N(N + 1)/2 = N * C + N / 2$ 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

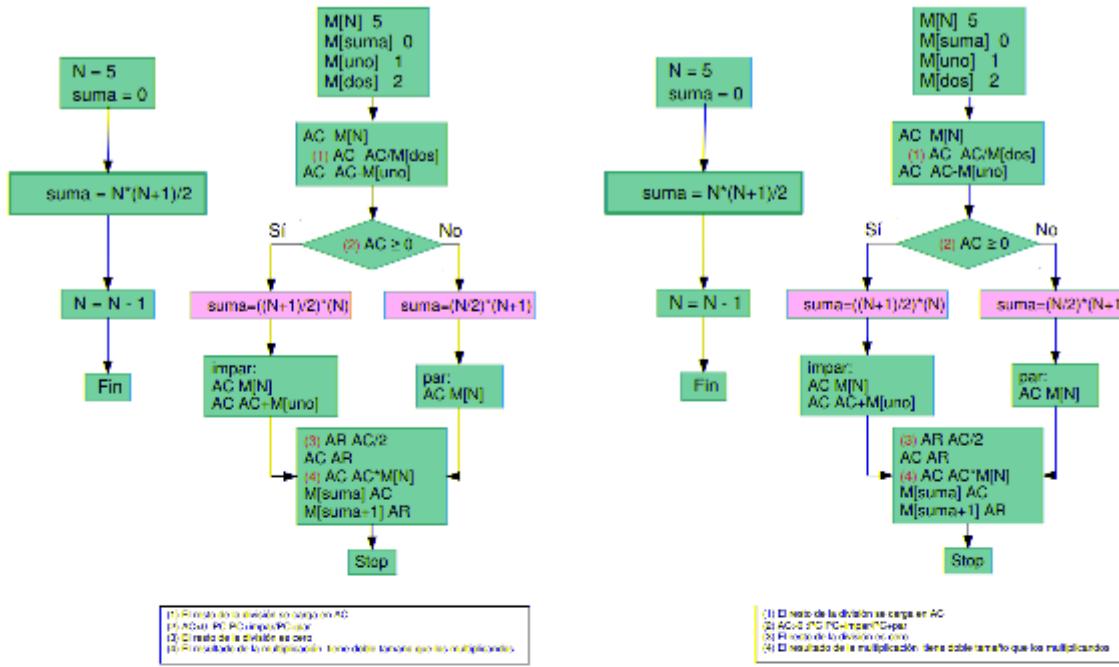


Figure 79. Diagrama de Flujo: Pseudocódigo y RTL

Lenguaje Ensamblador IAS

- versión sum1toN_mul_A.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

```

- versión simplificada sum1toM_mul.ias: realizo primero la multiplicación $N(N+1)$ y el resultado siempre es par. A continuación divido por 2.

```

; 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
S(x)->Ac+ n    ;01 n    ;AC    <- M[n]
S(x)->Ah+ uno  ;05 uno ;AC    <- AC+1
At->S(x) y      ;11 y    ;M[y]  <- AC
S(x)->R y      ;09 y    ;AR    <- M[y]
S(x)*R->A n    ;0B n    ;AC:AR <- AR*M[n]
; Caso particular donde AC=0
R->A            ;0A      ;AC    <- AR
A/S(x)->R dos  ;0C 2   ;AR    <- AC/2
;Al ser par el dividendo el resto es cero
R->A            ;0A      ;AC    <- AR
At->S(x) y      ;11 y    ;M[y]  <- AC
halt
; como el numero de instrucciones es par no es necesaria la directiva .empty

```

```

; 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 IASSim

23.1.2. 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

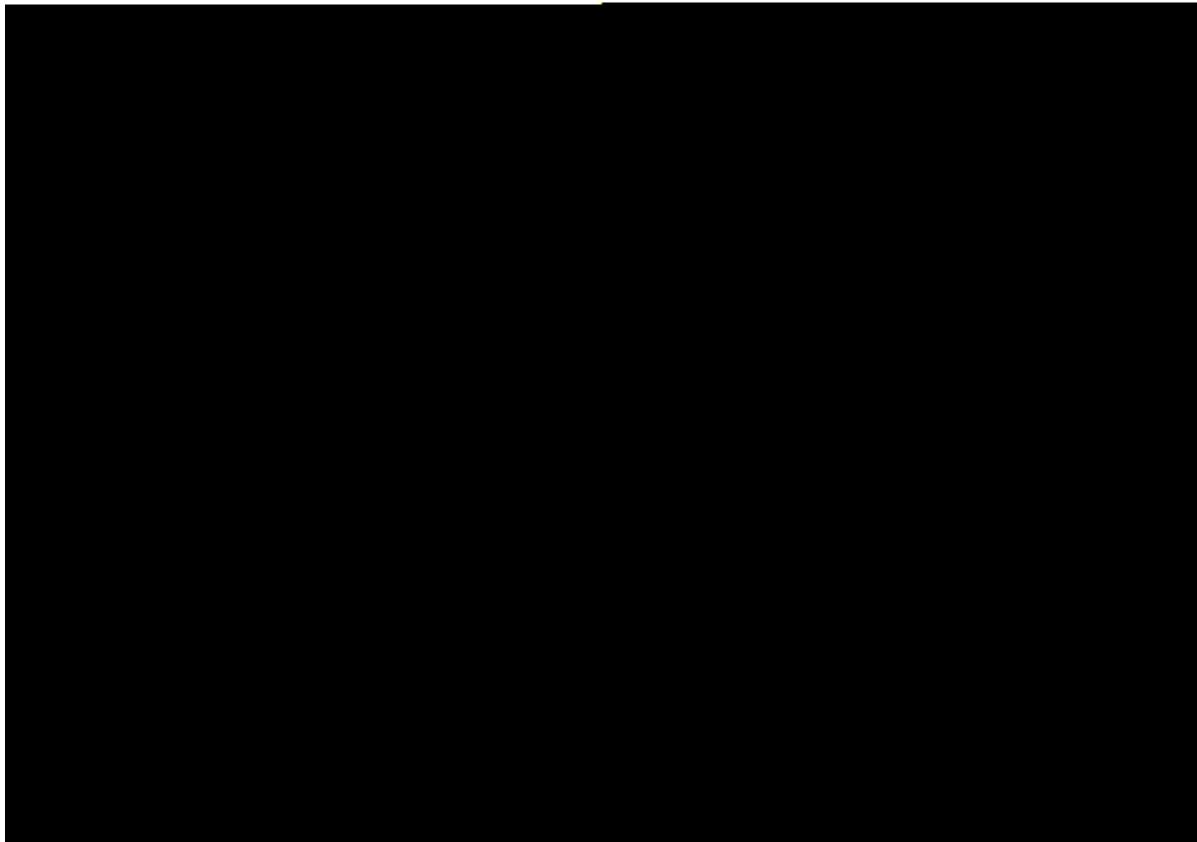
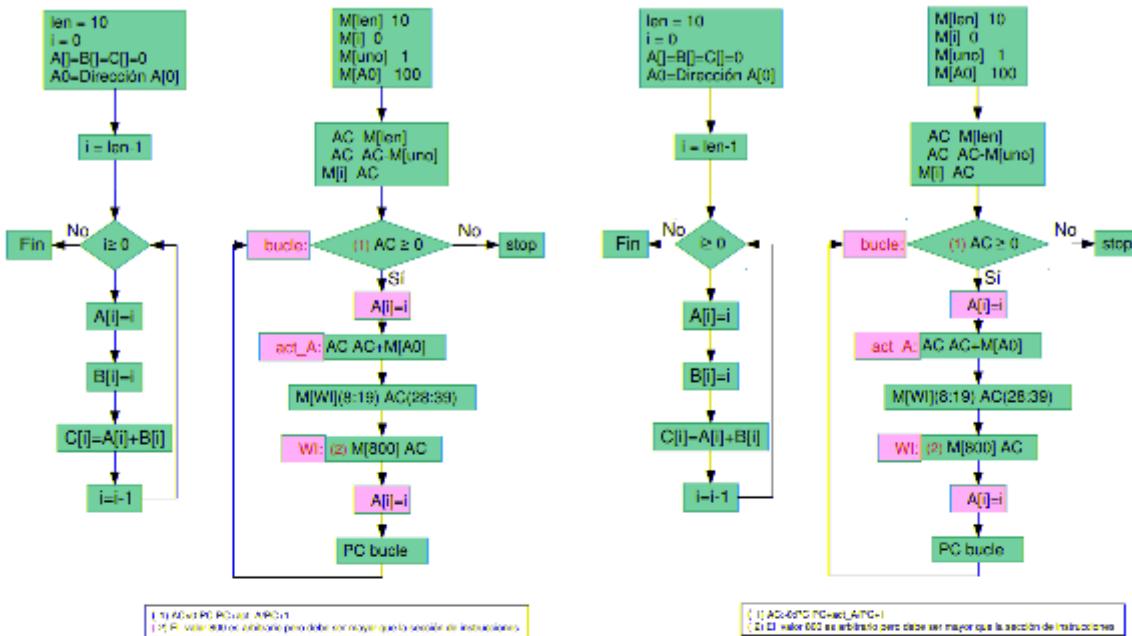


Figure 80. Diagrama de Flujo: Pseudocódigo y RTL

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 guarda 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 24. Simulador IASSim

24.1. Máquina Virtual Java JVM

- Instalar el Kit de Desarrollo Java ([Java Development Kit-JDK](#)) en el sistema ubuntu
 - [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.

24.2. 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.
- [Colby IASSim Web](#) : 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.

24.3. 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 81. 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 82. 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 25. Lenguajes de programación de Alto y Bajo Nivel

25.1. Lenguajes de programación de alto nivel vs lenguajes de bajo nivel

- Aunque en los años 1960 se programaba en el lenguaje máquina y lenguaje ensamblador, esto resultaba muy poco productivo ya que requería 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.

25.2. Ejemplo sum1toN en distintos lenguajes de Programación

- Algoritmo sum1toN $\rightarrow \sum_{i=1}^N i = N(N + 1)/2$
- 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 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
## Declaracin de variables
## SECCION DE DATOS
.section .data

n:    .quad 5

.global _start

## Comienzo del cdigo
## 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 travs de RDI segn convenio ABI AMD64
## salida
    mov $60, %rax ## cdigo de la llamada al sistema operativo: subrutina exit
    syscall ## llamada al sistema operativo para que ejecute la subrutina segn el
valor de RAX

.end

```

- ARM

```

/*
Programa: sum1toN.s
Descripcin: realiza la suma de la serie 1,2,3,...N
Es el programa en lenguaje ARM equivalente a sum1toN.ias de la mquina 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
*/
@ Declaracin de variables
.section .data
n:    .int 5

.global _start

@ Comienzo del cdigo
.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 26. Lenguajes de programación en Ensamblador

26.1. Manuales de referencia

26.1.1. Lenguaje Intel

- Manuales oficiales
 - [Intel](#): Vol 2
 - [AMD](#): apartado Manuals : vol 3
 - [AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions](#)
- Manuales no oficiales:
 - manual Intel quick: **recomendado**
 - [intel descriptivo i386](#)
 - [Repertorio ISA y Formato de Instrucción](#)
 - [kluge](#)
 - [Saltos Condicionales](#)

Netwide ASM (NASM) para el lenguaje de intel

- 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](#)

26.1.2. lenguaje AT&T

- [Oracle Solaris ASM](#)
 - En este documento a la sintaxis AT&T la denomina "Oracle Solaris".
 - [AT&T Solaris Manual amd64-i386](#) : lenguaje y traductor assembler.

26.1.3. Características arquitectura i386

- [Assembler: Características dependientes de la arquitectura x86](#)

26.1.4. Assembler (Traductor Ensamblador): Directivas

- [Directivas del traductor Assembler](#)

26.1.5. Discusión por qué ASM AT&T

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

26.1.6. 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 | | | | | | | | | |
| 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

26.1.7. 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 !) | ± ± ± ± ± |
| 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 ± |

| Nombr e | Comentario | Código | Operación | O D I T S Z A P C |
|---------|----------------------|-----------------|-----------|-------------------|
| 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

26.1.8. 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 ± ± ? ± ± |

26.1.9. 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 |

26.1.10. SALTOS (generales)

- [wiki x86 assembly](#)

| Nombre | Comentario | Código | Operación |
|--------|-----------------------------|-----------|-----------|
| CALL | Llamado a subrutina | CALL Proc | |
| JMP | Saltar | JMP Dest | |
| JE | Saltar si es igual | JE Dest | (= JZ) |
| JZ | Saltar si es cero | JZ Dest | (= JE) |
| JCXZ | Saltar si CX es cero | JCXZ Dest | |
| JP | Saltar si hay paridad | JP Dest | (= JPE) |
| JPE | Saltar si hay paridad par | JPE Dest | (= JP) |
| JPO | Saltar si hay paridad impar | JPO Dest | (= JNP) |
| JNE | Saltar si no es igual | JNE Dest | (= JNZ) |

| | | | |
|-------|--------------------------|------------|---------|
| JNZ | Saltar si no es cero | JNZ Dest | (= JNE) |
| JECXZ | Saltar si ECX es cero | JECXZ Dest | 386 |
| JNP | Saltar si no hay paridad | JNP Dest | (= JPO) |
| RET | Retorno de subrutina | RET | |

26.1.11. 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 | 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) |

26.1.12. 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

26.1.13. 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*

26.2. Intel x86-32 /i386

26.2.1. sum1toN.s

- Lenguaje ensamblador ATT para la arquitectura x86-32

```
### Programa: sum1toN.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
```

26.2.2. hola_mundo.s

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

###

```

#### hola_mundo.s
####
#### Programa simple de iniciaci n para el desarrollo de programas en
Ensamblador x86-32 AT&T.
####
#### Compilaci n:
#### assemble using: as --32 hola_mundo.s -o hola_mundo.o
#### link using: ld -melf-i386 hola_mundo.o -o hola_mundo
#### Driver gcc: gcc -m32 -nostartfiles hola_mundo.s -o
hola_mundo
####
#### revised on: Septiembre 2022 -- for Linux's i386 environment
####
####

.att_syntax

## Declaraci n de s mbolos externos
.global _start      # visible entry-point

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

mensaje: .ascii "Hola mundo\n"
longitud: .2byte . - mensaje      #tama o en bytes de la cadena mensaje

## Secci n para el C digo de las Instrucciones en Lenguaje Ensamblador
.section .text

_start:

    mov    $4, %eax      # SYS_WRITE
    mov    $1, %ebx      # device ID-number
    mov    $mensaje, %ecx # message address
    mov    longitud, %edx # message length
    int    $0x80          # enter the kernel

## terminate this program
    mov    $1, %eax      # SYS_EXIT
    mov    $0, %ebx      # return value
    int    $0x80          # enter the kernel

.end               # no more to assemble

```

26.2.3. hola_mundo: C digo M quina Binario

- Secci n de Datos

```
08049ff4 <mensaje>: 48 6f 6c 61 20 4d 75 63 64 6f 0a      H o l a S P 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 de Instrucciones**

```
## Sección para el Código de las Instrucciones en Lenguaje Ensamblador
```

```
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

## terminate this program

080481a7: b8 01 00 00 00      mov    $0x1,%eax
080481ac: bb 00 00 00 00      mov    $0x0,%ebx
080481b1: cd 80                int    $0x80
```

Volcado de un programa binario

- Mediante el comando: `objdump -d hola_mundo`, donde hola_mundo es el módulo binario ejecutable.

Almacenamiento del programa binario en la Memoria Principal

- 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:
 - instrucción máquina arquitectura amd64.
 - **4001a4: 48 83 ec 10** → **subq \$16,%rsp**
 - **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

Interpretación de una instrucción en Código Máquina: Formato de Instrucción de la ISA Intel x86-64

- 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.
 - [manual oficial de intel x86 ó x86-64](#): 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.

26.3. Intel x86-64 / AMD 64

26.3.1. 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
```

26.3.2. Hola Mundo

- Módulo Fuente hola_mundo.s en lenguaje ensamblador.

```
### -----  
### hola_x86-64_att.s  
###  
### Programa simple de iniciacin para el desarrollo de programas en  
Ensamblador x86-64 AT&T.  
###  
### Ficheros complementarios: macros_x86-64_gas.h  
###  
###  
### Compilacin:  
###      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 Linuxs x86_64 environment  
###  
### -----  
.att_syntax  
  
## Incluir el fichero con las Macros  
.include "macros_x86-64_gas.h"  
  
## Declaracin de smbolos externos  
.global _start      # visible entry-point  
  
## Reserva de Memoria para datos variables  
.section .data  
  
msg0:  .ascii "Hola Mundo\n"  
len0:   .quad  . - msg0    #tamao en bytes de la cadena msg0  
  
## Seccin para el Cdigo 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'

```

26.3.3. Miscellaneous

Tipos de Datos

- Tipos de Datos:
 - Dirección de memoria o referencia a memoria: la etiqueta de nombre 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

Números Reales

- `kluge`

- interesantes los ejemplos de operaciones con números reales

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 \leftarrow M[0x4001a4]
 - IR \leftarrow MBR
 - AC \leftarrow RSP
 - AC \leftarrow AC-16 ; (ALU resta)
 - RSP \leftarrow AC
 - PC \leftarrow PC+1
 - MAR \leftarrow PC

26.3.4. sum1toN.s: lenguaje intel

- 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
```

26.4. Arquitectura ARM

26.4.1. 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
```

26.4.2. ISA

- **ARM**: Advanced RISC Machine

- Developer Guides

26.5. Motorola 68000

26.5.1. 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
```

26.5.2. ISA

- Referencias
 - [Instruction Set Basic](#)
 - [Wikibook](#)
 - [Manual de Referencia](#)
 - [Motorola 68K ó M68000](#)
 - 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)
 - Prejuegos datos: \$ hexadecimal
- Memoria
 - Big Endian : LSB en la dirección más alta y MSB en la dirección más baja

26.6. Arquitectura MIPS

26.6.1. 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 $x \leftarrow (y)op(z)$

Datos:

Enteros Complemento a 2 : byte (1B), media palabra (2B), palabra (4B)

Nº Reales: IEEE-754 simple y doble precisión

- [MIPS architecture](#)
- [Versiones de la ISA MIPS](#)
- [procesadores con arquitectura MIPS: R2000, etc](#)
- [quick tutorial](#)
- [Emulador MIPS Online](#)

Chapter 27. Toolchain: Cadena de Herramientas en el proceso de compilación

27.1. Toolchain

- Se conoce por Toolchain a los distintos programas o herramientas que intervienen en la obtención de una aplicación en lenguaje máquina a partir de módulos en lenguajes simbólicos como C, ensamblador y módulos librería.
- Las 3 herramientas principales en el proceso de traducción son : el compilador, el ensamblador y el linker. También hay herramientas de análisis de los programas en lenguaje binario (Pej desensamblador).
 - [binutils](#):
 - [as: traductor assembler](#): opciones de la línea de comandos, directivas, tipos de datos, etc
 - [ld: linker](#)
 - [objdump, readelf,](#)

27.2. Proceso de compilación de un programa en lenguaje C

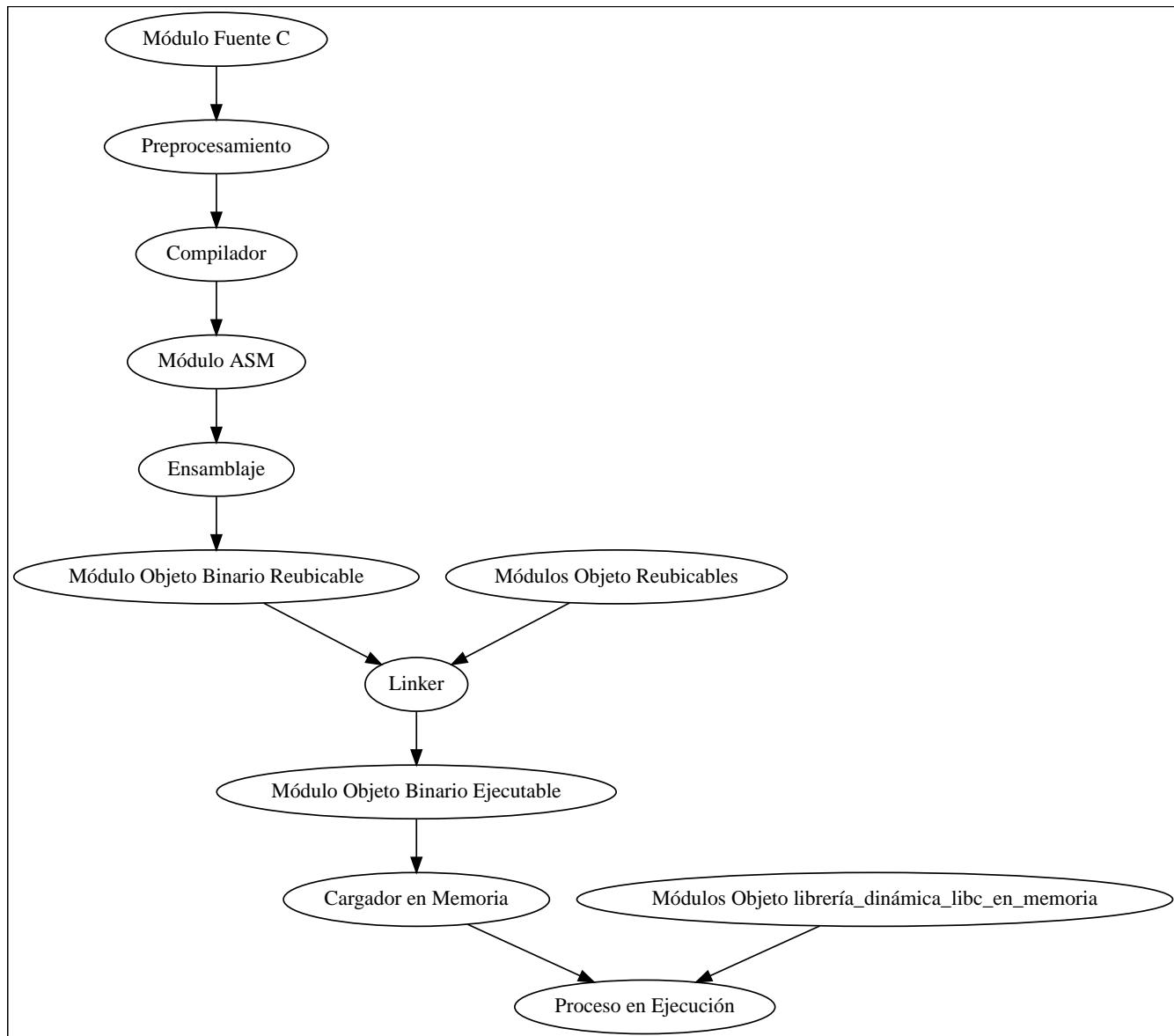


Figure 83. Proceso de Compilación

27.3. 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.

27.4. Assembler "as"

27.4.1. Directivas

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

Manual

- [binutils as](#): manual oficial
- [Linux Assembly howto](#): referencias a diferentes assemblers
- [MIT](#)

Chapter 28. Practicando la Programación desde el principio

28.1. 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](#) 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*.
 - Libro de introducción a la programación en lenguaje ensamblador : '[Programar ASM ... pero sí es muy fácil](#)

28.2. Plataforma de Desarrollo

28.2.1. Herramientas

- Editores
 - **Editores**: gedit, emacs, vim, sublime, kate,
 - **Herramientas integradas de edición, compilación, depuración**: eclipse CDT, netbeans, code::blocks, codelite, Microsoft's Visual Studio Code Editor, jetbrains clion, jenany, 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

28.2.2. Programación online

- [ias assembler unicamp online](#)
- [gdb online](#)

28.2.3. 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**

28.3. Documento Memoria: Contenido y Formato

28.3.1. Contenido

- Durante el desarrollo de la práctica :
 - Es necesario reeditar el código fuente de los programas desarrollados con *comentarios*.
 - Compilar el módulo fuente mediante *comandos en línea*
 - Anализar 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.

28.3.2. 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 ni acentos, ni espacios en blanco
 - XXX significa el grupo de prácticas: P1 ó P2 ó P1P2 ó P91
 - N significa el número de la sesión de prácticas: 1,2,3,4 ó 5.

28.3.3. 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.

28.4. 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**. Si no se realizan los ejercicios de autoevaluación de prácticas y se añaden a las memorias la nota máxima de las memorias será de **6 puntos**.



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.

28.5. Programación

28.5.1. 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**

28.6. Compilación

28.6.1. 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 depurador 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).

28.6.2. Punto de entrada al programa: `main` vs `_start`

- El punto de entrada al programa ha de ser nominado mediante la etiqueta `_start` ó `main`.
 - Directiva **global**: el punto de entrada se define en el módulo fuente y se declara como global, es decir tiene que ser accesible por otros programas. El punto de entrada tiene que ser accesible por el *linker* que lo declara como un símbolo definido externamente.



Si no se especifica la opción `-nostartfiles` del compilador `gcc`, la etiqueta de entrada al

programa ensamblador ha de ser **main**. Si se especifica la opción `-nostartfiles` entonces la entrada al programa será **_start**

- Compilación si el punto de entrada es **_start**: `gcc -nostartfiles -g -m32 -o sum1toN sum1toN.s`
- Compilación si el punto de entrada es **main**: `gcc -g -m32 -o sum1toN sum1toN.s`

28.6.3. Fases de la compilación



Figure 84. Proceso de Compilación

- Parar la compilación en la 1^a fase: preprocessamiento: `gcc -E sum1toN.c -o sum1toN.i`
 - **.i:* Salida del preprocessador: 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

memoria principal y ejecutado por la CPU.

- `gcc -m32 --save-tempo -o sum1toN sum1toN.c`
 - `--save-tempo`: gcc genera (save) los 3 ficheros parciales (tempo) 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.

28.6.4. 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

28.6.5. 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`

28.7. Depuración

- El debugger que utilizamos es el programa *gdb* de GBU : Gnu DeBugger
- [Manual GDB](#)
- Para poder depurar un programa hay que compilarlo con la opción de depuración **-g**: `gcc -g -m32 -o sum1toN sum1toN.s`
 - comprobar con `file sum1toN` que contiene los símbolos para la depuración
- Ejecutar *gdb* → se abre la ventana de comandos del depurador . El prompt es (gdb)

```
(gdb) file sum1toN -> reading symbols from binary program
(gdb) breakpoint main -> añade un punto de ruptura donde se detendrá la ejecución
(gdb) run -> ejecución
(gdb) layout src ó Control-x Control-a -> se añade la ventana con el código fuente
(gdb) focus src ó Control-x o -> se cambia de ventana donde están activadas las flechas del teclado para poder navegar.
```

- Las expresiones que emplean los comandos están en el lenguaje de programación de C

```
print &sum -> imprime la dirección de la variable sum
print sum -> imprime el contenido de la variable sum
print *sum_pointer -> imprime el contenido del objeto al que apunta el puntero
sum_pointer
```

28.8. Errores Comunes

28.8.1. 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`)

28.8.2. gdb

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

28.9. Programar y Depurar desde cero

28.9.1. Empezando: ASM

herramientas

- `gdb --version`
- `gcc --version`

Módulo fuente ASM

- `sum1toN.s`

```
### Programa: sum1toN.s
### Descripción: realiza la suma de la serie 1,2,3,...5
### 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
# ejecutable.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
```

Compilación

- Con `_start`: `gcc -nostartfiles -g -m32 -o sum1toN sum1toN.s`
- Con `main`: `gcc -g -m32 -o sum1toN sum1toN.s`

Propiedades Módulo Binario

- Módulo Binario: `file sum1toN`

Ejecución

- `./sum1toN`
- `echo $?`

Depurando

- Depurador `gdb`

```
file sum1toN
layout src ó Control-x Control-a
break main ó b main
run ó r
rTAB    -> autocompletado
help r  -> ¿ r es run?
print n ó p n
print /x n
p /t n
p /o n
next ó n
n 6
continue ó c
start ó s
n 6
p $ecx

set $ecx=-1
p $ecx
info regs
layout regs
layout src

set var n=10
set var &n=10
set var {int}&n=10
set var {int *}&n=10

set var {char[10]}&n="Hola"
p /s n

set $pc=&main

quit
```

```
info sources
shell ls
```

```
help layout
layout src
layout asm
layout split
help p
    formatos de print: /t /x /o /s /a
```

28.9.2. Empezando: C

- sum1toN.c

```
/*
Programa: sum1toN.c

gcc -g -m32 -o sum1toN sum1toN.c
file sum1toN
./sum1toN
echo $?

// Declaración de la función exit()
#include <stdlib.h>

// Módulo Principal
int main (void) {
    //Declaración de variables locales e inicialización de los parámetros del bucle
    int sum=0,n=5;
    //Bucle que genera los sumandos y realiza la suma
    while(n>0){    //Condición de salida del bucle cuando el sumando es negativo
        sum+=n;
        n--;          //Actualización del sumando
    }
    exit (sum);
}      //exit: finaliza la ejecución del programa y devuelve el argumento sum al
S.operativo.
```

- Compilación: `gcc -g -m32 -o sum1toN sum1toN.c`
- Módulo Binario: `file sum1toN`
- Depurador `gdb`

```
file sum1toN
layout src
break main
run
next
continue
start
```

```
next 6
print sum
quit
```

```
file sum1toN -> symbols table confirmation
layout src
breakpoint main ó b main
r ó run
h l ó help list
n ó next -> ejecutar siguiente línea del módulo fuente
n 6 -> ejecuta 6 veces n
c ó continue
s ó start
q ó quit
```

```
ptype sum
whatis sum
p /d sum
p /x sum
p /t sum
p /o sum
p /a &sum
p sum
p &sum
set var sum=22
p sum
set var {int}&sum=-3
p /x sum
```

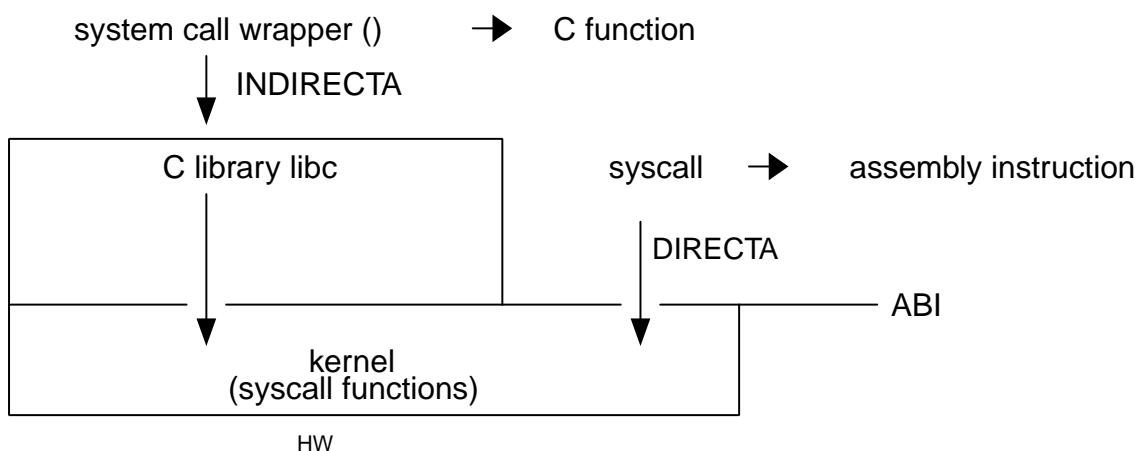
+

```
info sources
shell ls
help layout
layout src
layout asm
layout split
```

Chapter 29. Llamadas al Sistema Operativo

29.1. 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-  
R8-R9,syscall] -> kernel syscall write  
* API      ->      wrapper function           -> ABI  
-> kernel syscall
```

29.2. 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.

29.3. 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

29.4. 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'
```

29.4.1. 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

29.4.2. 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

29.5. 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)`

29.6. 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

```

29.7. 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

```

29.8. Línea de Comandos

29.8.1. 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 parametro 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, entrecomillarlo 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;
}
```

29.8.2. 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 25. 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 |

29.8.3. 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_args.s*

```
### gcc imprimir_args
### ./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

```

29.8.4. 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/as_tutorial/algoritmos_x86-32/basicos/sum_input"
## p /s *(char**)(%rsp+8) -> 0xfffffd26e
"/home/candido/tutoriales/as_tutorial/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
contiene el pointer al argumento string

```

```

        mov 8(%rsp),%rbx      #ebx tiene el contendio de la pila= dirección del
string
        xor %rcx,%rcx
        movb (%rbx),%cl      #caracter ASCII

## string argument pointers
        mov 16(%rsp),%rax     #eax tiene el contendio de la pila= dirección
del string. argv[2]
        mov 24(%rsp),%rbx     #eax tiene el contendio de la pila= dirección
del string. argv[3]
## fetch string indirect
## convert ascii numbers to values
        xor %rcx,%rcx
        xor %rdx,%rdx
        movb (%rax),%cl      # indirección para acceder al string
referenciado por argv[1]
        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:
## epilogo

```

```
mov %rbp,%rsp      # frame anterior
pop %rbp
ret                # recuperar dirección de retorno
```

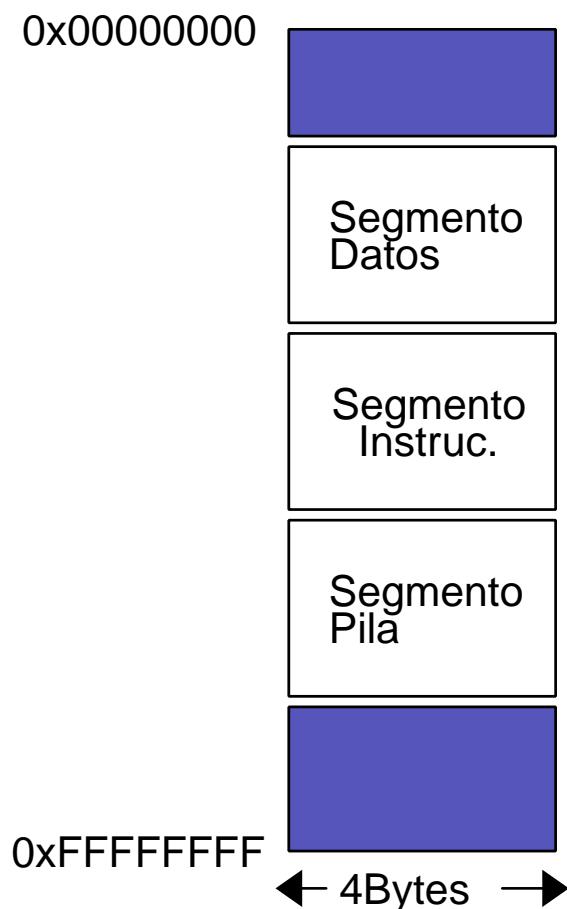
2. maximum_linea_com.s

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

Chapter 30. Pila

30.1. Concepto

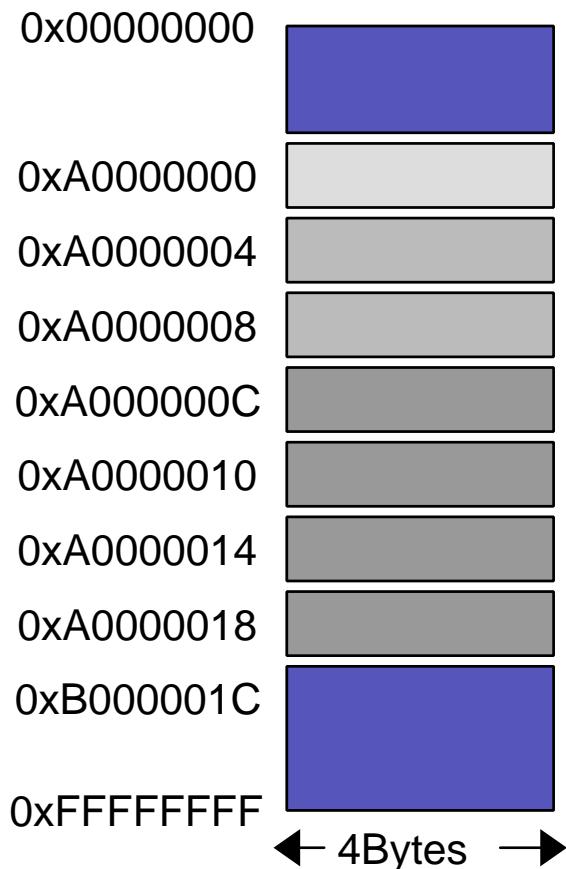
- Stack ó Pila:
 - Estructura de Datos Last Input First Output (LIFO)
- Memoria Externa
- Dirección de apilamiento: En sentido de direcciones de memoria DECRECIENTE.
- Un programa está estructurado en segmentos: Segmento datos, Segmento instrucciones, Segmento pila,
...
- Memoria Principal Segmentada:



30.2. 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

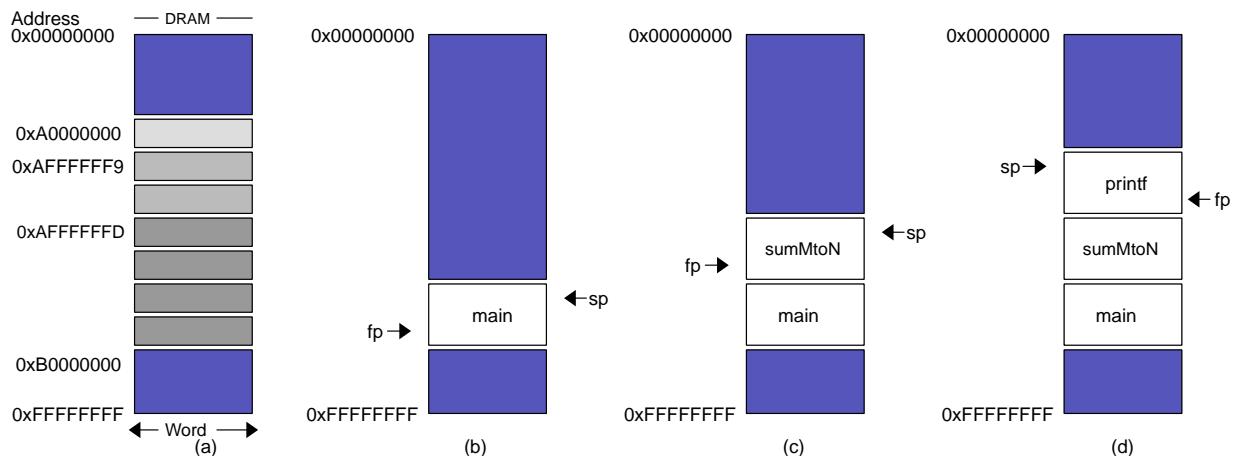
- en 8.
- Si el dato a apilar es menor que la anchura de la pila será necesario extenderlo. El tipo de extensión dependerá del tipo de dato (entero con signo, etc)
 - Segmento Pila de la arquitectura i386:



30.3. 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)



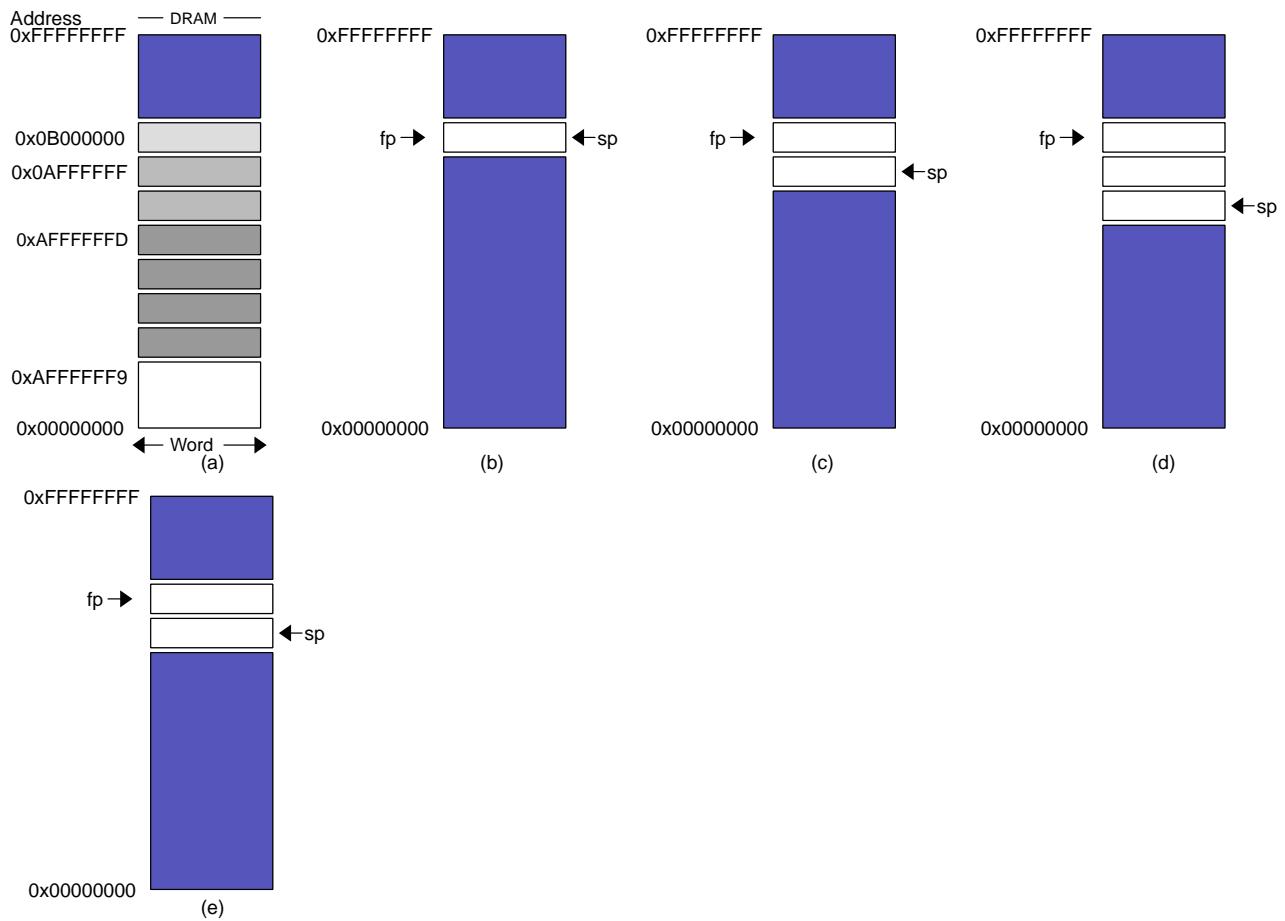
- (a) La pila no esta formada
- (b) Llamada a main: se forma el frame de main. El frame crece y decrece según apilamos y extraemos
- (c) Llamada de main a sumMtoN: el frame sumMtoN se forma sobre el anterior de main: nuevos punteros FP y SP.
- (d) Llamada de sumMtoN a printf: el frame printf se forma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

30.4. 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 . $SP \leftarrow SP - 1 * WordSize$ 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. $SP \leftarrow SP + 1 * WordSize$



- (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

30.4.1. Anidamiento de llamadas

- TODO

Chapter 31. Lenguaje de Programación C

31.1. 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.

31.2. Casting

31.2.1. Concepto

- Sintaxis

```
(type_name) expression
```

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

31.2.2. 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 = 8.0000/5 = 1.6000

31.3. Puntero

31.3.1. Referencias

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

31.3.2. 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.

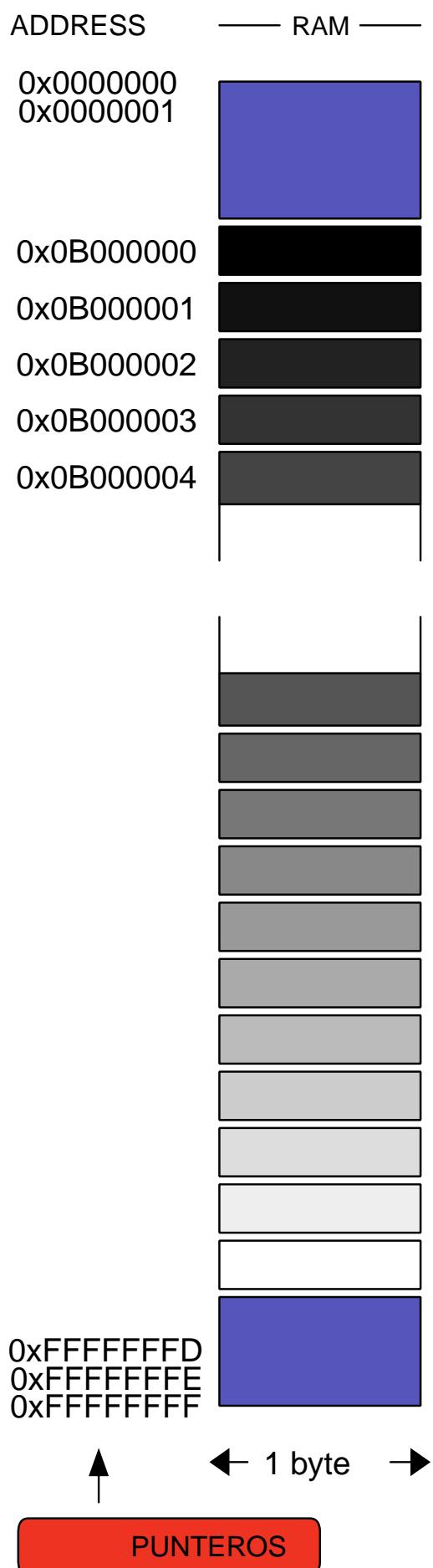


Para el aprendizaje de: conceptos relacionados con los punteros, su sintaxis, su aplicación, etc..., es necesaria la ejecución de los programas en modo PASO A PASO para poder visualizar los contenidos y referencias de los objetos en memoria. Utilizaremos el debugger

31.3.3. 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

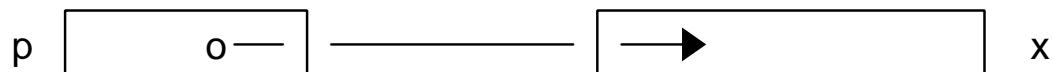
En cambio una VARIABLE PUNTERO:

- 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*

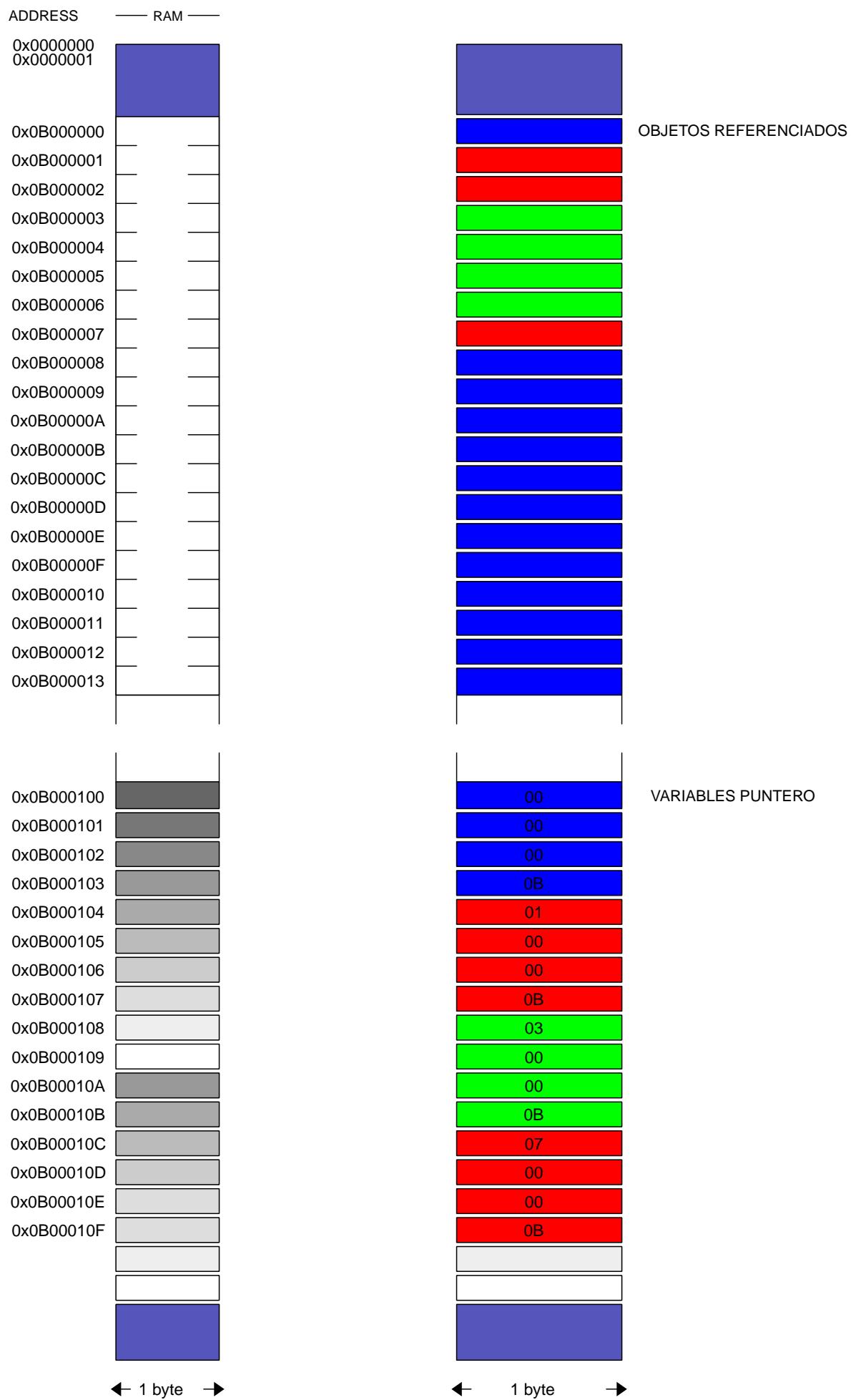


p : identificador de la variable puntero

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.

31.3.4. 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);
}

```

31.3.5. 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.

31.3.6. Operador Dirección

Símbolo &

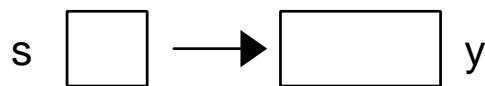
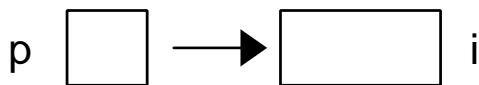
```

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.



31.3.7. 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);
```

31.3.8. 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.

31.3.9. 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

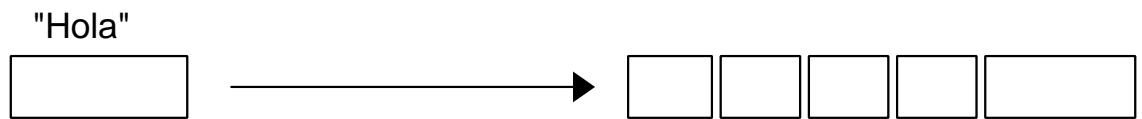
- Concepto
 - Un array es un puntero y una lista de elementos. El puntero apunta al primer elemento de la lista.
 - Cuando se crea un array se crean dos objetos
 - Los elementos del array cuya asignación de memoria es contigua
 - El puntero que apunta al primer elemento del array
- Ejemplo
 - Array de 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

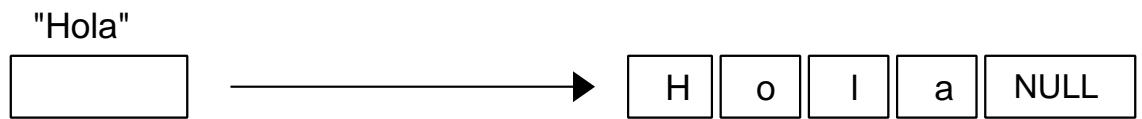
- Indexación: primer elemento MÁS la posición elemento i
 - $data_items + i$
- Modificar las expresiones de referencia a los elementos del array por expresiones aritmética de punteros

31.3.10. String Literal

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

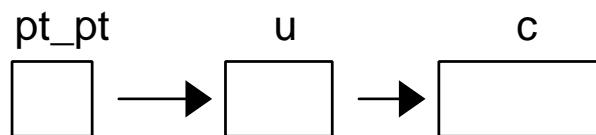


- Inicializar Array con el String *Hola*



- 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` ?

31.3.11. Puntero a Puntero



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

31.3.12. 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`

31.3.13. 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 32. FPU x87

32.1. FPU x87

32.1.1. 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.

32.1.2. Refs

- Programming With the x87 Floating-Point Unit: Intel Vol. 1 8-1
- Computer Systems: A Programmer's Perspective, 2/E (CS:APP2e) Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University

Chapter 33. Exámenes de Cursos Anteriores

33.1. Año 2018

33.1.1. 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 respuestas 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.

33.2. 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 al 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

buffer: # Reserva memoria para el buffer sin inicializar.

## INSTRUCCIONES
## Punto de entrada

_start:
## iniciozo el argumento de salida con el valor cero

## iniciozo la variable puntero EAX

## iniciozo el bucle con el número de iteraciones. 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 34. Miaulario: Videoconferencia

34.1. 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

34.2. Instalación de Zoom

- El conferenciente 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`

34.3. Guía de usuario Zoom

34.3.1. Configuración

- Testear el audio

34.4. 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](#) 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
- [UPC] [El ensamblador... pero si es muy fácil](#): IA-32 (i386) (sintaxis AT&T)
- [ATT] [Oracle AT&T language](#)
- [WikiBook] [Apuntes WikiBook:x86 Assembly: AT&T](#)
- [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](#)
- [NASM_bristol] [Apuntes Bristol Community College: Prog. NASM](#)
- [i386] [i386 \(32 bits\)](#)
- [A64] [amd64 \(64 bits\): recomendado](#)
- [IAL] [Intel asm language](#): Intel oficial Vol 1 Basic Architecture
- [AMD] [AMD oficial](#). Vol 3. 2.3 Summary of Registers and Data Types
- [paul_carter] [Paul Carter PC Assembly Language. Acceso libre. 2006.](#): Netwide Assembler NASM, Intel language
- [j_duntemann] [Jeff Duntemann](#). Assembly Language Step-by-Step: Programming with Linux. Wiley Ed. 3rd Edition. 2009.
- [irvine] [Kip R. Irvine](#). Assembly Language for x86 Processors. Pearson. 6th Edition. 2014.

Documentos de Programación de Bajo Nivel

- [ABI_i386] [ABI i386](#)
- [ms_llamada] [Convención de llamada MicroSoft](#)

Lenguaje de Programación C

- [c_king] [K.N.King](#) C programming, a Modern Approach W.W. Norton 2^aEd. 2008.

Herramientas de Desarrollo de Programas

- [I386][AS i386](#): syntax, mnemonics, register
- [GNU] [GNU Software Development](#)
- [GAS] [GNU ASsembler](#)
- [GDB] [debugger GDB](#)
- [GCC] [Compilador GCC](#)
- [CPP] [Preprocessor cpp](#)
- [BiU] [herramientas GNU binutils:as,ld,objdump,...](#)
- [VIM] [Vim](#)
- [EMACS] [Emacs](#)

Artículos

- [jvn] [linuxvoice](#)
- [w_uni] [wisconsin university RTL](#)

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.