

TEMA 1

1. Unidades funcionales

Las unidades funcionales según una arquitectura von Neumann distinguen 5 componentes: E/S, M y CPU (ALU + UC).

- Entrada: codifica/digitaliza/transmite (*lectura*).
- Memoria: almacena datos, programas, resultados de operaciones...
- CPU: unidad de procesamiento central que procesa información de E/S ejecutando el programa. La ALU es la unidad aritmético-lógica y se encarga de las operaciones y la UC, unidad de control, controla los circuitos.
- Salida: codifica/almacena/transmite (*escritura*).

La memoria almacena instrucciones y datos. Las instrucciones máquina se dividen en tres tipos:

- Transferencia (mov, in, out) M, E/S
- Operaciones (add, and) ALU
- Control (jmp, call, ret, set) UC

Un **programa almacenado** determina el comportamiento máquina porque contiene instrucciones reproducibles y un flujo de programa predeterminado.

En memoria, todos son datos que se interpretan como programa (codops) si son leídos en etapa de captación o que compilan o desensamblan si es código usado como datos. Hay varios tipos de codificación:

- Instrucciones: codops (codificación en bloque, por extensión, según fabricante).
- Enteros: binario (complemento a dos), BCD...
- Alfabéticos: ASCII, EBCDIC...
- Punto flotante: IEEE-754 simple/doble precisión...

El **lenguaje máquina** es el único que entienden los circuitos del computador (CPU). Las instrucciones se forman por bits agrupados en campos:

- Campo de código de operación: indica la operación correspondiente a la instrucción.
- Campos de dirección: especifican los lugares (o posición) donde se encuentra o donde ubicar los datos con los que se opera.

E/S

La **entrada** codifica la información del operador → M / CPU, recupera información previamente almacenada y comunica ordenadores entre sí. La **salida** codifica la información del resultado → operador humano, la almacena para uso posterior y comunica con otros computadores. Existen muchos dispositivos duales E/S.

MEMORIA

Almacenamiento primario (memoria semiconductora):

- Palabras n bits accesibles en 1 operación básica R/W:
 - Longitudes palabra típicas: 16-64bits.
 - Muy frecuente: memoria de bytes (asuntos alineamiento, ordenamiento).
- Accesible aleatoriamente (RAM) por dirección (posición):
 - Bus direcciones, bus datos, bus control (R/W), T_{acceso} .
 - Tamaños memoria típicos (PC): 8GB...32GB.
 - Tiempos acceso típicos: ~ns (DDR4-2400 19.2GB/s CL15 Lat 12.5ns) » DDR4-2400 → $F_{bus}=1200\text{MHz}$, $T_{cyc}=0.833\text{ns}$, $\text{Lat}_{CAS} \text{ CL15} \rightarrow 12.5\text{ns}$.
 - Jerarquía memoria: cache L1, L2 (on-chip), L3, MP.
 - Programa almacenado en memoria principal.

Almacenamiento secundario (óptico/magn. E/S):

- No es memoria von-Neumann, es E/S.
- Fichero swap se considera como parte de la jerarquía memoria.

CPU

ALU

- Componente más rápido del computador (junto con UC).
- Registros: almacenamiento más rápido (más que L1):
 - Operandos/Resultado de/a memoria/registros.
 - Arquitecturas R/R, R/M, M/M.

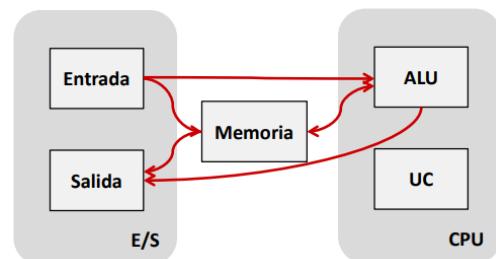
- Operaciones aritméticas (add, mul, div...):
 - Enteras y punto flotante.
- Operaciones lógicas (and, rol...):
 - Bit a bit.

UC

- Componente más rápido del computador (con ALU).
- Controla todos los demás circuitos (ALU, M, E/S):
 - Según lo indicado por el programa almacenado en MP.
 - Instrucciones transferencia → señales control M y E/S.
 - Instrucciones aritmético/lógicas → señales control ALU.
 - Temporización señales (dirección, datos, R/W).

Posibilidades funcionamiento:

- Programa E → MP
- Datos E → MP
- Ejecución programa: Datos → ALU → resultados
 - E / M → ALU → S / M
- Resultados → S
- Todo controlado según indique programa MP
 - Interpretado por la UC



SOBRE MEMORIA

Organización en bytes:

- ¿tamaño posición M = registro CPU (longitud palabra)?
 - ideal, pero no frecuente.
 - típicamente, posiciones 1B (direcccionamiento por bytes).
 - no necesidad empaquetamiento cadenas (strings).
 - Problemas: alineamiento, ordenamiento.

SOBRE MEMORIA DE BYTES

Ordenamiento en memoria de bytes:

- Criterio del extremo menor (little-endian): primero se almacena el byte menos significativo (LSB); en posición M más baja.
- Criterio del extremo mayor (big-endian): primero el MSB (en posición M más alta).

Alineamiento en memoria de bytes:

- Palabra de n bytes alineada ⇔ comienza en dirección múltiplo de n. Algunas CPUs requieren alineamiento accesos M (si no, bus error) y otras acceden más rápido si acceso alineado.
- Palabra no cruza línea de cache, página...

Clasificaciones m/n y pila-acumulador-RPG

- Tipos de CPU según operandos de las instrucciones ALU: también suele afectar a operandos instrucciones transferencia.
- Clasificación m/n: operaciones ALU admiten n operandos, máximo m de ellos de memoria.
- Combinaciones típicas:
 - Máquinas pila: 0/0 Repertorio: Push M, Pop M, Add, And...
 - Máquinas de acumulador: 1/1, operando implícito: registro acumulador A (más rápido que M). Repertorio: Load M, Store M, Add M, And M...
 - Máquinas de RPG (Registros de Propósito General): (x/2, x/3), múltiples “acumuladores”. Repertorio: Move R/M R/M, Add R/M R/M R/M.

RPG: Clasificación R/M

Para máquinas RPG

- Arquitecturas R/R (registro-registro): 0/2, 0/3. Add R1, R2, R3. Típico de RISC
- Arquitecturas R/M (registro-memoria): 1/2, 1/3 (2/3 poco frecuente). Add R1, R2 // Add R1, B // Add A, R2. Típico de CISC.

- Arquitecturas M/M (memoria-memoria): 2/2, 3/3(poco frecuente). $Add R1, R2 // Add R1, B // Add A, R2 // Add A, B$. Permite operar directamente en memoria, pero tiene demasiados accesos a memoria por instrucción máquina.

Sobre Repertorios

ISA

- Arquitectura del Repertorio (Instruction Set Architecture)
- Registros, Instrucciones, Modos de direccionamiento...

RISC

- Comput. repertorio reducido (Reduced Instruction Set Computer)
- 0/2, 0/3
- Pocas instrucciones, pocos modos, formato instrucción sencillo
- UC sencilla → muchos registros

CISC

- Comput. repertorio complejo (Complex Instruction Set Computer)
- 1/2, 1/3 (y resto)
- “más próximos a lenguajes alto nivel”
- Debate RISC/CISC agotado, diseños actuales mixtos

2. Conceptos básicos de funcionamiento

CICLO DE EJECUCIÓN

Programa en MP

CPU (UC) tiene PC (program counter):

- Posición MP de la siguiente instrucción.
- Captación: leer dicha posición $IR \leftarrow M[PC]$: usando MAR/MDR (Memory Address/Data Register), Instruction Register (IR). Se interpreta como codop y se incrementa PC.
- Decodificación: desglosar codop/operandos(regs). Posible etapa Operando(M): captar dato/ incrementar PC.
- Ejecución: llevar datos ALU / operar.
- Almacenamiento: salvar resultado regs / MP

Pensar tareas realizadas por UC para ejecutar instrucción.

Otras consideraciones

- Arquitectura M/M: varias captaciones operando. $PC++$, si las direcciones ocupan más posiciones M.
- Arquitectura R/M: cuando resultado en M (Add R0, A): acceso memoria adicional (Write): $M[MAR] \leftarrow MDR \leftarrow ALU_{out}$. UC activa señal Write
- Arquitectura R/R: varias instrucciones (Load A, R1 / Add R1, R0). Tiene como efecto colateral: R1 perdido, la ventaja: CPU más simple, veloz, pequeña (longitud/formato instrucción).
- Ciclo interrumpido por $IRQ \rightarrow ISR$; mecanismo subrutina/salvar contexto (PC/estado) salvo eso, comportamiento totalmente predeterminado por programa.
- CPU completa ($+L1+L2\dots+L3$) en 1 chip VLSI. La CPU nunca lee de memoria un dato aislado: lee de cache, si hay fallo, se trae un bloque entero se explica en clase así por motivos académicos.

Sobre formatos de instrucción

RISC: Pocas instrucciones, pocos modos, muchos registros, 0/2-0/3; formato instrucción sencillo, tal vez sólo 2-3: transferencia, ALU, ctrl. Ej: *formato instrucciones ALU de un RISC 32bits 128regs tipo 0/2*.

CISC: Muchas instrucciones y modos, menos registros, 1/2-1/3 (y resto); varios formatos de instrucción, distintas longitudes, codops long. var. también

Modos de direccionamiento

Un número acompañando a un codop puede significar muchas cosas, según el formato de instrucción, la instrucción concreta... Cada operando de la instrucción tiene su modo de direccionamiento.

- Inmediato: el número es el valor del operando.
- Registro: el número es un índice de registro.
- Memoria: la instrucción lleva índices de registro y/o desplazamiento (dirección en memoria). La dirección efectiva (EA) es la suma de todos ellos, el operando es $M[EA]$.

○ Directo	sólo dirección (disp)	$op=M[disp]$
○ Indirecto a través reg.	sólo registro (reg)	$op=M[reg]$
○ Relativo a base	registro y desplazamiento	$op=M[reg+disp]$
○ Indexado índice	(x escala) y dirección	$op=M[disp + index*scale]$
○ Combinado	todo	$op=M[disp+base+idx*sc]$

3. Estructuras de bus

Justificación buses (paralelos): E/S, M, CPU deben conectarse para pasar datos. Representación binaria/velocidad transferencia: palabras n bits M/ALU → bus datos n bits; direcciones m bits M → bus addr m bits; bus control para líneas UC (R/W...).

Bus único: La CPU escribe bus dirección y control R/W; también escribe bus datos si Write y puede haber señales IOR/W separadas de MemR/W. E/S/M comprueban si es su dirección, sólo en ese caso se conectan al bus de datos para evitar cortocircuito bus datos. Las ventajas son: sencillez, bajo coste, flexibilidad conexión, fácil añadir más dispositivos y la posibilidad de líneas de control de arbitraje para varios master.

Buses múltiples: típicamente: bus sistema (CPU-M) y bus E/S; también: múltiples buses E/S que separan dispositivos según velocidades, incluso: doble bus sistema; memoria datos/programa (arquitectura Harvard). Las ventajas: uno más rápido, ambos funcionan en paralelo y los inconvenientes: coste, complejidad.

Adaptación de velocidades

Velocidad componentes: CPU > Memoria >> E/S

Estados de espera: alargar ciclo bus si no se activa señal RDY (bus control) y permite conectar periféricos lentos a bus único.

Buffers/IRQ: dispositivo lento almacena datos en buffer rápido que evita retrasar CPU con estados de espera puesto que se dedica a otra tarea mientras tanto. Transferencia CPU a velocidad buffer (normal Memoria). Write: CPU escribe buffer, dispositivo genera IRQ al final. Read: CPU encarga lectura, dispositivo hace IRQ cuando listo.

DECODIFICACIÓN

Evitar cortocircuito bus datos: suponer por ejemplo que CPU es único dispositivo activo del bus, es decir, que puede escribir bus Addr. y Ctrl. cuando lo hace, se convierte en maestro del bus; luego veremos multiprocesadores, controladores DMA... varios activos requieren arbitraje para escoger maestro. Los demás dispositivos pasivos solo "escuchan" bus Addr, no pueden escribir, sólo leer; cuando la CPU les habla, se convierten en esclavos, es decir, se conectan al bus de datos y obedecen la orden R/W. #bits bus Addr. determina el "espacio de memoria".

Mapa de Memoria: dibujo de dónde está cada dispositivo en espacio Memoria. E/S puede ser "mapeada a memoria" o en espacio E/S separado.

Decodificación completa: se usan todos los bits Addr. MSB decodifican el dispositivo/módulo (CS). LSB direccionan dentro del dispositivo (Addr.).

Decodificación parcial: algunos (m) bits Addr. sin usar. El dispositivo aparece repetido 2^m veces en Memoria.

SOFTWARE DE SISTEMA

Cómo conseguir crear programa → MP → ejecutar

- Software de sistema implicado:
 - Shell (intérprete comandos): recibe órdenes usuario. EXEC: llamada para cargar y ejecutar aplicación.
 - Editor: permite crear código fuente (y archivar!).
 - Compilador / Enlazador: código objeto / ejecutable.
 - Sistema de ficheros (crear, copiar, abrir, leer): desde Shell / desde programa usuario.
 - Sistema E/S

Cómo se consigue encender → arrancar SO

- Soporte hardware: dirección de Bootstrap
- [Boot-P]ROM en espacio memoria apuntado
- Bootloader primario, carga arranque HD/FD/CD...
- Bootloader secundario (menú escoger SO, etc)...

Llamadas al sistema (ej: aplicación lee fichero/calcula/imprime):

- Usuario teclea nombre aplicación → EXEC: el shell invoca EXEC, proporcionando nombre fich.
- EXEC carga aplicación HD → M, pasa control: el propio SO proporciona zona M al cargar aplicación y EXEC retorna a aplicación, y ella retornará a Shell.

- Aplicación invoca OPEN/READ/CLOSE: proporciona zona memoria donde leer contenido.
- Aplicación calcula resultado, invoca PRINT/EXIT: proporciona datos a imprimir / código retorno.

SO gestiona recursos (especialmente multiuser/multitask)

- ej: solapar E/S final con carga siguiente tarea.
- ej: conmutar proceso en cuanto haga E/S.

Pensar tareas realizadas por SO para ejecutar aplicación

- Por ejemplo: leer datos HD, cálculos, imprimir resultados.
- Pensar entonces cómo solapar varias de esas aplicaciones

4. Rendimiento

- Medida definitiva: tiempo de ejecución del programa. El problema es ¿cuál programa? por lo que se acuerdan benchmarks que dependen de diseño CPU, repertorio instrucciones... y del compilador (benchmarks en lenguaje alto nivel) y versión SO, librerías...
- Ejemplo anterior: t5-t0 incluye HD, LPR. Tiempo transcurrido (wall-clock time) mide rendimiento sistema completo influido por prestaciones CPU, HD, LPR...
- Benchmarks CPU ejercitan sólo CPU. Tiempo de procesamiento (CPU time) influido por prestaciones CPU, M, caches, buses; la cache conserva lo accedido recientemente/más rápida, ventaja en ejecución bucles, p.ej.
- Reloj del procesador, la UC emplea varios ciclos de reloj en ejecutar una instrucción; pasos básicos 1 ciclo (conmutar señales control). Frecuencia $R = 1/P$. $500MHz = 1/2ns$ $1.25GHz = 1/0.8ns$
- Ecuación básica de rendimiento:

$$T = \frac{N * S}{R}$$

- T tiempo para ejecutar programa benchmark.

- N instrucciones (recuento dinámico bucles/subrutinas), no necesariamente igual a #instr. progr. objeto.

- S ciclos/instr. ("pasos básicos" de media).

Ideal: N y S ↓↓, R ↑↑

- N (instrucciones) depende de compilador(repertorio).

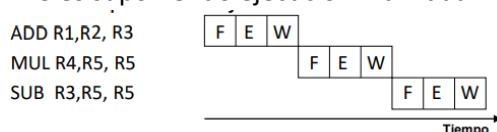
- S (ciclos/instr) depende de implementación CPU.

- R (MHz - GHz) depende de tecnología (y diseño CPU).

Alterar uno modifica los otros; aumentar R es a costa de aumentar S. Lo importante es que al final T↓

Segmentación de cauce (intenta que S≈1)

- NxS es suponiendo ejecución individual instrucciones

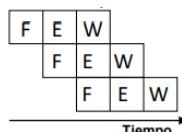


- Pero las distintas etapas hacen tareas distintas: UC puede tener circuitería separada para cada etapa:

– Fetch: captación

– Exec: ejecución

– Write: actualización registro



- una vez lleno el cauce, valor efectivo S=1 ciclo/instr: dependencias, competición, saltos y S≥1, S≈1.

Funcionamiento superescalar (que S<1)

Conseguir paralelismo a base de reduplicar UFs (unidades funcionales); combinado con segmentación, puede hacer S<1, se completa más de 1 instrucción por ciclo.

Común en CPUs actuales. Dificultades: emisión desordenada, corrección (mismo resultado que ejecución escalar).

Otras formas de reducir T

- Velocidad del reloj (R↑, S/R)

Tecnología ↑ ⇒ R↑; si no cambia nada más, Rx2 ⇒ T/2? ($T = NS / R$). Falso: Memoria también Rx2!!! o mejorar cache L1-L2.

Alternativamente, S↑ ⇒ R↑: "supersegmentación", reducir tarea por ciclo reloj, es difícil predecir ganancia, puede incluso empeorar.

- **Repertorio RISC/CISC (N-S)**

RISC: instr. simples para $R \uparrow \uparrow$, pero $S \downarrow \Rightarrow N \uparrow$

CISC: instr. complejas para $N \downarrow \downarrow$, pero $S \uparrow$, corregir $S \uparrow$ con segmentación \Rightarrow competición recursos

Actualmente técnicas híbridas RISC/CISC.

- **Compilador (N↓)**

Optimizador espacial ($N \downarrow$) o temporal ($N \times S \downarrow$) usualmente contrapuestos.

Espacial: requiere conocimiento arquitectura: repertorio, modos direccionamiento, alternativas traducción...

Temporal: requiere conocimiento detallado organización; reordenación instrucciones para ahorrar ciclos y evitar competición recursos.

La optimización debe ser correcta (mismo resultado).

Medida del rendimiento

Interesante para: diseñadores CPUs: evaluar mejoras introducidas; fabricantes: marketing y compradores: prestaciones/precio.

Benchmark: un único programa acordado, programas sintéticos no predicen bien T_{app} , programas reales muy específicos, colección programas considerados "frecuentes" (representativos), reducir a un único número usando media geométrica evita influencia computador referencia.

SPEC: System Performance Evaluation Corporation.

5. Perspectiva histórica

2ª Guerra Mundial

Tecnología relés electromagnéticos $T_{conmut.} = O(s)$. Previamente: engranajes, palancas, poleas. Tablas logaritmos, aprox. func. Trigonométricas. Generaciones 1-2-3-4ª 1945-55-65-75...

1ª Generación (45-55): tubos de vacío

Von Neumann: concepto prog. Almacenado, tubos vacío $100-1000 \times T_{conmut.} = O(ms)$ M: líneas retardo mercurio, núcleos magn. E/S: lect/perf. tarjetas, cintas magnéticas. Software: lenguaje máquina/ensamblador.

2ª Generación (55-65): transistores.

Invento Bell AT&T 1947 $T_{conmut.} = O(\mu s)$. E/S: procesadores E/S (cintas) en paralelo con CPU. Software: compilador FORTRAN.

3ª Generación (1965-75): Circuito Integrado

Velocidad CPU/M $\uparrow T_{conmut.} = O(ns)$. Arquitectura: μ Progr, segm.cauce, M cache. Software: SO multiusuario, memoria virtual.

4ª Generación (75-...): VLSI

μ Procesador: procesador completo en 1 chip; MP completa en uno o pocos chips: Intel, Motorola, AMD, TI, NS. Arquitectura: mejoras segm. cauce, cache, memoria virtual. Hardware: portátiles, PCs, WS, redes. Mainframes siguen sólo en grandes empresas.

Actualidad

Computadores sobremesa potentes/asequibles, internet, paralelismo masivo (Top500, MareNostrum, Magerit).

TEMA 2

CONCEPTOS BÁSICOS

1. Historia de los procesadores y arquitecturas de Intel

Procesadores Intel x86

Dominan el mercado portátil/sobremesa/servidor.

Diseño evolutivo: compatible ascendente hasta el 8086, introducido en 1978, va añadiendo características conforme pasa el tiempo. Computador con repertorio instrucciones complejo (CISC): muchas instrucciones diferentes, con muchos formatos distintos. Según como se cuenten, entre 2.034-3.683 instrucciones, pero sólo un pequeño subconjunto aparece en programas Linux.

(RISC) Computador con repertorio instrucciones reducido. RISC: muy pocas instrucciones, con muy pocos modos de direccionamiento que en principio funcionarían más rápido (reloj más rápido, segmentación...) aun así Intel gana en velocidad (no tanto en bajo consumo). Renacimiento RISC actual (p.ej. ARM, RISC-V), sobre todo bajo consumo.

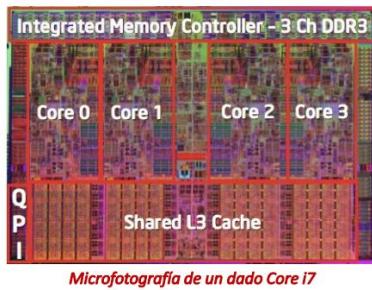
Evolución Intel x86: Hitos significativos

Nombre	Fecha	Transistores	MHz
■ 8086	1978	29K	5-10
			<ul style="list-style-type: none">▪ Primer procesador Intel 16-bit. Base para el IBM PC & MS-DOS▪ Espacio direccionamiento 1MB
■ 386	1985	275K	16-33
			<ul style="list-style-type: none">▪ Primer procesador Intel 32-bit de la familia (x86 luego llamada) IA32▪ Añadió "direcciónamiento plano"[†], capaz de arrancar Unix
■ Pentium 4E	2004	125M	2800-3800
			<ul style="list-style-type: none">▪ 1^{er} proc. Intel 64-bit de la familia (x86, llamada x86-64, EM64t) Intel 64
■ Core 2	2006	291M	1060-3500
			<ul style="list-style-type: none">▪ Primer procesador Intel multi-core
■ Core i7	2008	731M	1700-3900
			<ul style="list-style-type: none">▪ Cuatro cores, hyperthreading (2 vías)

Procesadores Intel x86: Visión general

■ Evolución de las máquinas

■ 386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2001	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M
■ Core i7 Skylake	2015	1.9B



■ Características añadidas

- Instrucciones de soporte para operación multimedia (ops. en paralelo)
- Instrucciones para posibilitar operaciones condicionales más eficientes
- Transición de 32 bits a 64 bits
- Más núcleos (cores)

Clones x86: Advanced Micro Devices (AMD)

Históricamente AMD ha ido siguiendo a Intel en todo pero las CPUs un poco más lentas, mucho más baratas. Y entonces reclutaron los mejores diseñadores de circuitos de Digital Equipment Corp. y otras compañías con tendencia descendente, construyeron el Opteron: duro competidor para el Pentium 4 y desarrollaron x86-64, su propia extensión a 64 bits. En años recientes Intel se reorganizó para ser más efectiva (1995-2011: líder mundial semiconductores, 2019: 2º detrás de Samsung) y AMD se ha quedado rezagada ya que externalizó su "fab" semiconductores (spin-off GlobalFoundries) aunque recientemente ha sacado nuevas CPUs competitivas (p.ej. Ryzen 3, 5, 7, 9, Threadripper).

La historia de los 64-bit de Intel

2001: Intel intenta un cambio radical de IA32 a IA64. Arquitectura totalmente diferente (Itanium), ejecuta código IA32 sólo como herencia y las prestaciones son decepcionantes.

2003: AMD interviene con una solución evolutiva x86-64 (ahora llamado "AMD64"). Intel se sintió obligada a concentrarse en IA64; difícil admitir error, o admitir que AMD es mejor.

2004: Intel anuncia extensión EM64T de la IA32 (ahora llamada Intel64). Extended Memory 64-bit Technology ↗
¡Casi idéntica a x86-64! Todos los procesadores x86 salvo gama baja soportan x86-64, pero gran cantidad de código se ejecuta aún en modo 32-bits

2. Lenguaje C, ensamblador, código máquina

Arquitectura: (también arquitectura del repertorio de instrucciones ISA): partes del diseño de un procesador que se necesitan entender para escribir código ensamblador. Ej: especificación del repertorio de instrucciones, registros.

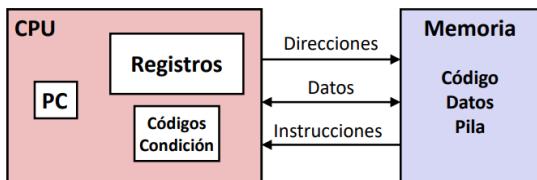
Formas del código: código máquina: programas (codops, bytes) que ejecuta el procesador. Código ensamblador: representación textual del código máquina.

Microarquitectura: Implementación de la arquitectura. Ej: tamaño de las caches y frecuencia de los cores.

Ejemplos de ISAs:

- Intel: (x86 =) IA32, Itanium (= IA64 = IPF), x86-64 (= Intel 64 = EM64t).
- ARM: Usado en casi todos los teléfonos móviles.
- RISC-V: Nueva ISA open-source.

Perspectiva Código Ensamblador/Máquina

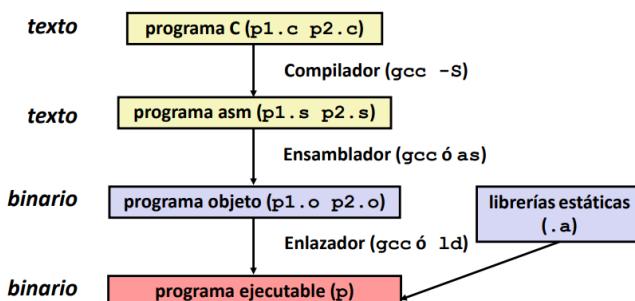


Estado visible al programador

- PC: Contador de programa: dirección de la próxima instrucción, llamado "RIP" (x86-64).
- Archivo de registros: datos del programa muy utilizados.
- Códigos de condición/flags de estado: almacenan información sobre la operación aritmética/lógica más reciente son usados para bifurcación condicional.
- Memoria: array direccionable por bytes, código y datos usuario y pila soporta procedimientos.

Convertir C en Código Objeto

- Código en ficheros p1.c p2.c
- Compilar con el comando: gcc -Og p1.c p2.c -o p
Usar optimizaciones básicas (-Og) [versiones recientes de GCC†].
Poner binario resultante en fichero p.



Compilar a ensamblador

Código C (sum.c)

```
long plus(long x, long y);  
  
void sumstore(long x, long y,  
              long *dest)  
{  
    long t = plus(x, y);  
    *dest = t;  
}
```

Ensamblador x86-64 generado[†]

```
sumstore:  
    pushq  %rbx  
    movq   %rdx, %rbx  
    call   plus  
    movq   %rax, (%rbx)  
    popq   %rbx  
    ret
```

Se obtiene con el comando gcc -Og -S sum.c y se produce el fichero sum.s

Representación Datos C, IA32, x86-64

Tamaño de Objetos C (en Bytes)

Tipo de Datos C	Normal 32-bit	Intel IA32	x86-64
▪ unsigned	4	4	4
▪ int	4	4	4
▪ long int	4	4	8
▪ char	1	1	1
▪ short	2	2	2
▪ float	4	4	4
▪ double	8	8	8
▪ long double	8	10/12	16
▪ char *	4	4	8

– o cualquier otro puntero

Características ensamblador

TIPOS DE DATOS

- Datos “enteros” de 1, 2, 4 u 8 bytes; valores de datos y direcciones (punteros sin tipo).
- Datos en punto flotante de 4, 8 ó 10 bytes.
- Código: secuencias de bytes codificando serie de instrucciones.
- No hay tipos compuestos como arrays o estructuras, tan sólo bytes ubicados contiguamente (uno tras otro) en memoria.

INSTRUCCIONES

- Realizan función aritmética sobre datos en registros o memoria. “Operaciones” = Instrucciones aritmético/lógicas.
- Transfieren datos entre memoria y registros: cargar datos de memoria a un registro y almacenar datos de un registro en memoria. “Instrucciones de transferencia”.
- Transferencia de control: Incondicionales: saltos, llamadas a procedimientos, retornos desde procesos. Saltos condicionales: “Instrucciones de control”.

Código Objeto

Ensamblador:

- Traduce .s pasándolo a .o
- Instrucciones codificadas en binario.
- Imagen casi completa del código ejecutable.
- Le faltan enlaces entre código de ficheros diferentes.

Enlazador:

- Resuelve referencias entre ficheros.
- Combina con librerías de tiempo de ejecución estáticas. Ej: código para malloc, printf.
- Algunas librerías son dinámicamente enlazadas, es decir, el enlace ocurre cuando el programa empieza a ejecutarse.

Ejemplo de Instrucción Máquina

*dest = t;

■ Código C

- Almacenar valor t adonde indica (apunta) dest

movq %rax, (%rbx)

■ Ensamblador

- Mover un valor de 8-byte a memoria
 - “Palabra Quad”[†] en jerga x86-64
- Operandos:
 - t: Registro %rax
 - dest: Registro %rbx
 - *dest: Memoria M[%rbx]

0x40059e: 48 89 03

■ Código Objeto

- Instrucción de 3-byte
- Almacenada en dir. 0x40059e

Desensamblar Código Objeto

Desensamblado

```
0000000000400595 <sumstore>:  
400595: 53          push    %rbx  
400596: 48 89 d3    mov     %rdx,%rbx  
400599: e8 f2 ff ff ff  callq   400590 <plus>  
40059e: 48 89 03    mov     %rax,(%rbx)  
4005a1: 5b          pop     %rbx  
4005a2: c3          retq
```

Desensamblador objdump -d sum

Herramienta útil para examinar código objeto, analiza el patrón de bits de series de instrucciones y produce una versión aproximada del código ensamblador (correspondiente). Puede ejecutarse sobre el fichero a.out (ejecutable completo) o el .o.

Otra alternativa es desde el depurado gdb (gdb sum)

disasembale sumstore: desensamblar procedimiento. x/14xb sumstore: examinar 14 bytes a partir de sumstore. Se puede desensamblar cualquier cosa que se pueda interpretar como código ejecutable. El desensamblador examina bytes y reconstruye el fuente asm.

3. Conceptos básicos asm: Registros, operandos, move

Registros enteros x86-64

%rax	%eax	%ax	%al	%r8	%r8d	%r8w	%r8b
%rbx	%ebx			%r9	%r9d		
%rcx	%ecx			%r10	%r10d		
%rdx	%edx			%r11	%r11d		
%rsi	%esi	%si	%sil	%r12	%r12d		
%rdi	%edi			%r13	%r13d		
%rsp	%esp			%r14	%r14d		
%rbp	%ebp			%r15	%r15d		

Un poco de historia: registros IA32

propósito general	%eax	%ax	%ah	%al
	%ecx	%cx	%ch	%cl
	%edx	%dx	%dh	%dl
	%ebx	%bx	%bh	%bl
	%esi	%si		
	%edi	%di		
	%esp	%sp		
	%ebp	%bp		

registros virtuales 16-bit
(compatibilidad ascendente)

Mover Datos

movq Source, Dest.

Tipos de operandos:

- Inmediato:** Datos enteros constantes. *Ej:* \$0x400, \$-533. Como constante C, pero con prefijo '\$'. Codificado mediante 1, 2, ó 4 bytes
- Registro:** alguno de los 16 registros enteros. *Ej:* %rax, %r13. Pero %rsp reservado para uso especial. Otros tienen usos especiales con instrucciones particulares.
- Memoria:** 8 bytes consecutivos mem. en dirección dada por un registro. *Ej. más sencillo:* (%rax). Hay otros diversos "modos de direccionamiento".

%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp
%rbp
%rN

Combinaciones de Operandos movq

	Source	Dest	Src, Dest	Análogo C
movq	Imm [†]	Reg	movq \$0x4,%rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	temp2 = temp1;
	Mem	Reg	movq %rax,(%rdx)	*p = temp;
	Mem	Reg	movq (%rax),%rdx	temp = *p;

MODOS DIRECCIONAMIENTO A MEMORIA SENCILLOS

- Normal (R) Mem[Reg[R]]. El registro R indica la dirección de memoria! Exacto! Como seguir (desreferenciar) un puntero en C
movq (%rcx),%rax
- Desplazamiento D(R) Mem[Reg[R]+D]. El registro R indica el inicio de una región de memoria. La constante de desplazamiento D indica el offset
movq 8(%rbp),%rdx

MODOS DIRECCIONAMIENTO A MEMORIA COMPLETOS

- Forma más general D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]
 - D: "Desplazamiento" constante 1, 2, ó 4 bytes.
 - Rb: Registro base: Cualquiera de los 16 registros enteros
 - Ri: Registro índice: Cualquiera, excepto %rsp
 - S: Factor de escala: 1, 2, 4, u 8
 -
- Casos Especiales

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]

4. Operaciones aritméticas y lógicas

Instrucción para el Cálculo de Direcciones

leaq Src, Dest

- Src es cualquier expresión de modo direccionamiento (a memoria)
- Ajusta Dest a la dirección indicada por la expresión

Usos:

- Calcular direcciones sin hacer referencias a memoria. Ej., traducción de p = &x[i];
- Calcular expresiones aritméticas de la forma x + k*y k = 1, 2, 4 u 8

Alguna operaciones aritméticas

Instrucciones de dos operandos

Formato	Operación [†]
addq Src,Dest	Dest = Dest + Src
subq Src,Dest	Dest = Dest - Src
imulq Src,Dest	Dest = Dest * Src
salq Src,Dest	Dest = Dest << Src
sarq Src,Dest	Dest = Dest >> Src
shrq Src,Dest	Dest = Dest >> Src
xorq Src,Dest	Dest = Dest ^ Src
andq Src,Dest	Dest = Dest & Src
orq Src,Dest	Dest = Dest Src

También llamada shlq

Aritméticas

Lógicas

Cuidado con el orden de los argumentos. No se distingue entre enteros con/sin signo.

Instrucciones de un operando

Formato	Operación
incq Dest	Dest = Dest + 1
decq Dest	Dest = Dest - 1
negq Dest	Dest = - Dest
notq Dest	Dest = ~Dest

CONTROL

1. Control: códigos de condición

Estado del Procesador (x86-64, Parcial)

Información sobre el programa ejecutándose actualmente.

- Datos temporales (%rax, ...)
- Situación de la pila en tiempo de ejecución (%rsp)
- Situación actual del contador de programa (%rip)
- Estado de comparaciones recientes (CF, ZF, SF, OF)

Registros de propósito general

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

Códigos de Condición

AJUSTE IMPLÍCITO

Registros de un solo bit:

CF: Flag de Acarrero (p/sin signo)

SF: Flag de Signo (para operaciones con signo)

ZF: Flag de Cero

OF: Flag de Overflow (operaciones con signo)

Ajustados implícitamente por las operaciones aritméticas (interpretarlo como efecto colateral):

Ejemplo: addq Src,Dest \leftrightarrow t = a+b

CF a 1 sii sale acarreo del bit más significativo (desbord. op. sin signo)

ZF a 1 sii t == 0

SF a 1 sii t < 0 (como número con signo)

OF a 1 sii desbord. en complemento a dos (desbord. op. con signo)

(a>0 && b>0 && t=0) || (a<0 && b<0 && t>=0)

No afectados por la instrucción lea.

AJUSTE EXPLÍCITO

Compare

- cmpq Src2, Src1
- cmpq b,a equivale a restar a-b pero sin ajustar el destino

CF a 1 sii sale acarreo del MSB (c_n) (hacer caso cuando cmp sin signo)

ZF a 1 sii a == b

SF a 1 sii (a-b) < 0 (como número con signo)

OF a 1 sii overflow en complemento a dos (atender si cmp con signo) (a>0 && b0 && (a-b)>0)
definición overflow OF = (c_n ^ c_{n-1}) mientras que acarreo CF = c_n

Test

- testq Src2, Src1
- testq b,a equivale a hacer a&b pero sin ajustar el destino

ZF a 1 sii (a&b) == 0

SF a 1 sii (a&b) < 0

Ajusta los códigos de condición según el valor de Src1 & Src2, útil cuando uno de los operandos es una máscara. Para comprobar si un valor es 0, gcc usa testq, no cmpq (cmpq \$0, %rax testq %rax, %rax)

Consultando Códigos de Condición

Instrucciones SetCC Dest:

Ajustar el byte destino a 0/1 según el código de condición indicado con CC (combinación de flags deseada). Dst registro debe ser tamaño byte, Dst memoria sólo se modifica 1er LSByte.

SetCC	Condición	Descripción
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Sign (negativo)
setsns	~SF	Not Sign
setg	~(SF^OF)&~ZF	Greater (signo)
setge	~(SF^OF)	Greater or Equal (signo)
setl	(SF^OF)	Less (signo)
setle	(SF^OF) ZF	Less or Equal (signo)
seta	~CF&~ZF	Above (sin signo)
setb	CF	Below (sin signo)

Ajustar un byte suelto Dest según el código de condición.

Uno de los registros byte direccionables:

- No se alteran los restantes bytes.
- Típicamente se usa movzbl para terminar trabajo, las instrucciones de 32-bit también ponen los 32 MSB a 0.

2. Saltos condicionales

Instrucciones jCC

Saltar a otro lugar del código si se cumple el código de condición CC.

jCC	Condición	Descripción
jmp	1	Incondicional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Sign (negativo)
jns	~SF	Not Sign
jg	~(SF^OF)&~ZF	Greater (signo)
jge	~(SF^OF)	Greater or Equal (signo)
jl	(SF^OF)	Less (signo)
jle	(SF^OF) ZF	Less or Equal (signo)
ja	~CF&~ZF	Above (sin signo)
jb	CF	Below (sin signo)

Expresándolo con código Goto

C permite la sentencia goto, que salta a la posición indicada por la etiqueta.

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

Traducción en General Expresión Condicional (usando saltos)

Código C

Val = Test ? Then_Expr : Else_Expr;

Versión Goto

```
ntest = !Test;
if (ntest) goto Else;
val = Then_Expr;
goto Done;
Else:
    val = Else_Expr;
Done:
    . . .
```

Creates regions of code separated for the Then and Else expressions and only executes the appropriate one.

Usando Movimientos Condicionales

Instrucciones Movimiento Condicional.

Las instrucciones implementan: if (Test) Dest \leftarrow Src. En procesadores x86 posteriores a 1995 (Pentium Pro/II). GCC intenta utilizarlas, pero solo cuando sepa que es seguro.

¿Por qué?

Ramificaciones muy perjudiciales para flujo de instrucciones en cauces. El movimiento condicional no requiere transferencia de control.

Malos Casos para Movimientos Condicionales

Puesto que se calculan ambos valores.

- Cálculos costosos: $val = Test(x) ? Hard1(x) : Hard2(x)$; Sólo tiene sentido cuando son cálculos muy sencillos.
- Cálculos arriesgados: $val = p ? *p : 0$; Pueden tener efectos no deseables.
- Cálculos con efectos colaterales: $val = x > 0 ? x*=7 : x+=3;$ ' No deberían tener efectos colaterales.

Código C

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Versión Goto

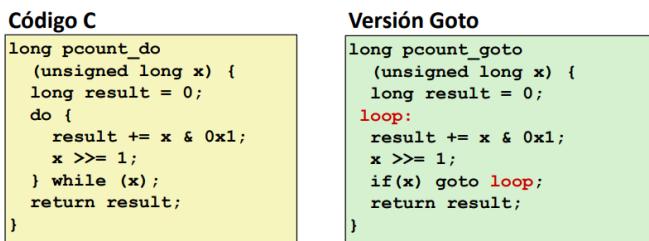
```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

3. Bucles

“Do-While”



Ejemplo

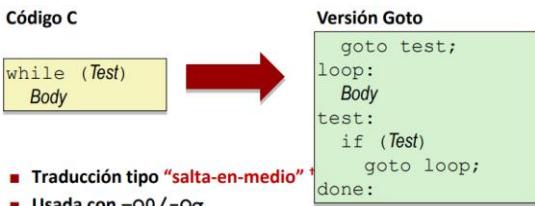


Contar número de 1's en el argumento x (“popcount”). Usar salto condicional para seguir iterando o salir del bucle.

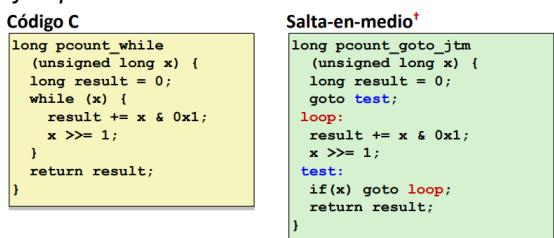
```
movl $0, %eax # result = 0
.L2:
  movq %rdi, %rdx
  andl $1, %edx # t = x & 0x1
  addq %rdx, %rax # result += t
  shrq %rdi # x >= 1
  jne .L2 # if (x) goto loop
  rep; ret
```

Registro	Uso(s)
%rdi	Argumento x
%rax	result

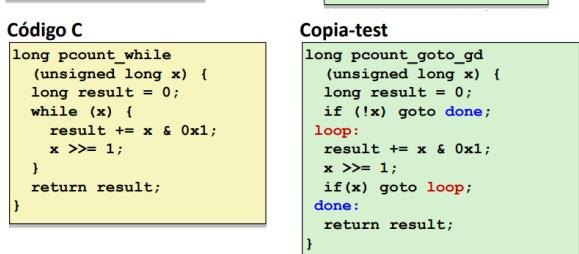
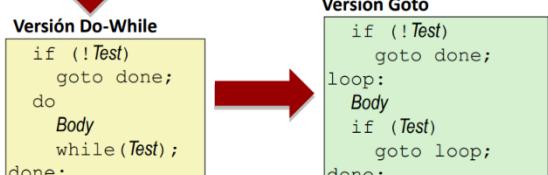
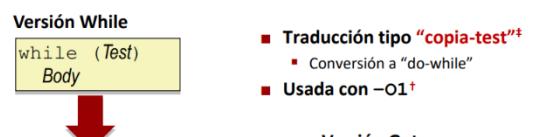
“While”



Ejemplo

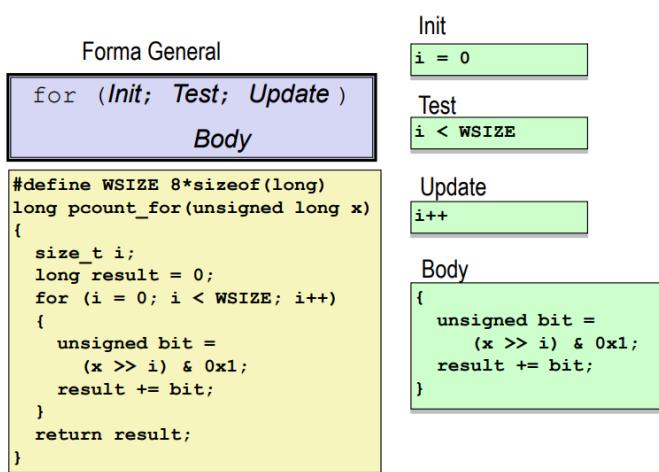


- Comparar con la versión do-while de la misma función
- El goto inicial empieza el bucle por **test** (“en medio”)



- Comparar con la versión do-while de la misma función
- El primer condicional guarda la entrada al bucle

"For"



Versión While

```
Init;
while (Test) {
    Body
    Update;
}
```

Bucle "For" → Bucle While

Versión For

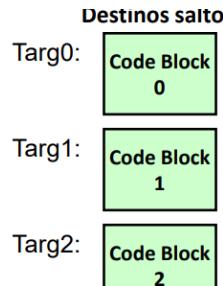
```
for (Init; Test; Update )
    Body
```



4. Sentencias switch

Forma switch

```
switch(x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    ...
    case val_n-1:
        Block n-1
}
```



Traducción aprox. (C ficticio, gcc supp. jump tables)

```
goto *JTab[x];
```

Estructura de la Tabla

- Cada destino salto requiere 8 bytes
- Dirección base es .L4

Saltos

- **Directo:** `jmp .L8`
- Destino salto indicado por etiqueta .L8
- **Indirecto:** `jmp * .L4(,%rdi,8)`
- Inicio de la tabla de saltos: .L4
- Se debe escalar por un factor de 8 (direcciones ocupan 8 bytes)
- Captar destino salto desde la Dirección Efectiva .L4 + x*8
 - Sólo para $0 \leq x \leq 6$

Tabla de saltos

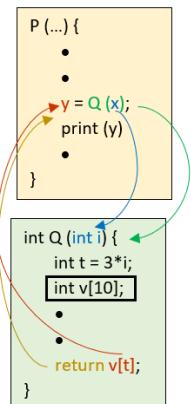
```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

PROCEDIMIENTOS

1. Mecanismos

- Transferencia de control: al principio del código del procedimiento y vuelta al punto de retorno.
- Transferencia de datos: argumentos del procedimiento y valor de retorno.
- Gestión de memoria: reservar durante ejecución procedimiento y liberar al retornar

Todos los mecanismos son implementados con instrucciones máquina. La implementación x86-64 de un proceso concreto usa sólo los mecanismos que éste requiera.



2. Estructura de la pila

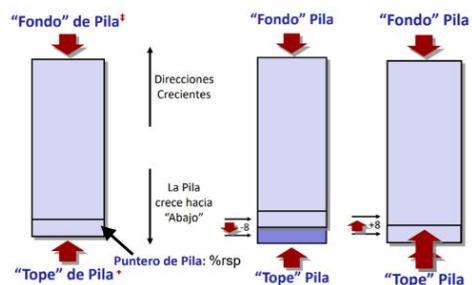
Región de memoria gestionada con disciplina de pila (LIFO), crece hacia posiciones inferiores, el registro %rsp contiene la dirección más baja de la pila ("tope").

Push

pushq Src: captura el operando en Src, decremente %rsp en 8 y escribe operando en la dirección indicada por %rsp

Pop

popq Dest: lee valor de dirección indicada por %rsp, incrementa %rsp en 8 y almacena el valor en Dest



3. Convenciones de llamada: Pasando el control

Usar la pila para soportar llamadas y retornos de procedimientos.

- Llamada a procedimiento: *call label*: recuerda la dirección de retorno en la pila, salta a etiqueta label y codificada con direccionamiento relativo a IP.
- Dirección de retorno: dirección de la siguiente instrucción justo después de la llamada (call).
- Retorno de procedimiento: ret. Recupera la dirección (de retorno) de la pila y salta a dicha dirección.

4. Convenciones de llamada: Pasando los datos

Registros

Pila

■ Primeros 6 argumentos

%rdi
%rsi
%rdx
%rcx
%r8
%r9

...
Arg n
...
Arg 8
Arg 7

■ Valor de retorno

%rax

■ Sólo se reserva espacio en la pila cuando se necesita

5. Convenciones de llamada: Gestionando datos locales

Lenguajes basados en pila

Lenguajes que soportan recursividad: Ej: C, Pascal, Java, el código debe ser "Reentrant": múltiples instancias simultáneas de un mismo procedimiento. Se necesita algún lugar para guardar el estado de cada instancia: argumentos, variables locales y puntero (dirección) de retorno.

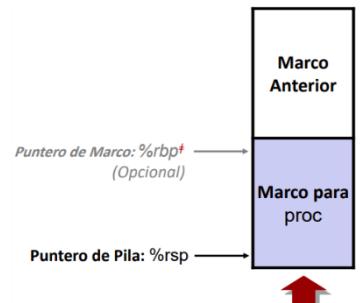
Disciplina de pila: estado para un procedimiento dado, necesario por tiempo limitado, desde que se le llama hasta que retorna. El invocado retorna antes de que lo haga el invocante.

La pila se reserva en Marcos: estado para una sola instancia de procedimiento

Marcos de Pila

Contenido: información de retorno, almacenamiento local (si necesario) y espacio temporal (si necesario).

Gestión: espacio se reserva al entrar el procedimiento; código de “Inicialización”, incluye el “push dir.ret.” de la instrucción *call*. Se libera al retornar; código de “Finalización” e incluye el “pop cont.prog.” de la instrucción *ret*.



Marco de Pila x86-64/Linux

Contenidos marco pila (de “Tope” a Fondo): “Confección de la lista de argumentos”: parámetros (7+) función a punto de ser llamada. Variables locales, si no se pueden mantener en registros . Contexto registros preservados. Antiguo puntero de marco (opcional); usando -fno-omit-frame-pointer (ó -O0). Dir.retorno, que pertenece al marco anterior.

Marco de pila del invocante: dirección de retorno salvada por la instrucción *call* y argumentos (7+) para esta llamada.

Convenciones de Preservación de Registros

Si cuando el procedimiento *yoo* (invocante) llama a *who* (invocado) usa registro para almacenamiento temporal (ej).

```
yoo:
    ...
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    ...
    ret
```

```
who:
    ...
    subq $18213, %rdx
    ...
    ret
```

El contenido del registro %rdx es sobreescrito por *who*, lo que podría causar problemas; por lo que se necesita alguna coordinación.

- “Salva Invocante”: el que llama salva valores temporales en su marco antes de la llamada.
- “Salva Invocado”: el llamado salva valores temporales en su marco antes de usar (regs.) y los restaura antes de retornar al que llama.

Uso de Registros en Linux x86-64

SALVA INVOCANTE

%rax: valor de retorno; puede ser modificado por el proceso.

%rdi, ..., %r9: argumentos; pueden ser modificados por el proceso.

%r10, %r11: temporales; pueden ser modificados por el proceso.

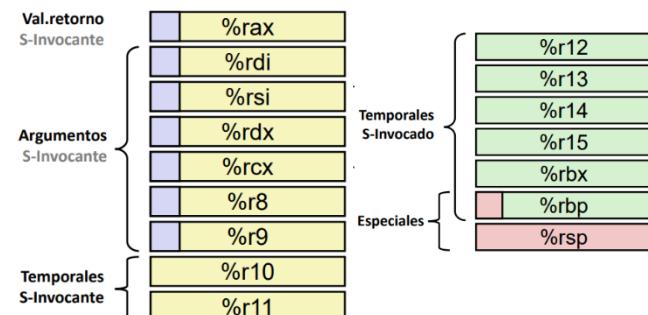
SALVA INVOCADO

%rbx, %r12 - %r15: invocado debe preservar y restaurar.

ESPECIALES

%rbp: es salva-invocado, por lo que debe preservar y restaurar, pero puede que se use como marco pila y usarse intermezcladamente.

%rsp: forma especial de salva-invocado ya que restaura su valor original a la salida del procedimiento.



6. Recursividad

OBSERVACIONES

Manejada sin especiales consideraciones: marcos pila implican que cada llamada a función tiene almacenamiento privado; variables locales y registros preservados, dirección de retorno salvada.

Convenciones preservación registros previenen que una llamada a función corrompa los datos de otra, a menos que el código C explícitamente lo haga (p.ej. buffer overflow).

Disciplina de pila sigue el patrón de llamadas/retornos; si P llama a Q, entonces Q retorna antes que P. LIFO

También funciona con recursividad mutua; P llama a Q o Q llama a P.

7. Resumen

La pila es la estructura de datos correcta para llamada/retorno procedimientos, si P llama a Q, entonces Q retorna antes que P.

Recursividad (y recursividad mutua) con mismas convenciones de llamada normales; se pueden almacenar valores tranquilamente en el marco de pila local y en registros salva-invocado, pone argumentos 7+ de la función en tope de pila y devuelve el resultado en %rax.

Los punteros son direcciones de valores, globales o en pila.

DATOS

1. Tipos de datos básicos

ENTEROS

Almacenados y manipulados en registros (enteros) de propósito general. Con/sin signo depende de las instrucciones usadas.

Intel	ASM [#]	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

PUNTO FLOTANTE

Almacenados y manipulados en registros punto flotante.

Intel	ASM	Bytes	C
Single	s	4	float
Double	d	8	double
Extended	t	10/12/16	long double

2. Arrays

T A[L];

Array de tipo T y longitud L. El identificador A (Tipo T*) puede usarse como puntero al elemento 0.

Multidimensionales (Anidados)

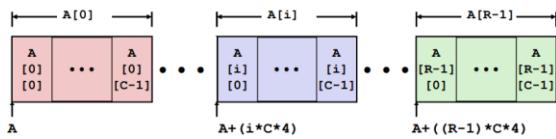
T A[R][C];

Array 2D de tipo T; con R filas y C columnas los elementos requieren K bytes y el tamaño total del array es de R*C*K bytes y se almacena por filas.

Acceso a filas

A[i] es un array de C elementos; cada elemento de tipo T requiere K bytes y la dirección de comienzo es A + i*(C*K).

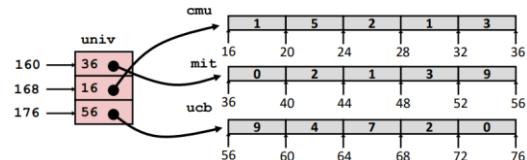
int A[R][C];



Acceso a elementos

A[i][j] es elemento de tipo T, que requiere K bytes. La dirección se calcula como A + i*(C*K)+j*K = A+(i*C + j)*K.

Multi-Nivel



Acceso a elementos

Acceso a elemento Mem[Mem[univ+8*index]+4*digit]. Debe hacer dos lecturas de memoria: primero obtener puntero al array fila y después acceder elemento dentro del array.

Matriz NxN

Dimensiones fijas: se conoce valor de N en tiempo de compilación. Dimensiones variables, indexado explícito: forma tradicional de implementar arrays dinámicos.

Dimensiones variables, indexado implícito: soportado ahora por gcc.

Acceso a elementos

- int A[n][n];
- Dirección A + i * (C * K) + j * K
- C = n, K = 4
- Hay que realizar multiplicación entera

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n], size_t i, size_t j)
{
    return A[i][j];
}
```

```
# n en %rdi, a en %rsi, i en %rdx, j en %rcx
imulq    %rdx, %rdi          # n*i
leaq    (%rsi,%rdi,4), %rax  # A + 4*n*i
movl    (%rax,%rcx,4), %eax  # A + 4*n*i + 4*j
ret
```

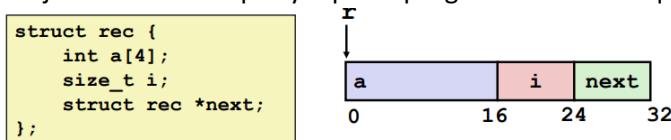
```
#define N 16
typedef int fix_matrix[N][N];
/* Get element A[i][j] */
int fix_ele(fix_matrix A,
            size_t i, size_t j)
{
    return A[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element A[i][j] */
int vec_ele(size_t n, int *A,
            size_t i, size_t j);
{
    return A[IDX(n,i,j)];
}
```

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n],
            size_t i, size_t j) {
    return A[i][j];
}
```

3. Estructuras

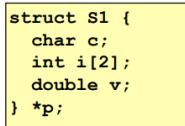
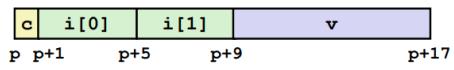
Se representan como bloque de memoria, suficientemente grande como para contener todos los campos; cuyos campos se refieren mediante sus nombres: struct.field, pointer->field. Se ordenan según la declaración, incluso si otro orden pudiera producir una representación más compacta y el compilador determina posición/tamaño conjunto de los campos ya que el programa a nivel máquina no entiende las estructuras del código fuente.



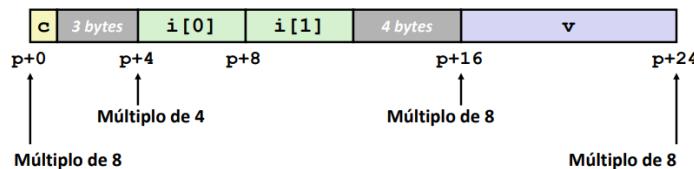
Acceso: puntero indica primer byte de la estructura y se accede a los elementos mediante sus desplazamientos.

Alineamientos

Datos desalineados



Datos alineados: el tipo de datos primitivo requiere K bytes y la dirección debe ser múltiplo de K.



PRINCIPIOS

El tipo de datos primitivo requiere K bytes, la dirección debe ser múltiplo de K, es requisito en algunas máquinas; recomendado en x86-64. Se deben alinear puesto que a la memoria se accede (físicamente) en trozos (alineados) de 4 u 8 bytes (dependiendo del sistema) y es ineficiente cargar o almacenar dato que cruza frontera quad Word, además de que la memoria virtual es muy delicada cuando un dato se extiende a 2 páginas. El compilador inserta huecos en estructura para asegurar el correcto alineamiento campos.

Dentro de la estructura deben cumplirse los requisitos de alineamiento de cada elemento, además cada estructura tiene un requisito de alineamiento K (mayor alineamiento de cualquier elemento); la dirección inicial y la longitud deben ser múltiplos de K. Si el requisito de alineamiento máximo es K; el struct debe ocupar globalmente un múltiplo de K.

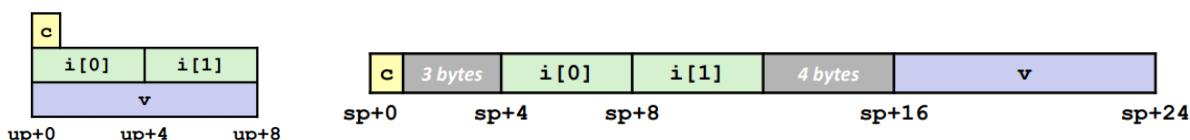
Array de estructuras

Para acceder a un elemento hay que calcular el desplazamiento del elemento del array y luego el dentro de la estructura.

Para ahorrar espacio hay que declarar primero los tipos de datos grandes.

4. Uniones

En una unión se reserva memoria de acuerdo al elemento más grande, aunque solo puede usarse un campo a la vez. Se ahorra espacio.



Ordenamiento de Bytes

Las palabras short/long/quad, son almacenadas en memoria como 2/4/8B consecutivos. ¿Cuál es el byte más (menos) significativo? Lo que puede causar problemas al intercambiar datos binarios entre máquinas.

Big Endian (extremo mayor): el byte más significativo está en la dirección más baja ("viene primero") Sparc .

Little Endian (extremo menor): el byte menos significativo está en la dirección más baja. Intel x86, ARM con Android e iOS.

Bi Endian : Se puede configurar de cualquiera de las dos formas. ARM

TEMA 3

La CPU está constituida por la unidad de procesamiento o camino de datos (“datapath”) y la unidad de control.

1. Unidad de procesamiento o camino de datos

La unidad de procesamiento comprende elementos hardware como:

- Unidades funcionales (ALU, desplazad., multiplic., etc.).
- Registros: de uso general (varios GPR), de estado, contador de programa (PC), puntero de pila (SP), instrucción (IR), de dato de memoria (MDR / MBR) y de dirección de memoria (MAR).
- Multiplexores.
- Buses internos.

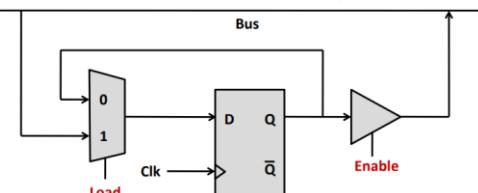
Unidad de procesamiento con un bus

- Componentes interconectados mediante bus común.
- MDR: dos entradas, dos salidas.
- MAR: unidireccional (procesador → memoria).
- R0...Rn-1: GPR, SP, índices...
- Y, Z, TEMP: registros transparentes al programador y almácen temporal interno en la ejecución de una instrucción.
- MUX: selecciona entrada A de la ALU: la constante 4 se usa para incrementar el contador de programa.
- La UC genera señales internas y externas (memoria).

Una instrucción puede ser ejecutada mediante una o más de las siguientes operaciones: transferir de un registro a otro, realizar operación aritmética o lógica y almacenar en registro, cargar posición de memoria en registro y almacenar registro en posición de memoria.

Transferir de un registro a otro: cada registro usa dos señales de control: *Load*: Carga en paralelo y

Enable: Habilitación de salida (buffer triestado).



Realizar operación aritmética o lógica y almacenar en registro: ALU: circuito combinacional sin memoria, el resultado es almacenado temporalmente en Z. Las señales de control de la ALU podrían estar codificadas.

Ej: R3 ← R1 + R2 1.Enable R1, Load Y 2.Enable R2, Select Y, Add, Load Z 3.Enable Z, Load R3.

CARGAR POSICIÓN DE MEMORIA EN REGISTRO

- Transferir dirección a MAR.
- Activar lectura de memoria.
- Almacenar dato leído en MDR (MDR dos entradas, dos salidas).
- La temporización interna debe coordinarse con la de la memoria: la lectura puede requerir varios ciclos de reloj y el procesador debe esperar la activación de señal de finalización de ciclo de memoria

Ej: Load (R1) → R2

1. Enable R1, Load MAR
2. Comenzar lectura
3. Esperar fin de ciclo de memoria, Load MDR desde memoria
4. Enable MDR hacia bus interno, Load R2.

ALMACENAR REGISTRO EN POSICIÓN DE MEMORIA

- Transferir dirección a MAR.
- Transferir dato a escribir a MDR (MDR dos entradas, dos salidas).
- Activar escritura de memoria.
- La temporización interna debe coordinarse con la de la memoria: la escritura puede requerir varios ciclos de reloj por lo que el procesador debe esperar la activación de señal de finalización de ciclo de memoria

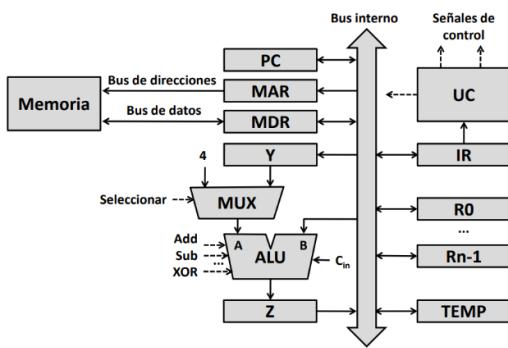
Ej: Store R2 → (R1) 1. Enable R1, Load MAR 2. Enable R2, Load MDR desde bus interno

3. Comenzar escritura
4. Esperar fin de ciclo de memoria

Ejecución de una instrucción completa

Ej.: Add (R3) → R1

1. Enable PC, Load MAR, Select 4, Sumar, Load Z
 2. Comenzar lectura, Enable Z, Load PC, Load Y*
 3. Esperar fin de ciclo de memoria, Load MDR desde mem.
 4. Enable MDR hacia bus interno, Load IR
 5. Decodificar instrucción
 6. Enable R3, Load MAR
 7. Comenzar lectura, Enable R1, Load Y
 8. Esperar fin de ciclo de memoria, Load MDR desde mem.
 9. Enable MDR hacia bus interno, Select Y, Sumar, Load Z
 10. Enable Z, Load R1, Saltar a captación
- } captación
- } ejecución



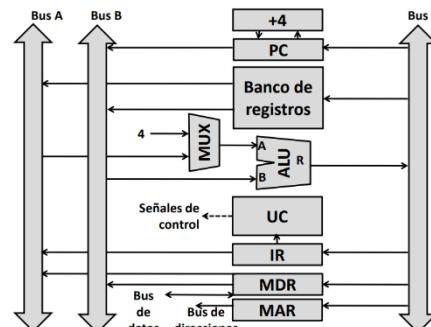
Unidad de procesamientos con buses múltiples

- Banco de registros con tres puertos: dos registros pueden poner sus contenidos en los buses A y B y un dato del bus C puede cargarse en un registro; en el mismo ciclo.
- La ALU; no necesita los registros Y y Z ya que puede pasar A o B directamente a R (bus C).
- Unidad de incremento (+4), pero la constante 4 como fuente de la ALU sigue siendo útil para incrementar otras direcciones en instrucciones de movimiento múltiple.

Ejecución de una instrucción completa

Ej.: R6 ← R4 + R5

1. Enable PC, R=B, Load MAR
 2. Comenzar lectura, Incrementar PC
 3. Esperar fin de ciclo de memoria, Load MDR desde mem.
 4. Enable MDR hacia B, R=B, Load IR
 5. Decodificar instrucción
 6. Enable R4 hacia A, Enable R5 hacia B, Select A, Sumar, Load R6, Saltar a captación
- } captación
- } ejecución



2. Unidad de control

La unidad de control interpreta y controla la ejecución de las instrucciones leídas de la memoria principal, en dos fases:

- Captación (y secuenciamiento) de las instrucciones: la UC lee de MP la instrucción apuntada por PC, $IR \leftarrow M[PC]^*$ y determina la dirección de la instrucción siguiente y la carga en PC
- Ejecución/interpretación de las instrucciones en IR: UC reconoce el tipo de instrucción, manda las señales necesarias para tomar los operandos necesarios y dirigirlos a las unidades funcionales adecuadas de la unidad de proceso, manda las señales necesarias para realizar la operación, manda las señales necesarias para enviar los resultados a su destino.

Señales de entrada a la UC: señal de reloj, instrucción actual (codop, campos de direccionamiento...), estado de la unidad de proceso y señales externas (por ej. interrupciones).

Señales de salida de la UC: son de dos tipos, las *señales que gobiernan la unidad de procesamiento*: carga de registros, incremento de registros, desplazamiento de registros, selección de entradas de multiplexores y selección de operaciones de la ALU ... Y las *señales externas*: lectura/escritura en memoria.

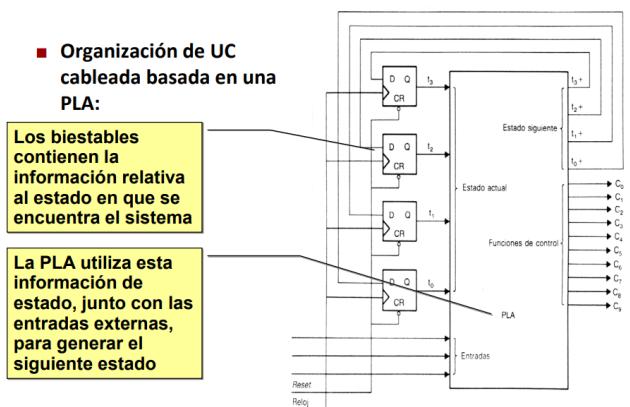
Hay dos tipos de unidades de control en función del diseño que siguen:

- **Control fijo o cableado ("hardwired")**: se emplean métodos de diseño de circuitos digitales secuenciales a partir de diagramas de estados; el circuito final se obtiene conectando componentes básicos como puertas y biestables, aunque más a menudo se usan PLA.
- **Control microprogramado**: todas las señales que se pueden activar simultáneamente se agrupan para formar palabras de control, que se almacenan en una memoria de control (normalmente ROM). Una instrucción de lenguaje máquina se transforma sistemáticamente en un programa (microprograma) almacenado en la memoria de control. Por lo que hay mayor facilidad de diseño para instrucciones complejas, lo que hace que sea el método estándar en la mayoría de los CISC.

UC Cableada

Se diseña mediante puertas lógicas y biestables siguiendo uno de los métodos clásicos de diseño de sistemas digitales secuenciales. El diseño es laborioso y difícil de modificar debido a la complejidad de los circuitos, pero suele ser más rápida que la microprogramada; se utilizan PLA (Programmable Logic Arrays) para la implementación y debido a las modernas técnicas de diseño y a RISC ha tomado nuevo auge la realización de UC cableadas.

Técnicas de diseño por computador (CAD) para circuitos VLSI (compiladores de silicio): resuelven automáticamente la mayor parte de las dificultades de diseño de lógica cablead y generan directamente las máscaras de fabricación de circuitos VLSI a partir de descripciones del comportamiento funcional del circuito en un lenguaje de alto nivel.



Ventajas: la minimización del esfuerzo de diseño, mayor flexibilidad y fiabilidad y ahorro de espacio y potencia.

EJEMPLO DE UC CABLEADA

Implementación de una unidad de control cableada sencilla (ODE) “Ordenador Didáctico Elemental”.

Pasos a seguir para llegar al diseño físico:

1. Definir una máquina de estados finitos.

Dado el diagrama de flujo de la UC de ODE, detallamos éste como un conjunto finito de estados y transiciones entre ellos.

2. Describir dicha máquina en un lenguaje de alto nivel.

El lenguaje concreto depende del programa que utilicemos para “compilar” la descripción de la máquina. Estos lenguajes tienen sentencias para definir: entradas y salidas y estados y transiciones condicionales e incondicionales entre estados.

3. Generar la tabla de verdad para la PLA.

Según la descripción que hayamos hecho de la máquina de estados podemos usar un programa que use el modelo Mealy (las salidas dependen de entradas y de estado presente) o uno que use el modelo Moore (salidas dependen exclusivamente de estado actual).

4. Minimizar la tabla de verdad.

Mediante un programa que utiliza algoritmos heurísticos rápidos.

5. Diseñar físicamente la PLA partiendo de la tabla de verdad.

Automáticamente: mediante un programa especial para diseño de layouts de PLA. Semiautomáticamente: diseñando mediante un programa de CAD de circuitos VLSI cada una de las celdas que, repetidas convenientemente, forman la PLA; Dando una especificación de cómo han de colocarse (tabla de verdad minimizada).

Esquema simplificado de la PLA usada para la UC de ODE: CMOS de dos fases de reloj: no hacen falta biestables de estado siguiente, $\Phi_1=1 \Rightarrow$ se leen las entradas y se precarga el plano AND, $\Phi_2=1 \Rightarrow$ se evalúa el plano AND y se precarga el plano OR y $\Phi_2=0 \Rightarrow$ se evalúa el plano OR.

UC microprogramada

Consiste en emplear una memoria (de control) para almacenar las señales de control de los períodos (ciclos) de cada instrucción.

Origen histórico: Maurice V. Wilkes (1913-2010) en 1951-1953 propone el siguiente esquema: dos memorias A y B, construidas con matrices de diodos; las señales de control se encuentran almacenadas en la memoria A y la memoria B contiene la dirección de la siguiente microinstrucción. Se permiten microbifurcaciones condicionales, mediante un biestable y un decodificador que selecciona entre dos direcciones de la matriz B.

Microinstrucción: cada palabra de la memoria de control

Microprograma: conjunto ordenado de microinstrucciones cuyas señales de control constituyen el cronograma de una (macro)instrucción del lenguaje máquina.

Ejecución de un microprograma: lectura en cada pulso de reloj de una de las microinstrucciones que lo forman, enviando las señales leídas a la unidad de proceso como señales de control.

Microcódigo: conjunto de los microprogramas de una máquina.

Ventajas

- Simplicidad conceptual, la información de control reside en una memoria.
- Se pueden incluir, sin dificultades, instrucciones complejas, de muchos ciclos de duración, el único límite es el tamaño de la memoria de control.
- Las correcciones, modificaciones y ampliaciones son mucho más fáciles de hacer que en una unidad de control cableada; no hay que rediseñar toda la unidad, sino sólo cambiar el contenido de algunas posiciones de la memoria de control.
- Permite construir computadores con varios juegos de instrucciones, cambiando el contenido de la memoria de control (si es RAM permite emular otros computadores).

Desventaja

- Lentitud frente a cableada, debido a una menor capacidad de expresar paralelismo de las microinstrucciones.

3. Control microprogramado

Formato de las microinstrucciones

Las señales de control que gobiernan un mismo elemento del datapath y se suelen agrupar en campos. Ejemplos: señales triestado que controlan el acceso a un bus, señales de operación de la ALU y señales de control de la memoria.

Formato no codificado: hay un bit para cada señal de control de un campo.

Formato codificado: para acortar el tamaño de las microinstrucciones se codifican todos o alguno de sus campos.

Inconveniente: hay que incluir decodificadores para extraer la información real y tiene menor capacidad de expresar paralelismo.

Solapamiento de campos: si sólo unas pocas señales de control están activas en cada ciclo, o existen con frecuencia señales excluyentes, que no se pueden activar simultáneamente se puede acortar la longitud de las microinstrucciones solapando campos. Inconvenientes: retardo introducido por el demultiplexor y hace incompatibles las operaciones de los campos solapados.

MICROPROGRAMACIÓN VERTICAL: mucha codificación, microinstrucciones cortas, pero, escasa capacidad para expresar paralelismo (la longitud del programa se ve incrementada).

MICROPROGRAMACIÓN HORIZONTAL: ninguna o escasa codificación. Gran capacidad para expresar un alto grado de paralelismo en las microoperaciones a ejecutar (simultáneamente). Pero las microinstrucciones son largas.

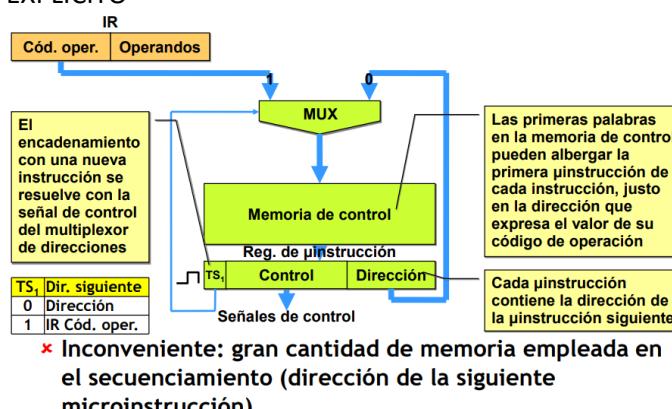
Nanoprogramación

El objetivo es reducir el tamaño de la memoria de control e implica una memoria a dos niveles: memoria de control y nanomemoria.

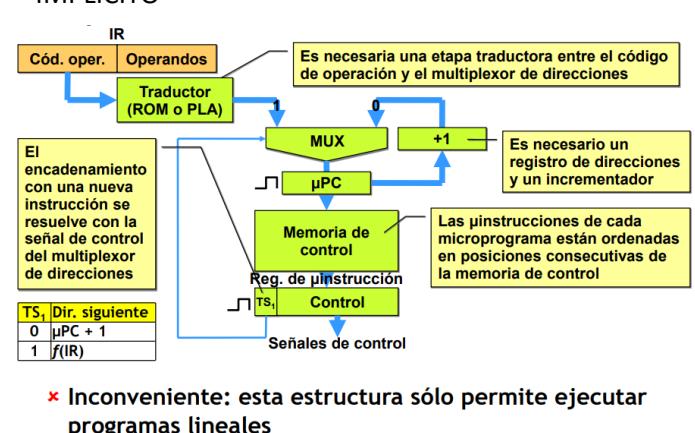
Ahorro de memoria = $n * w - (n * \lceil \log_2 m \rceil + m * w)$

Secuenciamiento de μinstrucciones

EXPLÍCITO



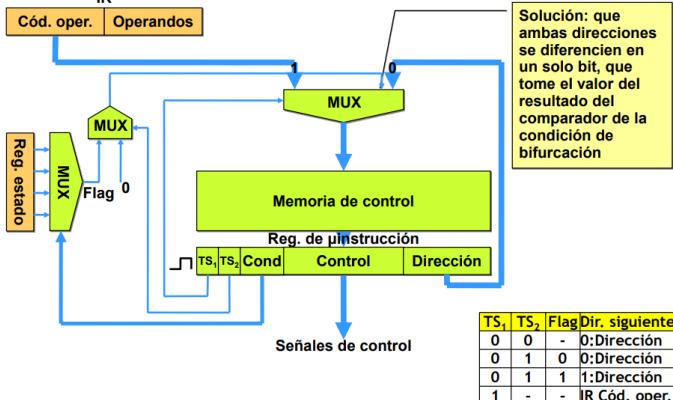
IMPLÍCITO



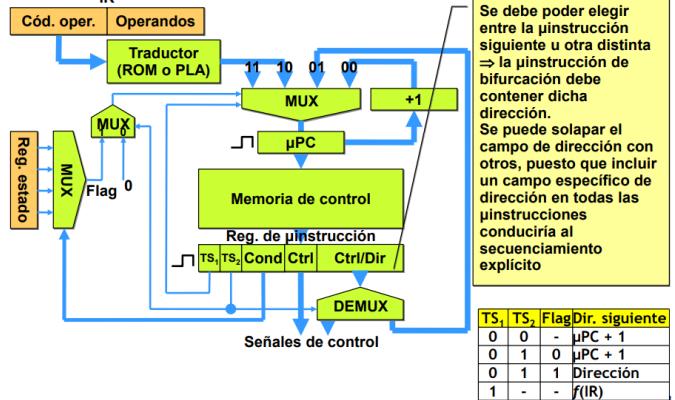
MICROBIFURCACIONES CONDICIONALES

Las instrucciones máquina de bifurcación condicional presentan dos cronogramas alternativos, diferentes a partir del punto en el que se hace la comprobación de la condición de bifurcación. Los microprogramas correspondientes han de presentar una microbifurcación condicional para seleccionar la rama deseada. Es necesario que la microinstrucción de bifurcación pueda elegir entre dos direcciones para poder seguir por uno de los dos caminos alternativos.

Ir bifurcaciones condicionales en secuenciamiento explícito (2)



Ir bifurcaciones condicionales en secuenciamiento implícito



Microbucleos: existen instrucciones máquina con operaciones repetidas. Ejs: desplazamiento múltiple, multiplicación, división y cadenas por lo que surge la necesidad de microbucleos. Se puede utilizar la bifurcación condicional más un contador con autodecremento que cuando llegue a 0 se bifurca (fuera del bucle) y debe poder inicializarse desde una microinstrucción.

Microsubrutinas: las instrucciones máquina tienen con frecuencia partes comunes, por lo que surge la necesidad de microsubrutinas. La llamada a microsubrutinas exige un almacenamiento para guardar la dirección de retorno y la solución usual es añadir una pila al registro de direcciones (μ PC).g

Control residual

Control inmediato: hasta aquí, todas las señales de control necesarias para manipular la microarquitectura estaban codificadas en campos de la microinstrucción actual.

Control residual: en ciertos casos puede ser útil que una microinstrucción pueda almacenar señales de control en un registro (de control residual) para usarlas en ciclos posteriores. El objetivo es optimizar el tamaño del microprograma y se usa en microsubrutinas o conjuntos compartidos de μinstrucciones o en caso de que parte de la información de control permanezca invariable durante muchas μinstrucciones.

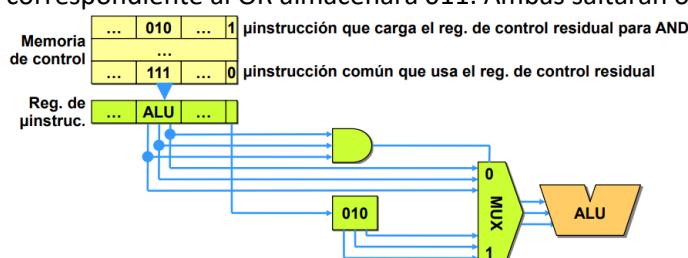
En microsubrutinas o conjuntos compartidos de μinstrucciones: *Ejemplo*: Procesador con dos instrucciones máquina para realizar las operaciones AND y OR entre dos cadenas de bytes. ♦ Se puede utilizar una microsubrutina o un conjunto común de microinstrucciones para las dos instrucciones máquina, incorporando un registro de control residual en el diseño. Supongamos que la ALU está controlada por 3 bits:

000 Suma
001 Resta
010 AND
011 OR
100 XOR
101 NOR
110 Pasar entrada izquierda
111 No utilizada

Utilizaremos el valor 111 para especificar que las señales de control de la ALU no proceden de este campo sino del registro de control residual

Necesitamos un método para cargar información en el registro de control residual, por ej. una microinstrucción especial (puede ser la microinstrucción de llamada a microsubrutina) con un campo que indique el valor a almacenar y un bit que indique que se desea almacenar información en el registro.

El microcódigo de la instrucción que realiza el AND almacenará 010 en el registro de control residual y la correspondiente al OR almacenará 011. Ambas saltarán o llamarán al microcódigo común.



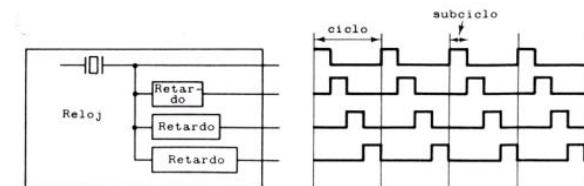
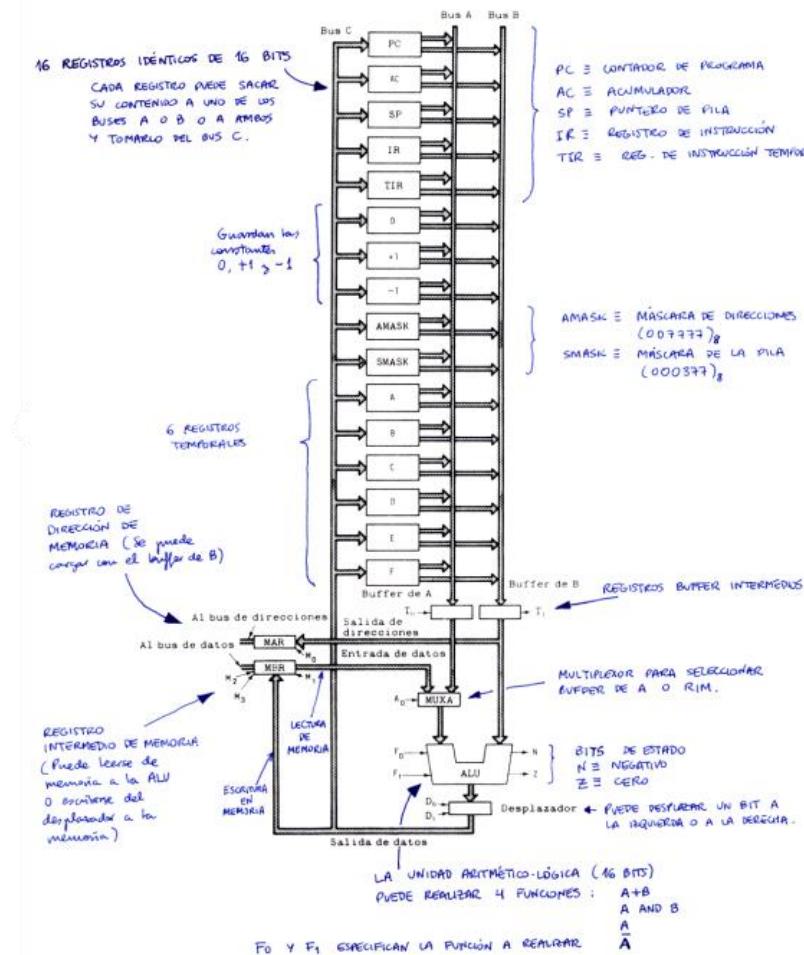
El concepto del ej. anterior puede generalizarse para la especificación de otros elementos, por ej: registros destino, operaciones de memoria, direcciones o condiciones de salto. Además, se reduce el número de μinstrucciones.

En caso de que parte de la información de control no varíe durante muchas μinstrucciones: en lugar de mantener la misma información en μinstrucciones sucesivas, ésta se coloca en el registro de control residual durante un período deseado de tiempo. En este caso el registro de control residual se suele denominar registro de setup (configuración). Se reduce la anchura de las μinstrucciones.

Ej. de arquitectura microprogramada

Camino de datos, diseño horizontal (repertorio de instrucciones máquina) y diseño vertical.

CAMINO DE DATOS

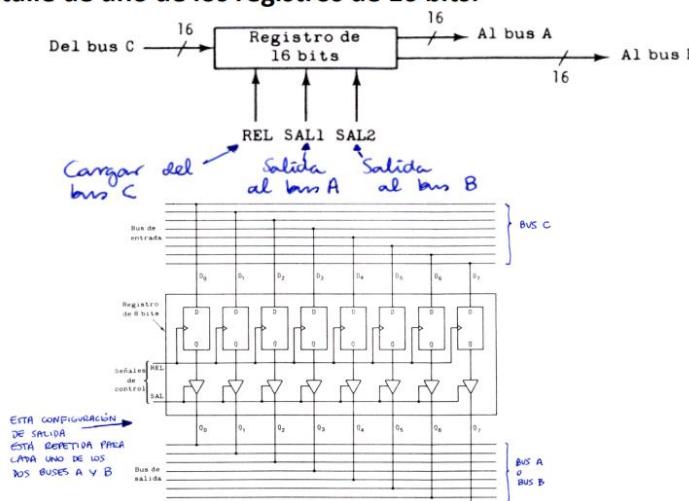


Temporización:

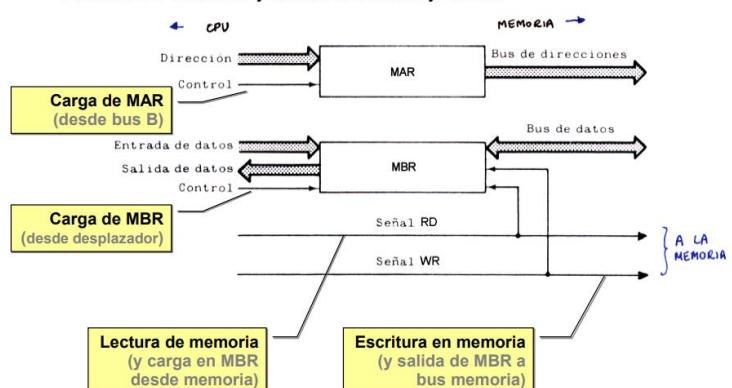
Un ciclo básico de la UC consiste en una secuencia de 4 subciclos controlada por un reloj de 4 fases.

Subciclos: se lee de la memoria de control la siguiente microinstrucción a ejecutar y se carga en el registro de microinstrucción, los contenidos de uno o dos registros pasan a los buses A y B y se cargan en los buffers, las entradas de la ALU están estabilizadas. Se da tiempo a la ALU y al desplazador a que produzcan una salida estable y se carga MAR si es necesario y la salida del desplazador está estabilizada. Se almacena el bus C en un registro y en MBR si es necesario.

Detalle de uno de los registros de 16 bits:



Detalle de señales y buses de MAR y MBR:



DISEÑO HORIZONTAL

Señales de control necesarias:

16 para salida de los registros hacia el bus A.

16 para salida de los registros hacia el bus B.

16 para carga de los registros desde el bus C.

2 para carga de los buffers de A y B.

2 para controlar la función de la ALU.

2 para controlar el desplazador.

4 para controlar MAR (1) y MBR (3).

Lo que hace un total de 61 señales.

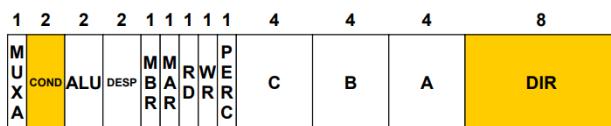
Podemos reducir las señales necesarias:

- Codificando las señales de los registros (suponiendo que el contenido del bus C sólo se almacene en un registro) 48 bits \rightarrow 12 bits ($3 \times 16 \rightarrow 3 \times 4$).
- Eliminando los 2 bits para carga de los buffers de A y B (siempre se cargan en el mismo momento del ciclo máquina, por lo que puede activarlos el segundo subciclo del reloj) 2 bits \rightarrow 0 bits.
- Reduciendo el nº de señales que controlan MAR y MBR (RD puede usarse para cargar MBR desde la memoria y WR para darle salida) 4 bits \rightarrow 2 bits
- Es necesario añadir otras señales puesto que nos puede interesar generar N y Z sin almacenar el resultado, o almacenarlo sólo en MBR. Necesitamos un bit adicional PERC (permiso de C) para que se almacene el bus C en un registro (PERC = 1) o no (PERC = 0).

Por lo que nos quedan 22 bits de control.

Formato de microinstrucción (32 bits):

- 22 bits de control
- 2 bits de condición de salto
- 8 bits de dirección



MUXA	COND	ALU	DESP	A, B, C
0 = Buffer de A	0 = No salta	0 = A + B	0 = No desplaza	Selección de uno
1 = MBR	1 = Salta si N = 1	1 = A & B	1 = Desplaza 1 bit a la dcha.	de los 16 registros
	2 = Salta si Z = 1	2 = A	2 = Desplaza 1 bit a la izda.	
	3 = Salta siempre	3 = No A	3 = (no utilizado)	
		MBR, MAR, RD, WR, PERC		Dirección de salto
		0 = No		en la memoria de
		1 = Si		control

Unidad de control

Memoria de control: 256 palabras de 32 bits = 8192 bits. La unidad de incremento calcula $\mu\text{PC} + 1$. Un ciclo de memoria principal dura dos microinstrucciones: las dos señales que controlan la memoria, RD y WR están activas mientras estén presentes en el μIR y si una microinstrucción comienza una lectura de memoria poniendo RD = 1, también debe ser RD = 1 en la siguiente microinstrucción. La elección de la siguiente microinstrucción la realiza la lógica de microsecuenciamiento durante el subciclo 4 (cuando N y Z son válidos), a partir de N, Z y los dos bits COND (I = Izdo., D = dcho.): $\text{MUXM} = /I \cdot D \cdot N + I \cdot /D \cdot Z + I \cdot D = D \cdot N + I \cdot Z + I \cdot D$

Arquitectura

Memoria principal: 4096 palabras de 16 bits.

3 registros visibles por el programador de lenguaje máquina: PC : contador de programa, SP : puntero de pila y AC : acumulador.

3 modos de direccionamiento: directo: los 12 bits menos significativos son una dirección de memoria, indirecto: AC contiene una dirección de memoria y local: los 12 bits menos significativos son un desplazamiento que se suma al puntero de pila para direccionar variables locales de procedimientos.

La pila crece hacia direcciones de memoria inferiores.

La E/S es mapeada en memoria \rightarrow no hay instrucciones de E/S específicas.

El registro AMASK (OFFF16) se utiliza para obtener el campo dirección en las instrucciones de direccionamiento directo y local.

El registro SMASK (00FF16) se utiliza para obtener el incremento/decremento del puntero de pila en las instrucciones INSP y DESP.

Lenguaje para micropogramar en alto nivel:

Los micropogramas se pueden escribir: en binario: 32 bits por microinstrucción, nombrando cada campo distinto de 0 y su valor (una microinstrucción por línea): (Ej: PERC = 1, C = 1, B = 1, A = 10) o con instrucciones de alto nivel tipo PASCAL.

•Funciones de la ALU:	•Saltos incondicionales:
ac:=a+ac;	goto 27
a:=band(ir,smask);	•Saltos condicionales:
ac:=a;	if n then goto 27;
a:=inv(a);	if z then goto 27;
•Desplazamiento:	•Examen de un registro sin almacenamiento:
tir:=lshift(tir+tir);	alu:=tir; if n then goto 27;
a:=rshift (a);	

Algunas sentencias y sus microinstrucciones

correspondientes:

	M	C	D	P		D
	U	O	A	E	M	M
	X	N	L	S	B	A
	A	D	U	P	R	R
	C	C	B	A	R	
mar:=pc; rd;	0	0	2	0	0	1
rd;	0	0	2	0	0	0
ir:=mbr;	1	0	2	0	0	0
pc:=pc+1	0	0	0	0	0	0
mar:=ir; mbr:=ac; wr;	0	0	2	0	1	1
alu:=tir; if n then goto 15;	0	1	2	0	0	0
ac:=inv(mbr);	1	0	3	0	0	0
tir:=lshift(tir); if n then goto 25;	0	1	2	2	0	0
alu:=ac; if z then goto 22;	0	2	2	0	0	0
ac:=band(ir,amask); goto 0;	0	3	1	0	0	0
sp:=sp+(-1); rd;	0	0	0	0	1	0
tir:=lshift(ir+ir); if n then goto 69;	0	1	0	2	0	0
	0	1	4	3	3	69

Microprograma: 79 microinstrucciones de 32 bits = 2528 bits. La decodificación de las instrucciones se realiza examinando IR bit a bit; con una proporción considerable de tiempo dedicada a la decodificación y el CPI disminuiría si se pudiera extraer el µPC a partir del código de operación de la instrucción máquina (goto f(ir)).

DISEÑO VERTICAL

Cada microinstrucción contiene ahora 3 campos de 4 bits: OP: código de operación (dice qué hace la microinstrucción) y r1 y r2: dos registros (dirección en el caso de los saltos).

Binario	Nemotécnico	Instrucción	Significado
0000	ADD	Suma	r1 := r1 + r2
0001	AND	AND bit a bit	r1 := r1 & r2
0010	MOV	Mueve reg. a reg.	r1 := r2
0011	NEG	Complementa	r1 := inv(r2)
0100	SHL	Desplaza a la izda.	r1 := lshift(r2)
0101	SHR	Desplaza a la dcha.	r1 := rshift(r2)
0110	MOV_MBR	Mueve MBR a registro	r1 := MBR
0111	TST	Examina registro	if r2 < 0 then n:=true; if r2 = 0 then z:=true
1000	LD_1	Comienza lectura	MAR := r1; RD
1001	ST_1	Comienza escritura	MAR := r1; MBR := r2; WR
1010	LD_2	Termina lectura	RD
1011	ST_2	Termina escritura	WR
1100	-	(no usado)	
1101	BN	Salta si N = 1	if n then goto dir
1110	BZ	Salta si Z = 1	if z then goto dir
1111	JMP	Salta siempre	goto dir

Señales generadas por el decodificador de OP: requiere una PLA con 4 entradas, 13 salidas y 15 términos producto.

Binario	Nemotécnico	ALU A	ALU B	DESP A	DESP B	NZ	MUXA	Y	MAR	MBR	RD	WR	LMS A	LMS B
0000	ADD					1		1						
0001	AND		1			1		1						
0010	MOV	1				1		1						
0011	NEG	1	1			1		1						
0100	SHL	1		1		1		1						
0101	SHR	1			1	1		1						
0110	MOV_MBR	1				1	1	1						
0111	TST	1				1								
1000	LD_1	1							1		1			
1001	ST_1	1							1	1		1		
1010	LD_2	1									1			
1011	ST_2	1									1			
1100	-													
1101	BN											1		
1110	BZ											1		
1111	JMP											1	1	

Microprograma: 160 microinstrucciones de 12 bits = 1920 bits. Menos memoria de control, pero UC más lenta (¿por qué? ¿cuánto más lenta (suponer todas las instrucciones equiprobables)?).

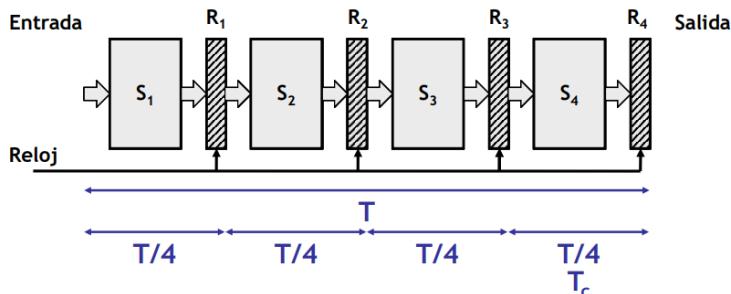
TEMA 4

1. Concepto de segmentación

La segmentación de cauce (pipeling) surge como respuesta a la pregunta de si es posible incrementar la velocidad de procesamiento, al margen de mejoras tecnológicas, sin incrementar el número de procesadores.

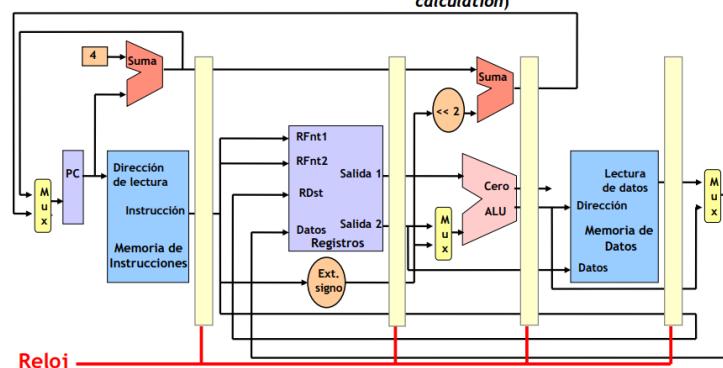
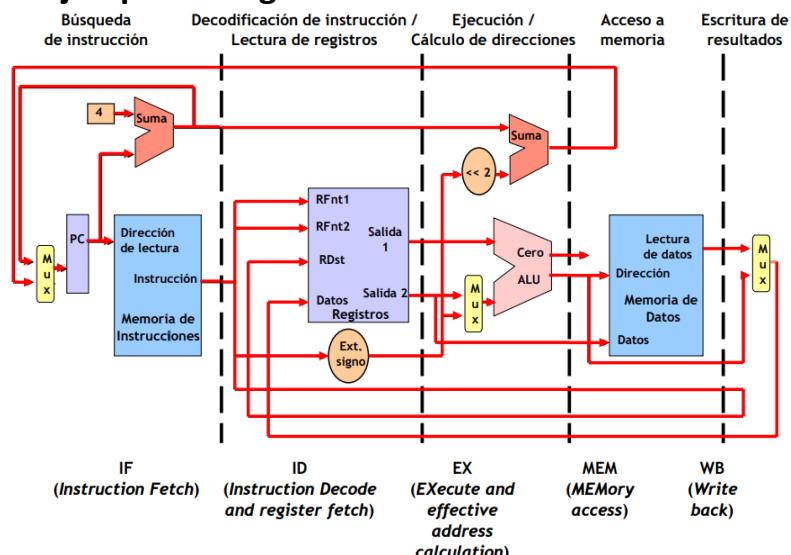
La segmentación consiste en subdividir el procesador en n etapas, permitiendo el solapamiento en la ejecución de instrucciones.

Las instrucciones entran por un extremo del cauce, son procesadas en distintas etapas y salen por el otro extremo.



Cada instrucción individual se sigue ejecutando en un tiempo T...
...pero hay varias instrucciones ejecutándose simultáneamente

2. Ejemplo de segmentación



Cada etapa del cauce debe completarse en un ciclo de reloj. Las fases de captación y de memoria si acceden a memoria principal, es varias veces más lento mientras que la caché permite acceso en un único ciclo de reloj. El periodo de reloj se escoge de acuerdo a la etapa más larga del cauce.

3. Aceleración

$$\text{Aceleración ideal} = \frac{kT}{(n-1+k)T/n} = \frac{nkT}{(n-1+k)T} = n \quad (k \rightarrow \infty)$$

Siendo k el número de instrucciones y n las etapas.

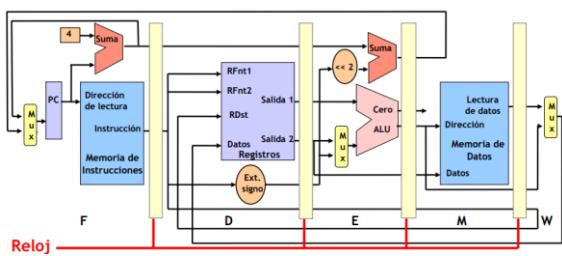
La aceleración ideal coincide con el número de etapas de segmentación.

Las causas que disminuyen la aceleración son el coste de la segmentación y la duración del ciclo de reloj impuesto por etapa más lenta $\rightarrow T_c > T/n$ y los riesgos debido al bloqueo del avance de instrucciones.

4. Riesgos

Un riesgo es una situación que impide la ejecución de la siguiente instrucción del flujo del programa durante el ciclo de reloj designado lo que obliga a modificar la forma en la que avanzan las instrucciones hacia las etapas siguientes llevando a la reducción de las prestaciones logradas por la segmentación.

Supongamos las siguientes etapas: F: búsqueda (fetch) de instrucción; D: decodificación de instrucción / lectura de registros; E: ejecución / cálculo de direcciones; M: acceso a memoria y W: escritura de resultados.

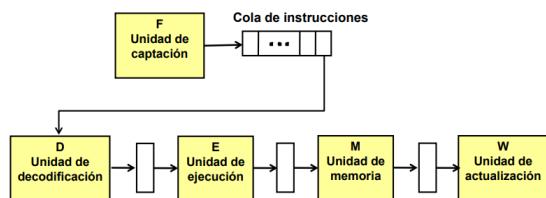


Riesgos estructurales

Conflictos por el empleo de recursos, dos instrucciones necesitan un mismo recurso.

Ej. lectura de dato + captación suponiendo una única memoria para datos e instrucciones.

lw	r4,20(r1)	F D E M W
and	r7,r2,r5	F D E M W
or	r8,r6,r2	F D E M W
add	r9,r2,r2	F D E M W
lw	r4,20(r1)	F D E M W
and	r7,r2,r5	F D E M W
or	r8,r6,r2	F D E M W
add	r9,r2,r2	- F D E M W



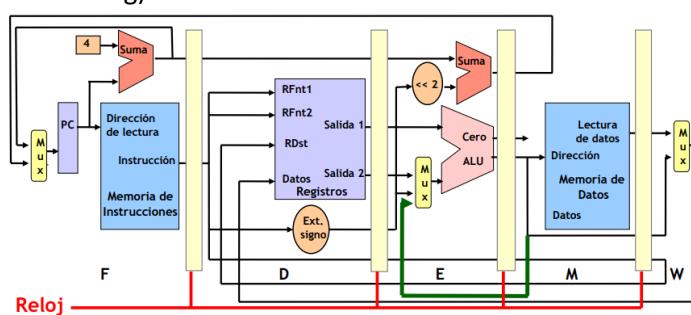
Para reducir el efecto de los fallos de caché se suelen capturar instrucciones antes de que sean necesarias (precaptación) y se almacenan en una cola de instrucciones.

Riesgos (por dependencias) de datos

Acceso a datos cuyo valor actualizado depende de la ejecución de instrucciones precedentes.

sub	r2,r1,r3	F D E M W
and	r7,r2,r5	F D E M W
or	r8,r6,r2	F D E M W
add	r9,r2,r2	F D W M W
sub	r2,r1,r3	F D E M W
and	r7,r2,r5	F D - - D E M W
or	r8,r6,r2	F - - - D E M W
add	r9,r2,r2	F D E M W

El nuevo r2 está a la salida de la ALU al terminar E. Si r2 se envía de nuevo a la ALU se elimina el retardo (register forwarding).



Las dependencias de datos las descubre el hardware al decodificar las instrucciones. Alternativamente puede resolverlas el compilador:

```

sub    r2,r1,r3      F D E M W
nop
nop
nop
and   r7,r2,r5      F D E M W

```

Tiene como ventajas que el hardware más simple, reorganiza las instrucciones para hacer trabajo útil en lugar de NOP; pero tiene como inconveniente que aumenta el tamaño del código.

Riesgos de control

Consecuencia de la ejecución de instrucciones de salto. Salto incondicional:

br	L1	F D E M W
and	r2,r1,r4	F D - - -
sub	r5,r6,r7	F - - - -
or	r8,r1,r6	
L1:add	r6,r1,r4	F D E M W

br	L1	F D E M W
and	r2,r1,r4	F - - - -
sub	r5,r6,r7	
or	r8,r1,r6	
L1:add	r6,r1,r4	F D E M W

Se pierden 2 ciclos (huecos de retardo de salto), ya que tras captar la instrucción br, hasta después del 3º ciclo (es decir, pasados otros 2 ciclos) no se conoce la dirección de salto. Por lo que es importante averiguar la dirección de salto lo antes posible; ej. en la etapa de decodificación y así solo pierde 1 ciclo, ya que tras captar la instrucción br, después del 2º ciclo ya se conoce la dirección de salto.

Degradación de prestaciones debida a los saltos: supongamos algún mecanismo hardware que permita descartar la ejecución de las instrucciones siguientes si se produce el salto.

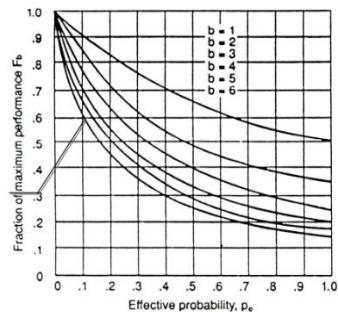
- b: nº de ciclos desperdiciados cuando se produce el salto.
- pb: probabilidad de que se ejecute una instrucción de salto (entre 0,15 y 0,30 normalmente).
- pt : probabilidad de que realmente se produzca el salto cuando se ejecuta una instrucción de salto.
- pe = pb pt : probabilidad efectiva de que se produzca un salto.

CPI: nº de ciclos de reloj por instrucción (suponer CPI = 1 sin saltos)

$$CPI = (1 - p_b) (1) + p_b [p_t (1 + b) + (1 - p_t) (1)] = 1 + p_b p_t b = 1 + p_e b$$

Fb: fracción de máximas prestaciones (relación entre el nº de ciclos CPI si no hubiera saltos y el nº de ciclos CPI con saltos).

$$F_b = \frac{1}{1 + p_e b}$$



La degradación crece rápidamente al crecer pe (más rápidamente a medida que b es mayor)

Salto retardado (delayed branch)

En lugar de desperdiciar las etapas posteriores a la de salto, una o más instrucciones parcialmente completadas se completarán antes de que el salto tenga efecto. El compilador busca instrucciones anteriores lógicamente al salto que pueda colocar tras el salto. Si el salto es condicional, las instrucciones colocadas detrás no deben afectar a la condición de salto.

Otra posibilidad es colocar la(s) instrucción(es) destino de un salto tras el salto. Aunque esto no funciona para saltos condicionales ya que no podemos "subir" una instrucción que sólo tiene que ejecutarse algunas veces (cuando el salto se produce) a una posición donde siempre se ejecuta.

Annuling branch

Un salto de este tipo ejecuta la(s) instrucción(es) siguiente(s) sólo si el salto se produce, pero la(s) ignora si el salto no se produce. Con un salto de este tipo, el destino de un salto condicional sí puede colocarse tras el salto.

Predicción de saltos

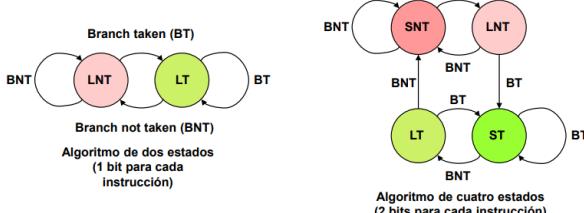
Intentar predecir si una instrucción de salto concreta dará lugar a un salto (branch taken) o no (branch not taken). Esto pasa si los resultados de las instrucciones de salto condicional fueran aleatorios, comenzar a ejecutar las instrucciones siguientes al salto desperdiciaría ciclos en la mitad de las ocasiones; se pueden minimizar las pérdidas de ciclos innútiles si para cada instrucción de salto se puede predecir con un acierto > 50% si el salto se producirá o no, se pueden hacer predicciones distintas si el salto es hacia direcciones menores o mayores. Hay dos tipos: estática, se toma la misma decisión para cada tipo de instrucción y dinámica cambia según la historia de ejecución de un programa.

ST: Muy probable saltar

LT: Probable saltar

LNT: Probable no saltar

SNT: Muy probable no saltar



5. Influencia en el repertorio de instrucciones

Es deseable que un operando no necesite más de un acceso a memoria: constante, registro, indirecto a través de registro e indexado (EA = reg. + desp., o EA = reg. + reg.). Es deseable que sólo puedan acceder a memoria las instrucciones de carga y almacenamiento.

Modo de direc. complejo, 2 accesos a memoria

lw	r4, (20(r1))	F D E M M W
and	r7, r2, r5	F D E - M W

Modo de direc. más simple, 1 acceso a memoria

lw	r3, 20(r1)	F D E M W
lw	r4, (r3)	F D E M W
and	r7, r2, r5	F D E M W

Idéntica duración, pero hardware más sencillo.

Códigos de condición

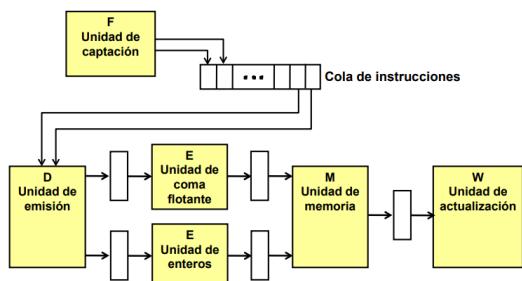
Las dependencias que introducen los bits de condición dificultan al compilador reordenar el código (deseable para evitar riesgos). En el ejemplo siguiente, la decisión de saltar se produce tras la etapa E de la instrucción cmp. Es deseable elegir en cada instrucción si afecta o no a los códigos de condición, para reordenar el código ya que, al hacerlo, se puede tomar la decisión de salto un ciclo antes, y desperdiciar un ciclo menos tras el salto.

6. Funcionamiento superescalar

Procesamiento segmentado: una instrucción tras otra con el rendimiento ideal de una instrucción por ciclo.

Procesamiento superescalar: varias instrucciones en paralelo, necesidad de varias unidades funcionales, emisión múltiple: puede comenzar a ejecutar más de una instrucción por ciclo de reloj, rendimiento: más de una instrucción por ciclo y fundamental poder captar instrucciones rápidamente: conexión ancha con caché + cola de instrucciones.

Ej. Procesador con dos unidades de ejecución:



El efecto negativo de los riesgos es más pronunciado, pero el compilador puede reordenar instrucciones para evitar riesgos.

fadd	rx, rx, rx	F D E₁E₂E₃M W
add	rx, rx, rx	F D E M W
fsub	rx, rx, rx	F D E₁E₂E₃M W
sub	rx, rx, rx	F D E M W

Las instrucciones pueden emitirse en orden y finalizar de forma desordenada (ej. add finaliza antes que fadd); lo que da múltiples problemas y soluciones, que se verán en AC.

TEMA 5

1. Funciones del sistema de E/S. Interfaces de E/S

El objetivo de los sistemas de E/S es: realizar la conexión del procesador con una gran variedad de dispositivos periféricos, teniendo en cuenta que las características de los dispositivos de E/S suelen diferir notablemente de las del procesador; en especial: la velocidad de transmisión de los periféricos; que es normalmente menor que la velocidad a la que opera el procesador y muy variable (pocos bytes/s hasta >100 MB/s). la longitud de palabra y los códigos para representar los datos. Para compatibilizar las características de los dispositivos de E/S con el sistema procesador/memoria se usan los circuitos de interfaz o controladores de periféricos.

Interfaces de E/S

Circuitos de interfaz o controladores de periféricos. Circuitos de adaptación de formato de señales y características de temporización entre el procesador y los dispositivos de E/S, proporcionan todas las transferencias de datos necesarias entre el procesador y los periféricos, utilizando un bus de E/S. Requieren uso de software: programas de E/S ejecutados por el procesador que controlan la transferencia de información hacia y desde los dispositivos de E/S. En computadoras de altas prestaciones se han utilizado procesadores especializados para las funciones de E/S: procesadores de E/S (IOP) o canales. Procesadores cuyos conjuntos de instrucciones se restringen a aquellas que se precisan en las operaciones de E/S y se hacen cargo de todas las transferencias con los periféricos.

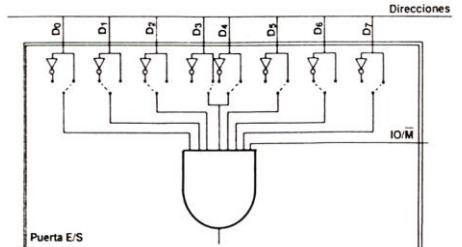
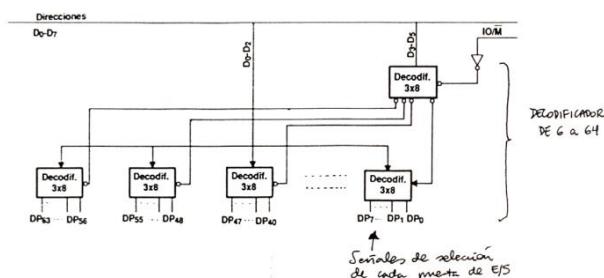
Funciones que debe incluir el sistema de E/S

Direccionamiento o selección del periférico:

El procesador sitúa en el bus de direcciones la dirección asociada con el dispositivo, si se conectan varios periféricos debe preverse la forma de que no haya conflictos de acceso al bus. Con p bits pueden direccionarse 2^p direcciones distintas (mapa de E/S). Cada dirección especifica uno o dos puertos de E/S: el hardware de cada dirección suele ser único (bien entrada o bien salida), pero a veces los circuitos de E y de S de una única dirección son independientes (misma dirección \Rightarrow dos puertos, uno de entrada y otro de salida). Cada interfaz de periférico emplea varios puertos para comunicarse con el procesador.

Hay dos técnicas de direccionamiento:

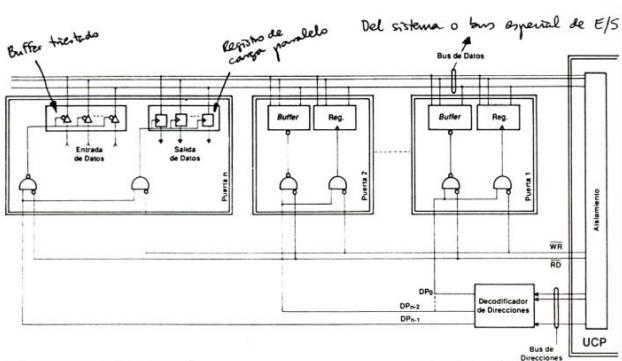
- Direccionamiento por selección lineal: asignar un bit del bus de direcciones a cada puerto.
- Direccionamiento por decodificación: decodificar los bits de dirección para seleccionar un puerto de una interfaz.
 - Decodificación centralizada: un decodificador selecciona cada puerto de E/S.
 - Decodificación en cada puerto de E/S: cada puerto reconoce su propia dirección.



Comunicación física entre el periférico y el procesador.

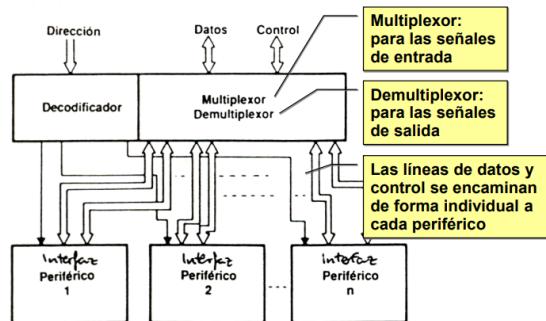
Técnicas básicas de conexión:

1. Bus



Permite conectar en paralelo un gran número de periféricos y es fácil expandir el sistema (añadiendo más tarjetas o circuitos de interfaz).

2. Multiplexor / demultiplexor



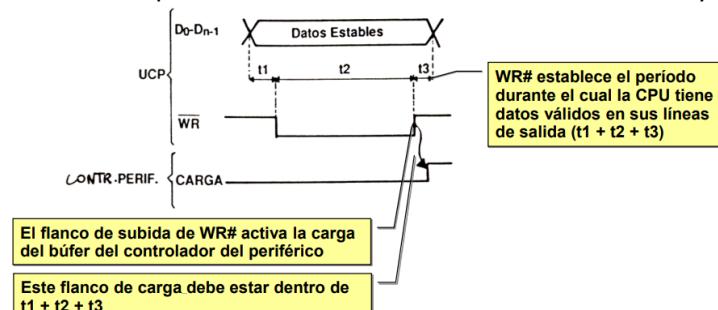
La expansión es difícil y hace falta mucha circuitería.

Sincronización:

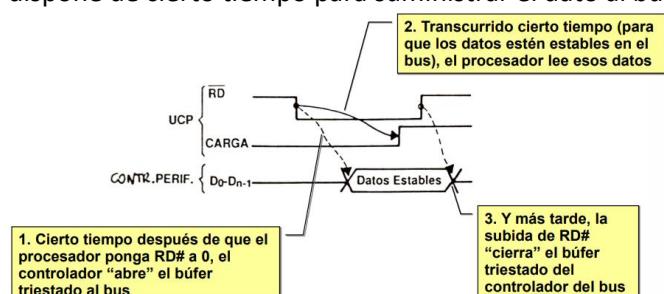
Acomodación de las velocidades de funcionamiento del procesador/MP y los dispositivos de E/S. Hay que establecer un mecanismo para saber cuándo se puede enviar o recibir un dato. Deben incluirse: palabras de memoria temporal en la interfaz que sirvan como búfer; la entrada o salida se hace sobre este búfer intermedio y la operación de E/S real se realiza sólo cuando el dispositivo está preparado y señales de control de conformidad para iniciar o terminar la transferencia (listo, petición, reconocimiento). La temporización de las transferencias puede ser síncrona o asíncrona.

TEMPORIZACIÓN SÍNCRONA

Escritura: el procesador sitúa los datos en el bus de datos y al mismo tiempo genera una señal WR# = 0.



Lectura: el procesador suministra una señal de lectura RD# = 0. A partir de entonces el controlador del periférico dispone de cierto tiempo para suministrar el dato al bus de datos.

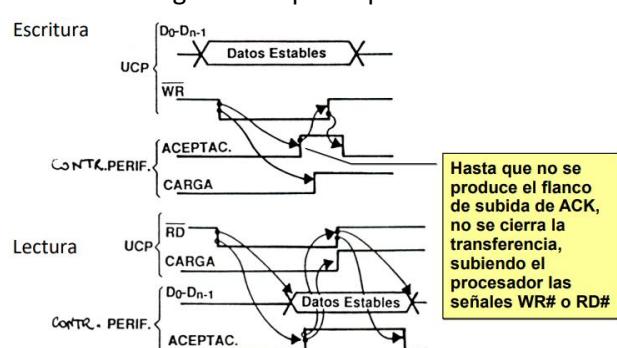


El procesador establece la temporización, que debe ser seguida por el controlador del periférico.

Si éste no es capaz de leer el bus, o de poner el dato, en el tiempo establecido ⇒ la transferencia fracasa; podría haberse direccionado un dispositivo lento, inexistente o apagado, en lectura el procesador almacenará un dato incorrecto y en escritura el procesador no sabrá si ha escrito con éxito.

TEMPORIZACIÓN ASÍNCRONA O CON "HANDSHAKING"

Se necesita una nueva señal de aceptación (ACK) con la que el controlador del periférico contesta a la petición de transferencia generada por el procesador.



Se establece un diálogo (handshaking) para adaptar el cronograma a las necesidades de tiempo del periférico.

Handshaking: establecimiento de una comunicación sincronizando dos dispositivos mediante acuse de recibo o intercambio de señales de control. Ventajas: se pueden conectar dispositivos con distintos requisitos de tiempo, se tiene una mayor garantía de que el dato sea válido, puesto que se exige una contestación positiva del periférico. Para evitar que el sistema quede bloqueado si no existe el periférico o éste no contesta, es necesario establecer un período de espera máximo, después del cual se considera la transferencia como errónea.

Conversión de datos:

Acomodación de las características físicas y lógicas de las señales de datos empleadas por el dispositivo de E/S y por el bus del sistema.

- Conversión de códigos (BCD, ASCII, EBCDIC, UNICODE, ANSI, etc.)
- Conversión serie / paralelo.
- Conversión de niveles lógicos para representar 1 y 0.
- Conversión A/D y D/A.

Control de los periféricos:

Interrogación y modificación de su estado: encendido, apagado, disponible... Envío de otras señales de control al periférico.

Mecanismo que determine la cantidad de información a transmitir en una operación de E/S y cuente el número de palabras/bytes ya transmitidos.

Detección de errores:

En el funcionamiento del periférico o en los datos, mediante códigos de paridad, polinomiales... Se repetirá la transferencia en caso necesario.

Conceptos a diferenciar

Transferencia elemental de información: envío o recepción de una única unidad de información (byte o palabra), ya sea un dato o una palabra de estado o control.

Operación completa de E/S: transferencia de un conjunto de datos: sector de un disco o línea de pantalla.



Dispositivos de E/S físicos: cuando el ordenador carece de SO o driver adecuados el programador debe tratar directamente con ellos, asumiendo sus detalles de funcionamiento y características físicas.

Dispositivos de E/S lógicos: programador efectúa transferencias de datos activando las rutinas de E/S que proporciona el SO. Ejemplo, puede escribir un programa que escriba en una impresora lógica (en realidad un bloque de espacio en disco). El SO asigna una de las impresoras físicas a la impresora lógica y controla el proceso de impresión (spooling).



E/S aislada o independiente: el procesador distingue internamente entre espacio de memoria y espacio de E/S.



E/S mapeada en memoria: el procesador no distingue entre accesos a memoria y accesos a los dispositivos de E/S.

E/s independiente frente a e/s en memoria

E/S AISLADA, INDEPENDIENTE, O CON ESPACIO DE E/S.

Emplea la patilla IO/M# del procesador. Nivel alto ⇒ Indica a memoria y a dispositivos de E/S que se va a efectuar una operación de E/S, al ejecutar instrucciones específicas de E/S: IN y OUT. Nivel bajo ⇒ Operación de intercambio de datos con memoria, al ejecutar instrucciones de acceso a memoria: LOAD, STORE o MOVE. Instrucciones específicas: IN y OUT (o READ y WRITE), con poca riqueza de direccionamiento. *Ej:* procesadores de 8 bits empleaban dirección de 8 bits para puertos de E/S ⇒ 256 puertos: IN puerto, OUT puerto y disponían de bus de direcciones de 16 bits ⇒ 64 K direcciones de memoria.

Ventajas:

- ✓ Diseño más limpio de la decodificación de las direcciones de memoria.
- ✓ Facilita la protección de E/S (por ejemplo, haciendo que las instrucciones IN, OUT... sean privilegiadas).
- ✓ Los programas son relativamente más rápidos por la decodificación más sencilla y el menor tamaño de las instrucciones de E/S.

Desventajas:

- ✗ Mayor complejidad en el diseño del procesador: hay que decodificar y ejecutar las instrucciones IN, OUT... y hay que generar la señal IO/M# para la cual se necesita una patilla más del procesador.

E/S MAPEADA EN MEMORIA

Se usan algunas direcciones de memoria para acceder a los puertos de E/S, tras decodificarlas adecuadamente. El procesador no distingue entre accesos a memoria y accesos a los dispositivos de E/S, no se usa la patilla IO/M#. No se dispone de instrucciones especiales, sino que se usan LOAD, STORE o MOVE y para evitar particionar el mapa dedicado a memoria, se agrupa la E/S en una zona bien definida al principio o final del mapa de memoria.

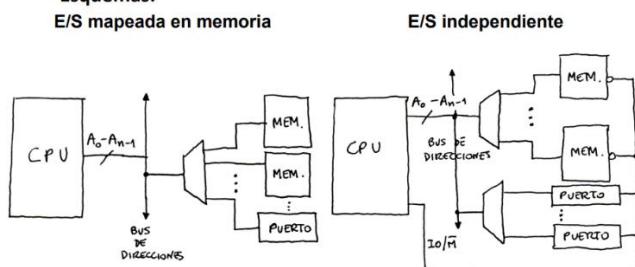
Ventaja:

- ✓ Menor complejidad en el diseño del procesador.

Desventajas:

- ✗ Cada puerto de una interfaz “ocupa” una dirección que no puede utilizarse para memoria.
- ✗ Suelen ser más largas que las específicas de E/S. Ej: en los microprocesadores de 8 bits: 3 bytes frente a 2. Puede disminuir la velocidad de procesamiento y aumentan los requisitos de memoria.

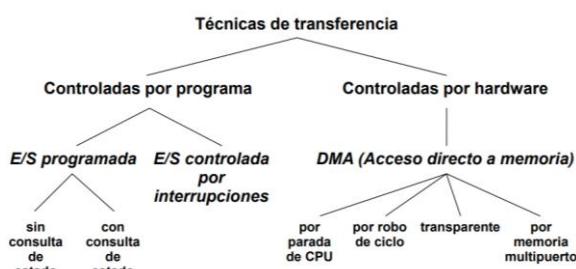
Esquemas:



Técnicas de E/S

Procedimientos para realizar el intercambio de información entre procesador, memoria y E/S.

Para realizar la comunicación con los periféricos, consideramos una versión simplificada de los controladores: uno o varios registros (puertos o puertas –ports– de E/S): datos, control y estado. Lógica de control: interpreta las señales de control/estado emitidas por el procesador y genera las señales de control/estado que el procesador exige.



E/S programada: el procesador participa activamente ejecutando instrucciones en todas las fases de una operación de E/S: inicialización, transferencia de datos y terminación.

E/S controlada por interrupciones: los dispositivos se conectan al procesador a través de líneas de petición de interrupción, que se activan cuando se requieren los servicios del procesador. En respuesta, el procesador suspende la ejecución del programa y ejecuta un programa de gestión de interrupción para transmitir datos con el dispositivo; como en la programada, los pasos de transferencia de datos están bajo el control directo de programas de control.

E/S mediante DMA: requiere presencia de un controlador DMA, actúa como controlador del bus y supervisa las transferencia de datos entre MP y dispositivos de E/S, no interviene el procesador directamente salvo inicialización.

Función o parámetro	Método de control de E/S		
	E/S programada	E/S controlada por interrupciones	Acceso directo a memoria
Comienzo de las operaciones de E/S	La CPU lee y comprueba el estado de los dispositivos de E/S (en el caso de consulta de estado).	El dispositivo de E/S envía una petición de interrupción a la CPU. Esta transfiere el control a una rutina de servicio P. (Para cada bloque)	El dispositivo de E/S envía una petición de interrupción a la CPU. La CPU transfiere el control a la rutina de servicio P'. P' inicializa el control de DMA.
Transferencia de datos de E/S	La CPU ejecuta un programa de transferencia de datos.	El controlador de DMA transfiere bloques de datos por el bus del sistema. (Para cada bloque)	
Finalización de las operaciones de E/S	Fin de ejecución del programa de transferencia de datos	La palabra-contador DMA llega a 0. El controlador de DMA envía una petición de interrupción a la CPU.	
Complejidad del circuito de interfaz de E/S	La menor	Baja	Moderada
Velocidad de respuesta a una petición de transferencia de datos por el dispositivo de E/S	Lenta	Rápida	La más rápida
Máxima velocidad de transferencia de E/S	Moderada	Moderada	Alta

2. E/S programada

Todos los pasos de una operación de E/S requieren la ejecución de instrucciones por parte del procesador. La transferencia de un byte se realiza mediante la ejecución de una instrucción de E/S: instrucción de transferencia de datos en la que un operando es un registro del controlador del periférico (puerto) y el otro operando es un registro del procesador (o posición mem.). Al decodificar la instrucción de E/S, la UC del procesador envía al exterior información de: dirección (sobre qué periférico se realiza la transferencia), tipo de operación (lectura / escritura) y temporización/sincronización y envía el dato o se dispone a recibirlo.

Salida: el procesador ejecuta una instrucción de carga de una palabra de memoria a un registro y una instrucción de salida que transfiere el dato desde el procesador a un puerto de salida.

Entrada: el procesador ejecuta una instrucción de entrada y una instrucción de almacenamiento.

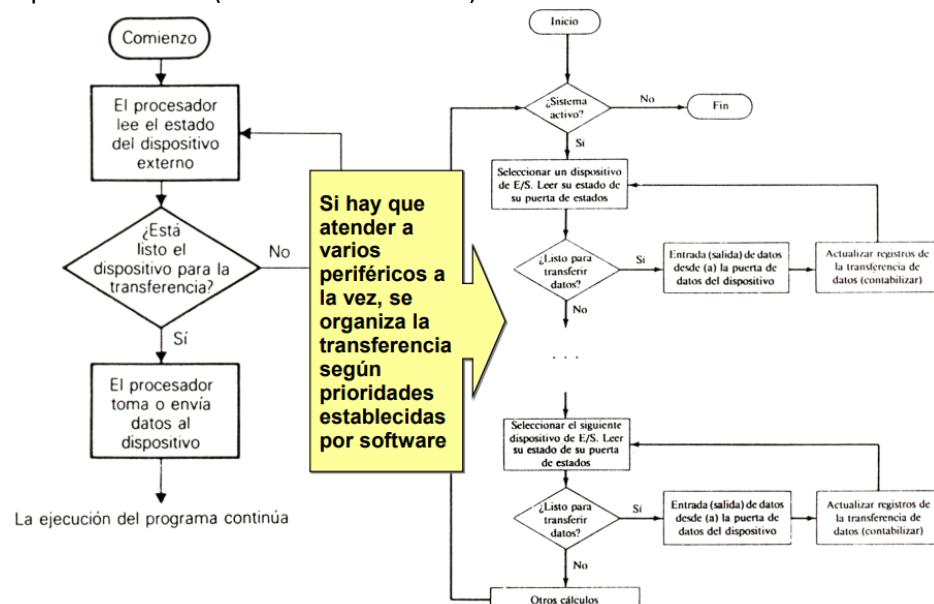
Además, se pueden necesitar instrucciones adicionales, por ejemplo, para actualizar el número de palabras transferidas.

Sin consulta de estado, o incondicional.

El procesador decide en qué momento se realiza la transferencia; el dispositivo externo debe estar siempre dispuesto a recibir datos (salida) o tener siempre datos disponibles (entrada). *Ej:* salida a un display de 7 segmentos, entrada desde un joystick.

Con consulta de estado, condicional, o con escrutinio

El procesador pregunta al periférico (a través de la interfaz) si está preparado o no para la transferencia. Además de los puertos de E/S, la interfaz tiene un registro de estado con información sobre: dato listo (si se usa como entrada) o periférico libre (si se usa como salida).



3. Interrupciones

Bifurcaciones normalmente externas, excepto en las software, al programa en ejecución, provocadas por diversas causas: externas (señales del exterior del procesador), o internas (la puede producir el propio procesador). Su objetivo es reclamar la atención del procesador sobre algún acontecimiento o hecho importante pidiendo que se ejecute un programa específico para tratar dicho acontecimiento, de manera que el programa en ejecución queda temporalmente suspendido (el código que se ejecuta en respuesta a una solicitud de interrupción se conoce como rutina de servicio de interrupción o ISR (Interrupt Service Routine)).

Hay que diferenciar entre:

- Interrupción propiamente dicha: diálogo de señales necesario para que el procesador acepte, de forma correcta, la solicitud de interrupción y salte al programa que se debe ejecutar.
 - Tratamiento de la interrupción: ejecución del programa de gestión de la interrupción.
 - La frontera entre las dos funciones puede resultar difusa, ya que hay pasos que en algunas soluciones se hacen mediante diálogo de señales y en otras mediante programas.

Secuencia de eventos



Causas

Clasificación en función de las distintas situaciones que pueden requerir que el procesador sea interrumpido:

1. Fallo del hardware

Los circuitos efectúan comprobaciones para verificar si funcionan correctamente, y si no es así, generan interrupciones.

Ej: Fallo de alimentación: interrupción de la más alta prioridad que guarda registros (puede ser necesario conectar baterías). Errores de paridad de memoria: interrupción que reintenta la transferencia varias veces.

2. Errores de programa

Situaciones de error introducidas por el programador.

Ej: Error de desbordamiento (overflow), división por cero; en ambos casos la interrupción cancela la ejecución del programa o transfiere el control a una función del usuario. Violación de la protección de memoria; el programa de usuario intenta acceder a una dirección en una parte protegida de la memoria ⇒ interrupción que cancela la ejecución y genera mensaje de error (segmentation fault) o ejecución de códigos de operación no válidos.

3. Condiciones de tiempo real

Se espera que el ordenador responda rápido a una situación.

Ej: Vigilancia de enfermos, interrupciones para hacer sonar timbres de alarma ante situaciones de peligro. Control de herramientas, interrupción para parar el equipo y evitar daños en la pieza o la máquina cuando algo falla.

4. Entrada/Salida

Para poder hacer uso del procesador mientras un periférico no está listo para realizar la transferencia (E/S por interrupción) o durante el tiempo que el periférico realiza la transferencia (DMA), es necesario que la interfaz del periférico o el controlador de DMA puedan interrumpir al procesador.

Tipos

Clasificación en función de la procedencia de la interrupción:

1. Interrupciones externas

Se inician a petición de dispositivos externos. Pueden ser enmascarables: se pueden habilitar o inhibir usando instrucciones del tipo EI (Enable Interrupt) y DI (Disable Interrupt). Ej: interrupciones de E/S o no enmascarables: tienen mayor prioridad. Ej: para gestionar fallos del hardware externo al procesador y condiciones de tiempo real.

2. Interrupciones internas (o excepciones o traps)

Se activan de forma interna al procesador mediante condiciones excepcionales, como errores del programa o fallo del hardware interno.

3. Interrupciones software

Las instrucciones de interrupción software se emplean para realizar llamadas al SO, son más cortas que las de llamada a subrutina y no se necesita que el programa llamador conozca la dirección del SO en memoria.

Determinación de la dirección de la ISR

Direcciones fijas (o interrupciones no vectorizadas): se fijan y definen en los circuitos del procesador. Ej: una dirección fija para la rutina de servicio de cada interrupción.

Interrupciones vectorizadas: se refiere a todos los esquemas en los cuales el dispositivo que solicita una interrupción suministra de algún modo la dirección de la rutina de servicio. Hay varios modos:

- Direcciónamiento absoluto: la interfaz suministra la dirección completa de su ISR.

- Direccionamiento relativo: la interfaz sólo envía parte de la dirección, que se completa por el procesador, añade más bits o suma una cantidad. Reduce el número de bits que se transmite, lo que simplifica su diseño, pero limita el número de dispositivos que el procesador puede identificar automáticamente.
- Envío de una instrucción de bifurcación completa – en lugar de una simple dirección.

Con los métodos anteriores, la ISR para un dispositivo determinado siempre debe comenzar en la misma localización, así que para conseguir mayor flexibilidad: el programador puede almacenar en esta localización una instrucción de salto a la rutina adecuada, pero esto se puede realizar automáticamente por el siguiente mecanismo de gestión de interrupciones:

- Direccionamiento indirecto (interrupciones vectorizadas propiamente dichas): la dirección enviada por la interfaz es la posición relativa en una tabla, residente en MP, que contiene las direcciones de las ISR y cada posición de la tabla se conoce como vector de interrupción.

Para sincronizar la transmisión de la dirección, el procesador envía una señal de control: INTA# (Interrupt Acknowledge, o aceptación de interrupción) equivalente a RD# en lectura. Causando que la fuente de interrupción sitúe la dirección en un bus, siendo entonces leída por el procesador, las señales de dirección se pueden enviar por un bus especial, por el bus de datos, o por el propio bus de direcciones.

Identificación de la fuente de interrupción

Los computadores pueden tener conectados una gran variedad de dispositivos con poder de interrupción.



La acción requerida al recibir una interrupción dependerá de la causa de ésta (de qué dispositivo la produjo).

Es necesario que haya diferentes rutinas de servicio de interrupción, para cada una de las causas

Es necesario identificar la causa o el dispositivo que produjo la interrupción

Soluciones:

- Una o múltiples líneas de interrupción con un dispositivo en cada línea, cada fuente de interrupción generará una señal de interrupción en la línea apropiada. La identificación del dispositivo es trivial y la dirección de comienzo de la ISR puede ser fija o vectorizada.
- Una o múltiples líneas de interrupción, con más de un dispositivo por línea. Es normal tener varios dispositivos (fuentes de interrupciones) conectados a la misma línea de interrupción, el dispositivo que solicita la interrupción ha de ser identificado y se puede realizar por software o por hardware.
Por software: se usa cuando la dirección de salto para todos los dispositivos conectados a una misma línea es única y fija. La ISR debe identificar el dispositivo que solicita la interrupción, examinando de uno en uno los dispositivos de la línea, para transferir el control a la ISR del solicitante. Es un método lento, pero económico desde el punto de vista hardware.
Por hardware: se usan interrupciones vectorizadas; es un método rápido, pero más costoso desde el punto de vista hardware.

Sistemas de prioridad en las interrupciones

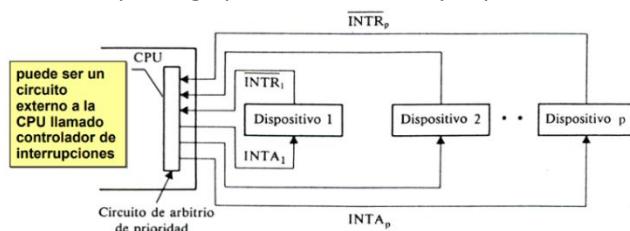
En un computador con más de un dispositivo con capacidad de interrupción, hay que establecer mecanismos de prioridad que resuelvan los problemas:

- Interrupciones simultáneas: cuando se produce una interrupción, no se acepta hasta que la instrucción ejecutada termine. Durante ese breve tiempo pueden haberse generado otras interrupciones, que requieren ser atendidas. ¿Qué interrupción se atiende primero?
- Interrupciones anidadas: se puede producir una interrupción antes de haber atendido completamente la anterior. ¿Debe terminarse de atender la primera interrupción, o se ha de aceptar y atender inmediatamente la nueva solicitud?
- Inhibición de interrupciones: mecanismos de prioridad para permitir que cada programa defina los tipos de interrupciones que puede tolerar.

INTERRUPCIONES SIMULTÁNEAS

Si llegan solicitudes de interrupción de dos o más dispositivos, el procesador debe disponer de algún medio para que sólo se dé servicio a una solicitud, y el resto se retrase o no se tenga en cuenta.

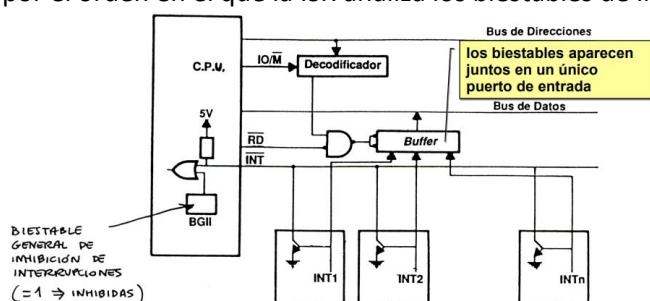
- Gestión de prioridades centralizada: hay un solo dispositivo en cada línea de interrupción, la CPU acepta la solicitud que llega por la línea de mayor prioridad.



- Gestión de prioridades distribuida: cuando varios dispositivos comparten una única línea de solicitud de interrupción es necesario implantar un mecanismo para asignar prioridades a estos dispositivos. Técnica de sondeo o “polling”, determinar por software el origen de la interrupción; la prioridad se implanta de forma automática según el orden en el que se escrutan los dispositivos, se pueden cambiar modificando la ISR para que sondee en un orden diferente. Y la técnica de encadenamiento o “daisy-chain”; el dispositivo que esté eléctricamente más cercano a la CPU tendrá la mayor prioridad.

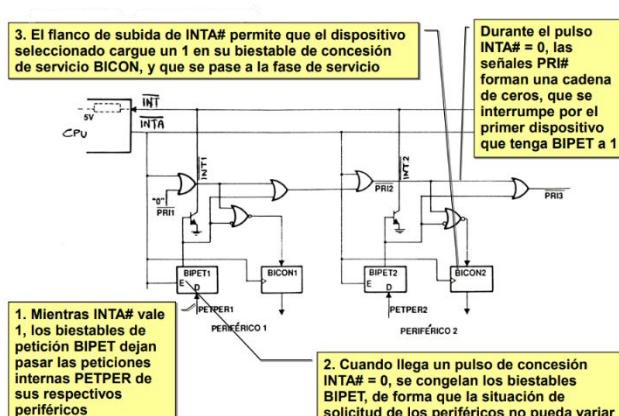
Polling

Se usa para identificar el origen de una interrupción y establecer un mecanismo software de asignación de prioridades. En esta solución, el ordenador dispone normalmente de una única línea de interrupción INT#, que sirve para que cualquier dispositivo solicite una interrupción. La línea INT# se organiza en colector abierto (OR cableado); cualquier dispositivo puede poner INTi=1 para solicitar la interrupción, lo que hace que INT# se active (se ponga a 0). La ISR está en una posición de memoria fija y se encarga de identificar cuál es el dispositivo que interrumpió, comprobando el valor de los biestables de interrupción de los dispositivos, que estarán a 1 para aquellos dispositivos que solicitan interrupción. La asignación de prioridades se hace por el orden en el que la ISR analiza los biestables de interrupción de los periféricos.

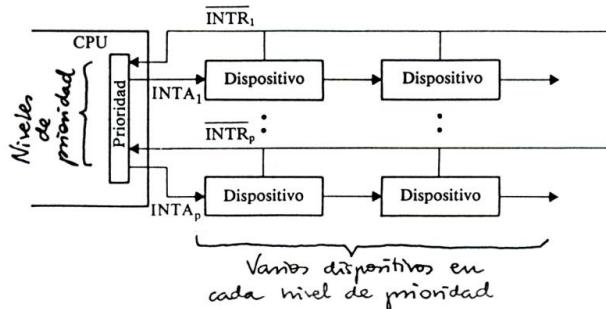


Daisy-chain

Se usa para establecer un mecanismo hardware de asignación de prioridades a los dispositivos; basado en dos señales comunes a todos los peticionarios y a la CPU: INT#: petición de interrupción e INTA#: concesión de interrupción. Los peticionarios se conectan a INT# en colector abierto, lo que permite que uno o varios dispositivos soliciten simultáneamente la interrupción, poniendo un 0 en INT#. La señal INTA# sirve, a modo de testigo, para que uno solo de los peticionarios sea atendido; es un pulso que recorre en serie, uno tras otro, los dispositivos y es tomado y eliminado por el primero que desea ser atendido.



- Gestión de prioridades híbrida: combinación de esquemas centralizado y distribuido (daisychain).



INTERRUPCIONES ANIDADAS

Si se produce una interrupción mientras se está atendiendo otra se debe:

- Inhabilitar las interrupciones durante la ejecución de la ISR; la ejecución de la ISR, una vez iniciada, siempre continuará hasta su finalización, antes de que la CPU acepte una segunda solicitud de interrupción. Esta solución es válida cuando las ISR sean breves: dispositivos simples para los cuales el retraso posible en la respuesta a la (segunda) solicitud sea aceptable.
- Permitir que la CPU acepte la segunda solicitud de interrupción durante la ejecución de la ISR, si su prioridad es mayor ya que, para algunos dispositivos, un largo retraso en la respuesta a una solicitud de interrupción los podría llevar a funcionar de forma errónea.

Ej: Reloj de tiempo real, envía solicitudes de interrupción a la CPU a intervalos regulares \Rightarrow ejecución de breve ISR que incrementa la hora (contadores de memoria). Necesita que se atienda la interrupción antes de que se produzca de nuevo, por lo que la CPU debe aceptar la solicitud de interrupción, aunque se esté atendiendo a otro dispositivo. Durante la ejecución de una ISR se aceptarán solicitudes de interrupción de algunos dispositivos, depende de su prioridad y cada vez que se acepta una interrupción, el contenido de PC se transfiere a la pila, y una vez atendida vuelve. La pila debe ser grande para que las interrupciones se puedan anidar a suficiente profundidad.

INHIBICIÓN DE INTERRUPCIONES

Hay situaciones en las que conviene evitar temporalmente que se produzcan interrupciones, para ello hay tres niveles de desactivación de interrupciones:

1. Desactivar todas las interrupciones.

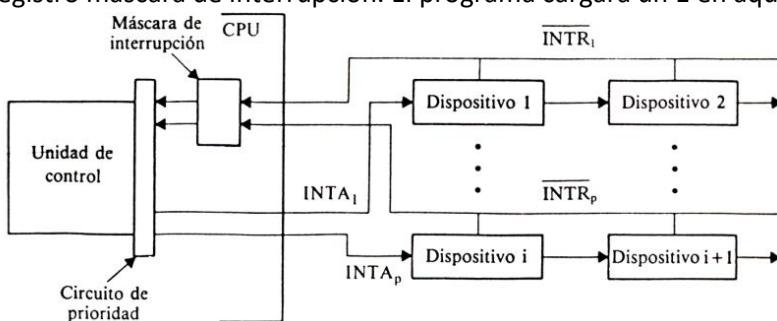
Para ello se puede usar un Biestable General de Inhibición de Interrupciones, si está a 1 (por ej.), el programa no podrá sufrir interrupciones y se puede cambiar su estado mediante instrucciones EI (Enable Interrupts) y DI (Disable Interrupts). Independientemente del valor de BGII, la CPU puede desactivar las interrupciones automáticamente: durante la ejecución de la primera instrucción de la ISR; si es una instrucción DI \Rightarrow el programador se asegura de que no ocurrirán más interrupciones hasta ejecutar EI y durante toda la ISR; no ocurrirán más interrupciones hasta ejecutar EI o retornar de la interrupción.

2. Desactivar interrupciones de inferior o igual prioridad.

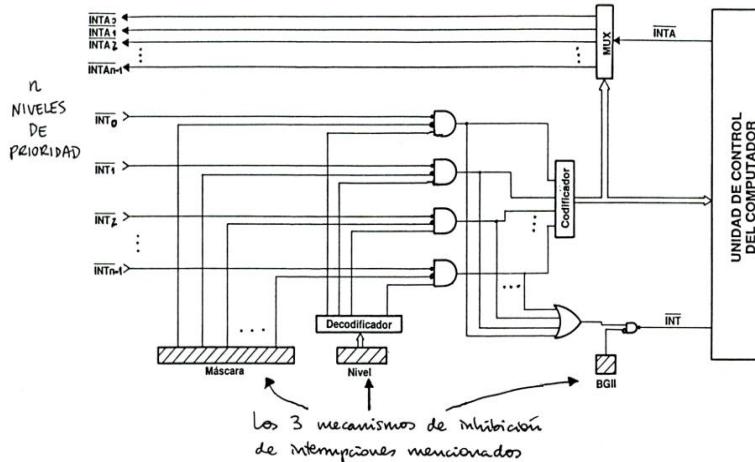
Se puede tener un registro (o varios bits del registro de estado del procesador) con el valor del menor nivel que puede interrumpir, un decodificador se encarga de desactivar todos los niveles de menor prioridad por lo que la CPU tendría un nivel de prioridad (prioridad del programa que se está ejecutando), y sólo acepará interrupciones de dispositivos con prioridades mayores que la suya.

3. Desactivar de forma selectiva determinados niveles de interrupción

La señal INTR# de cada nivel se puede enmascarar haciendo la operación AND con el bit correspondiente de un registro máscara de interrupción. El programa cargará un 1 en aquellos niveles de interrupción que se deseen inhibir



Ejemplo:

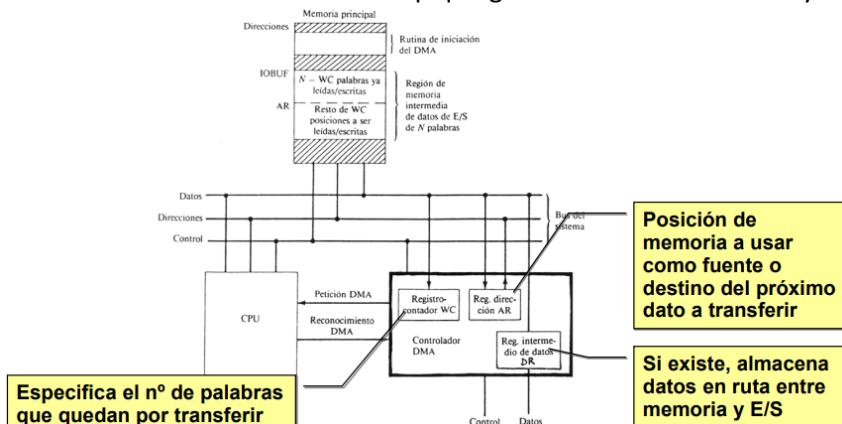


Las señales INTi# atraviesan un doble filtro: el bit de máscara ($1 \rightarrow$ nivel inhibido) y la condición: $i \leq \text{Nivel}$. El decodificador convierte el registro de nivel en los correspondientes ceros y unos en cada puerta AND para filtrar las peticiones de interrupción. INT# es la que produce la interrupción; se genera si alguna señal INTi# consigue atravesar su puerta AND correspondiente, y además el biestable general de inhibición de interrupciones está a 0. El codificador genera el valor i de la señal INTi# de mayor nivel que atraviesa el filtro, para: controlar el multiplexor que encamina la señal INTA# a su nivel correspondiente y el uso interno de la unidad de control, indicándole cual es el nivel a atender (funcionando como un vector de interrupción).

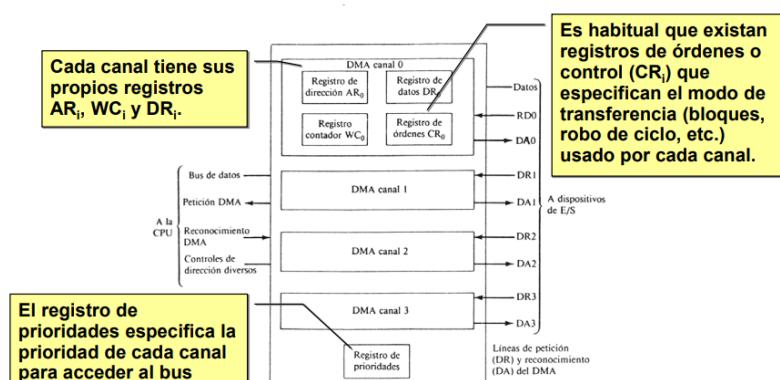
4. DMA (Acceso directo a memoria)

Permite realizar transferencias de datos entre la memoria y los dispositivos de E/S sin intervención directa del procesador, no se ejecutan instrucciones en el procesador para realizar la transferencia. Permite transferencias a la máxima velocidad permitida por el bus del sistema, la memoria y el periférico. En sistemas con un único bus, esta velocidad es mucho mayor que la máxima posible con E/S controlada por el procesador: DMA: un ciclo del bus por palabra (pocos ciclos de reloj), E/S controlada por procesador: ejecución de varias instrucciones para cada palabra (muchos ciclos de reloj) Se utiliza con dispositivos rápidos: Discos, tarjetas gráficas, tarjetas de red...

Utiliza un controlador de DMA: chip que genera señales de control y direcciones, actuando como maestro del bus.

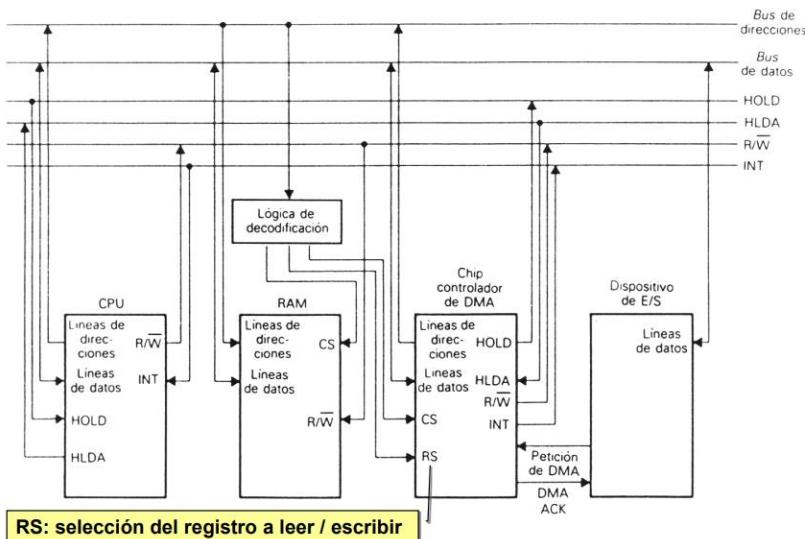


Existen DMAC que permiten varias operaciones de DMA independientes, la lógica de control y los registros asociados con cada una de esas operaciones se denomina canal de DMA.



Señales de control

El DMAC y el procesador deben compartir el bus, por lo que es necesario un mecanismo de control de acceso al bus, para ello se usan dos líneas de control entre DMAC y procesador: petición/solicitud de DMA o del bus: HOLD o BSRQ# o reconocimiento/cesión de DMA o del bus: HLDA o BSAK#. Cuando el DMAC necesita obtener el control del bus, activa HOLD, el procesador se aísla del bus y activa HLDA. El DMAC realiza la transferencia, se desconecta y desactiva HOLD.



RS: selección del registro a leer / escribir

Secuencia de eventos

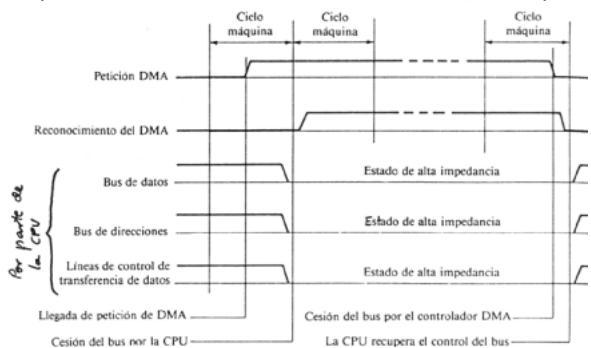
Una operación de E/S por DMA se establece con una corta rutina de inicialización. Varias instrucciones de salida para asignar valores iniciales a: AR, dirección de memoria de la región de datos de E/S IOBUF y WC, eñ número N de palabras de datos a transferir. Una vez inicializado, el DMAC procede a transferir datos entre IOBUF y el dispositivo de E/S: realiza una transferencia cuando el dispositivo de E/S solicite una operación de DMA a través de la línea de petición del DMAC, después de cada transferencia, se hace WC-- y AR++. La operación termina cuando WC = 0 ⇒ el DMAC (o el periférico) indica la conclusión de la operación enviando al procesador una petición de interrupción.

1. El procesador inicializa el DMAC programando AR y WC.
- 2. El dispositivo de E/S realiza una petición de DMA al DMAC.
3. El DMAC activa la línea de petición de DMA al procesador.
4. Al final del ciclo del bus en curso, el procesador lo pone en alta impedancia y activa la cesión de DMA.
5. El DMAC asume el control del bus.
6. El DMAC responde al dispositivo de E/S con aceptación.
7. El dispositivo de E/S transmite una nueva palabra de datos al registro intermedio de datos del DMAC.
8. El DMAC ejecuta un ciclo de escritura en memoria para transferir el registro intermedio a la posición M[AR].
9. El DMAC decrementa WC e incrementa AR.
10. El DMAC libera el bus y desactiva la línea de petición de DMA.
11. El DMAC compara WC con 0:
 - Si WC > 0 ⇒ se repite desde el paso 2.
 - Si WC = 0 ⇒ el DMAC se detiene y envía una petición de interrupción al procesador.

Métodos de control de DMA

Robo de ciclo (Cycle Stealing DMA)

La transferencia de un bloque se produce palabra a palabra, el DMAC “roba” periódicamente uno o varios ciclos máquina al procesador, durante los cuales utiliza el bus. El procesador utiliza el resto de los ciclos del bus; cuando el DMAC solicita el bus, debe esperar a que el procesador complete el ciclo máquina en curso; un robo de ciclo puede aceptarse en mitad de una instrucción (hace que la duración de las instrucciones no sea fija).



Transferencia de bloques o parada de CPU (Block Transfer DMA)

Se transmite un bloque (secuencia de palabras de datos) en una ráfaga continua, el DMAC toma el control del bus durante todo el período que dura la transferencia de datos. El procesador no tiene acceso al bus hasta que la transferencia termina, lo que le obliga a esperar períodos de tiempo relativamente grandes.

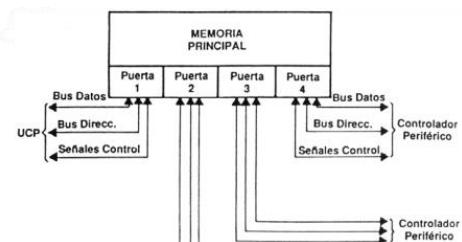
DMA intercalado o transparente (Interleaved DMA)

El DMAC toma el bus cuando el procesador no lo utiliza. *Ej:* mientras está decodificando un código de operación o efectuando operaciones internas de transferencia de registros. Para ello, el procesador debe emitir las señales de control adecuadas.

Memoria multipuerto

Existen memorias con varios módulos y varios registros de dirección y de memoria que pueden operar simultáneamente conectados a varios buses, siempre que no direccionen un mismo módulo de memoria. Se puede conectar a un puerto el procesador y a los demás puertos varios controladores de periféricos.

Si hay conflictos de acceso (peticiones simultáneas a un mismo módulo), se concede un acceso y se retardan los demás, según un sistema de prioridades.

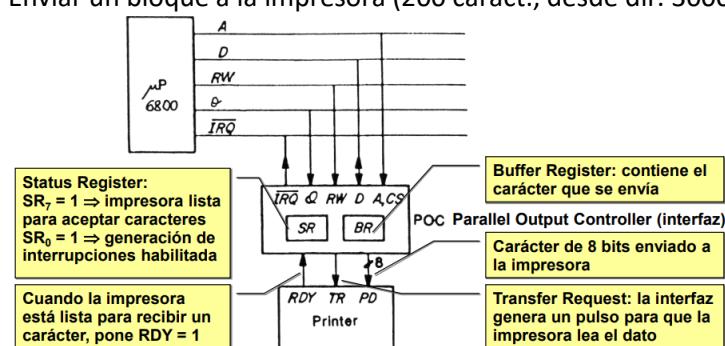


5. Ejemplos de E/S

Ejemplo de transferencia de E/S usando las tres técnicas

Objetivo:

Enviar un bloque a la impresora (200 caract., desde dir. 3000h). Configuración para E/S programada e interrupciones:



E/S programada:

```

; Inicialización
LDX    #$3000 ;      RegIX<-3000h (bloque almacenado a partir de la dir. 3000h)
LDAB   #200;        RegB<-200 (200 caract.)
JSR    PRBLQ

; Imprimir bloque
PRBLQ LDAA  0,X    ; RegA <- M[RegIX+0]
JSR    PRCHR
INX     ; RegX++ (apuntar a sig. caráct.)
DEC B   ; RegB-- (actualizar contador)
BNE PRBLQ ; Si RegB!=0 => enviar otro car.
RTS

; Imprimir el carácter almacenado en RegA
PRCHR TST    PSR    ; N (bit de signo) <- SR7
BPL    PRCHR ; Si N = 0 => esperar
STAA   PBR    ; BR <- A, impresora pone SR7 a 0 RTS

```

PSR y PBR son etiquetas para los puertos SR y BR, mapeados en memoria

E/S por interrupciones:

Cuando la interfaz está lista para recibir datos (impresora lista), interrumpe al 6800 por su línea IRQ#, forzándole a transferir el control a una ISR.

Puede haber varios dispositivos conectados a IRQ#.

La ISR comienza chequeando los bits “listo” de cada dispositivo, para ver cuál ha interrumpido. Se supone que la impresora tiene la mayor prioridad en este caso ⇒ es chequeada en primer lugar.

```
; Inicialización
LDX #\$3000      ; RegIX<-3000h (bloque almacenado a partir de la dir. 3000h)
STX BA           ; BA (Byte Address) <- RegIX
LDAA #200        ; RegA<-200 (bloque 200 caract.)
STAA BC          ; BC (Byte Count) <- RegA
LDAA #\$01        ; RegA <- 1 STAA PSR ; SR0 <- 1 (interfaz generará interrupciones)

BA   RMB  2      ; Reserve Memory Byte (2 bytes)
BC   RMB  1      ; Reserve Memory Byte (1 byte)

; ISR
ORG  \$5000

ISR  TST  PSR    ; Sondear impresora
BPL  POLLTERM   ; Si SR7 = 0 => saltar a Terminal
BRA  PRINT       ; Si SR7 = 1 => saltar a PRINT

POLLTERM
TST  TSR         ; Sondear otros dispositivos, comenzando por el terminal
ORG  \$FFF8
FDB  ISR         ; Form Double Byte (Vector de interrupción)

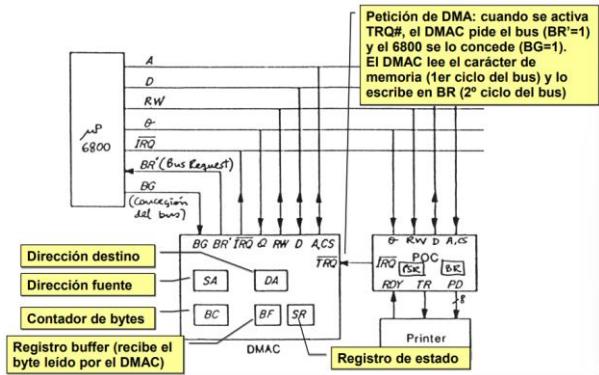
PRINT LDX  BA     ; RegIX <- Byte Address
LDAA  0,X        ; RegA <- M[RegIX+0] (sig. car.)
STAA  PBR        ; Enviar carácter
INX               ; RegX++ (apuntar a sig. caráct.)
STX  BA          ; RegB-- (actualizar contador)
DEC   BC          ; Si BC!=0 => retorno subrutina
BNE   RETURN      ; SR0 <- 0 si trans. blq. compl.

RETURN
RTI              ; Return From Interrupt

TERM ...          ; Rutinas de otros dispositivos
```

Cuando tiene lugar una interrupción, el 6800 toma la dirección de la ISR de M[FFF8h]

E/S por DMA:



- Escribir un 0 en SR7 inicia la transferencia de un bloque. Al completarse, el DMAC pone a 1 SR7 (bit "listo").
- SR0 = 1 habilita las interrupciones.
- Cuando SR7 = 1 y SR0 = 1, el DMAC genera una interrupción para indicar al 6800 el fin de la transferencia.
- El tipo de transferencia se selecciona escribiendo en SR[2:1]. SR[2:1] = 10 ⇒ transferencia de mem. a periférico.

; Inicialización del DMA

```

LDX      #$3000      ; SA <-
STX      DSA          ;      3000h (bloque en 3000h)
LDX      #200         ; BC <-
STX      DBC          ;      200 caracteres
LDX      #PBR         ; DA <-
STX      DDA          ;      dirección de BR
LDAA    #$05          ; 00000101b (Memoria->periférico,
STAA    DSR          ;      IRQ permitida)
LDAA    #$01          ; PSR <- Petición del POC
STAA    PSR          ;      al DMAC permitida

```

COMPARACIÓN DE TIEMPOS (DE PROCESADOR):

E/S por programa: suponiendo una impresora de 200 caracteres/s, el tiempo de procesador será 1 s (1 000 000 µs).

E/S por interrupciones: suponiendo un período de reloj $\theta = 1 \mu\text{s}$ ($f = 1 \text{ MHz}$), una ejecución de la ISR requiere 59 µs, además el hardware consume 9 ciclos de reloj cada vez que se atiende la interrupción. En total: 13 600 µs.

E/S por DMA: requiere dos ciclos del bus para transferir cada carácter, suponiendo un ciclo de reloj por ciclo del bus, el 6800 está inactivo 2 µs por cada carácter. Despreciando el tiempo para inicializar el DMA, 400 µs.

TEMA 6

JERARQUÍA DE MEMORIA

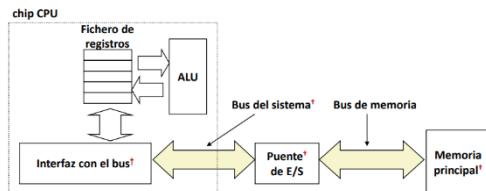
1. La abstracción de memoria (concepto de Lectura y Escritura)

Escritura: transferir datos de CPU a memoria `movq %rax, 8(%rsp)` Operación “Store”

Lectura: transferir datos de memoria a CPU `movq 8(%rsp),%rax` Operación “Load”

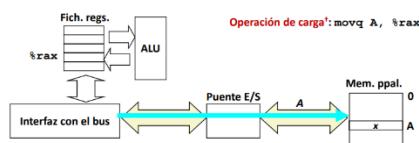
Estructura de buses CPU – Memoria (tradicional)

Un bus es un conjunto de cables en paralelo que transportan direcciones, datos, y señales de control. Típicamente a un bus se conectan varios dispositivos

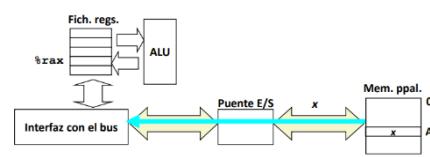


Transacción de Lectura de Memoria

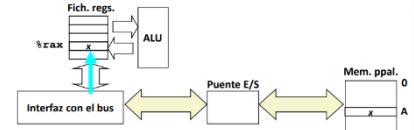
La CPU pone la dirección A en el bus de memoria



La memoria principal recibe dirección A del bus de memoria, recupera la palabra x, y la pone en el bus



La CPU lee palabra x del bus y la copia al registro %rax

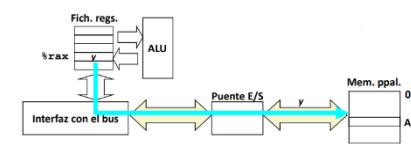


Transacción de Escritura a Memoria

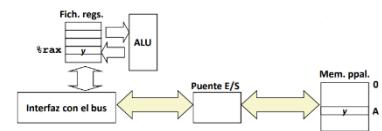
La CPU pone dirección A en el bus. La mem. ppal. la recibe y espera a que llegue la palabra de datos correspondiente



La CPU pone la palabra de datos y en el bus



La memoria principal recibe la palabra de datos y del bus y la almacena en la posición con dirección A



2. RAM: bloque constructivo de memoria principal

Tiempo de acceso: tiempo requerido para leer o escribir un dato en la memoria, se mide en ciclos o en ($n\cdot\mu\cdot m$) s.

Ancho de banda: (de la memoria de un computador)es el número de palabras a las que puede acceder el procesador (o que se pueden transferir entre el procesador y la memoria) por unidad de tiempo. Se mide en (K-M-G) B/s.

Métodos de acceso:

- Aleatorio (RAM): tiempo de acceso independiente de posición a acceder (SRAM, ROM)
- Secuencial (SAM): tiempo de acceso que depende de posición de los datos a acceder (Cinta magnética)
- Directo (semialeatorio, DASD – direct access storage device), tiempo de acceso que tiene una componente aleatoria y otra secuencial. (Discos giratorios (época en que lat. rotacional >> t. búsqueda))

Actualmente muchos dispositivos tienen 2 o más componentes acceso. DRAM: CL - T_{RCD} - T_{RP} - T_{RAS} (CAS latency, Row-Col delay, Row precharge, Row active); no es lo mismo acceder a nueva fila, a otra columna de la misma fila, a ráfaga...

Memoria de Acceso Aleatorio (RAM)

Características principales: tradicionalmente se empaqueta como un chip o incluida como parte de un chip procesador, la unidad básica almacenamiento es normalmente una celda (1 bit/celda). Múltiples chips de RAM forman una memoria.

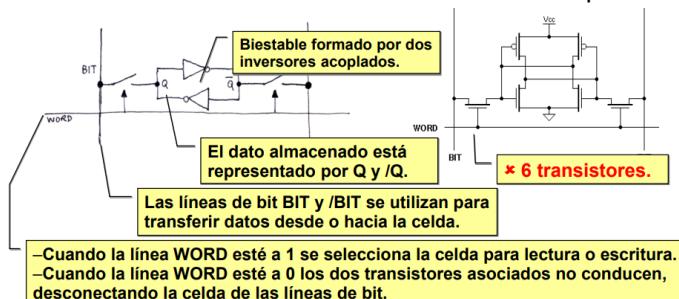
Tiene dos variedades: SRAM (RAM estática) ♣ DRAM (RAM Dinámica)

SRAM

6 transistores/bit (2 inversores (x 2 tr) + 2 puertas de paso). Mantiene el estado indefinidamente.

Celda de memoria SRAM

Estática ⇒ los datos almacenados se mantienen por un tiempo indefinido si hay alimentación.

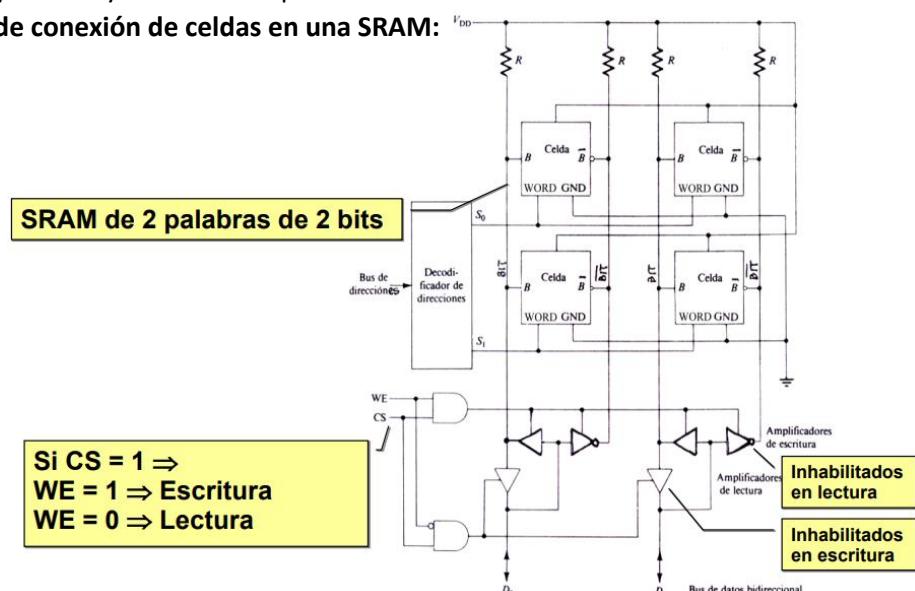


Operación de lectura: se pone WORD a 1 y Q se conecta a BIT y /Q a /BIT. Si Q = 1 (/Q = 0) la línea /BIT se pone a 0.

BIT = 1 y /BIT = 0; se lee un 1. Si Q = 0 (/Q = 1) la línea BIT se pone a 0. BIT = 0 y /BIT = 1; se lee un 0.

Lectura no destructiva y es a mayor velocidad que DRAM.

Esquema simplificado de conexión de celdas en una SRAM:

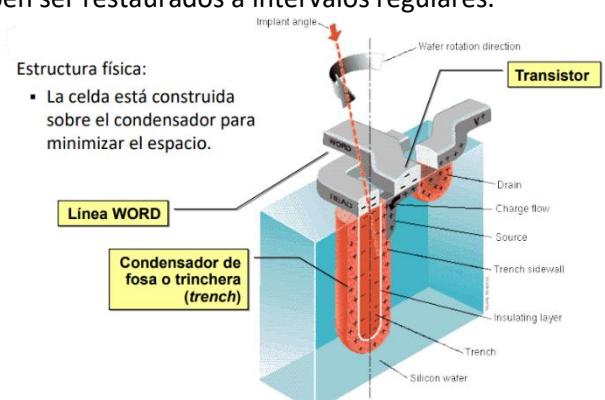
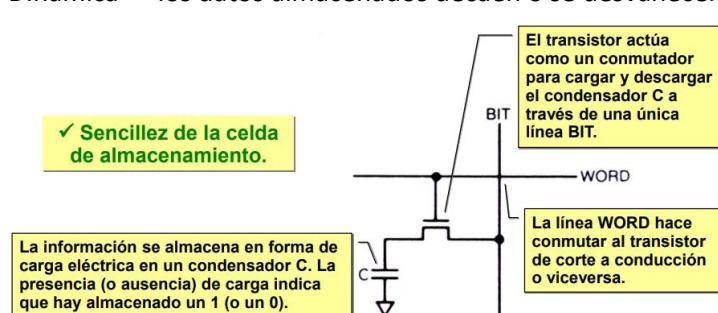


DRAM

(1 Transistor + 1 condensador)/bit; condensador orientado verticalmente. Debe refrescar estado periódicamente

Celda de memoria DRAM

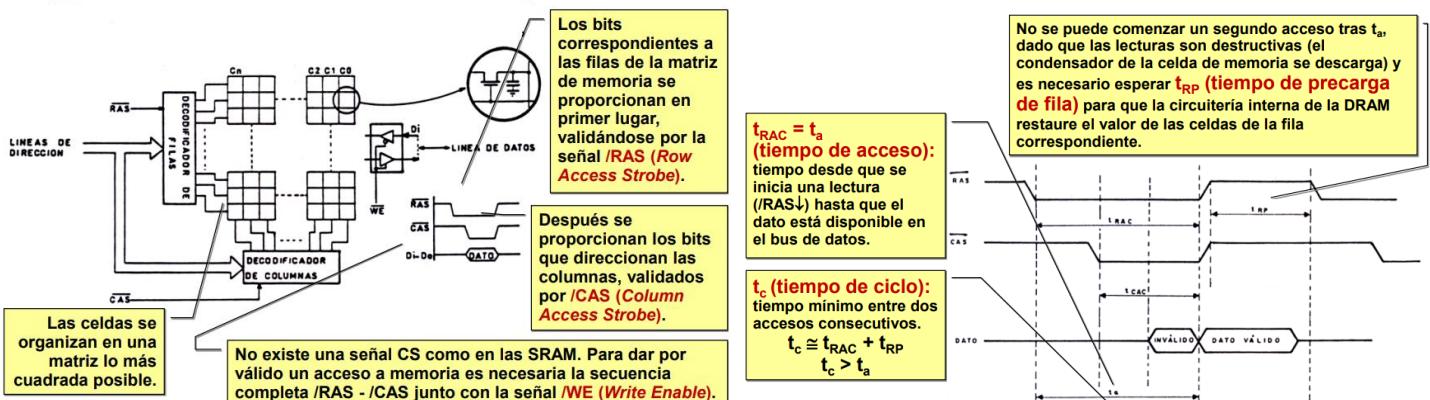
Dinámica ⇒ los datos almacenados decaen o se desvanecen y deben ser restaurados a intervalos regulares.



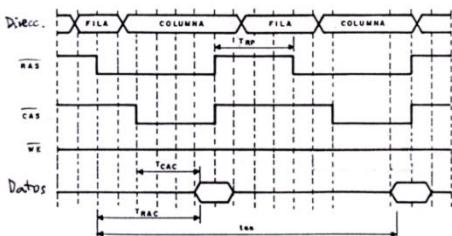
Operación de lectura: la circuitería externa convierte a BIT en una línea de salida, seleccionándose la celda con WORD = 1. Si C está cargado (= 1) se descarga a través de la línea BIT, se produce un pulso de corriente que es detectado por un amplificador de salida ("sense amplifier") y aparece un 1 en la línea de datos de salida. Si C está descargado (= 0), no se produce pulso de corriente y aparece un 0 en la línea de datos de salida. La lectura es destructiva; debe ir seguida de una escritura que restaure el estado original y es realizada automáticamente por los circuitos de control de la DRAM.

Circuitos de memoria DRAM

Capacidad elevada de los chips DRAM, las direcciones han de proporcionarse multiplexadas en el tiempo.



Acceso a dos palabras:

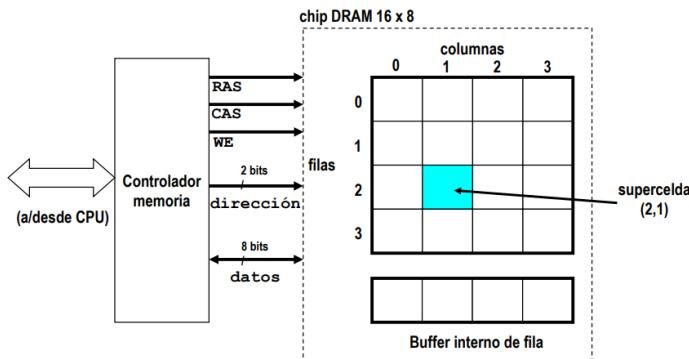


t_{aa} (tiempo de acceso a las dos palabras):
 $t_{aa} \approx t_{RAC} + t_{RP} + t_{RAC}$

$$\text{Ej.: } t_{aa} \approx 80 \text{ ns} + 60 \text{ ns} + 80 \text{ ns} = 220 \text{ ns}$$

Organización DRAM convencional

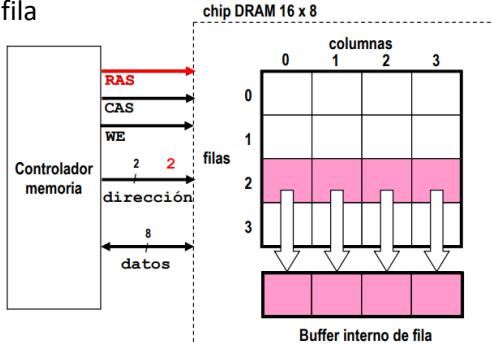
DRAM d x w : d · w bits total organizados como d superceldas de tamaño w bits.



Lectura de Supercelda DRAM

Paso 1(a): RAS (Row address strobe, comando Activate) selecciona fila 2

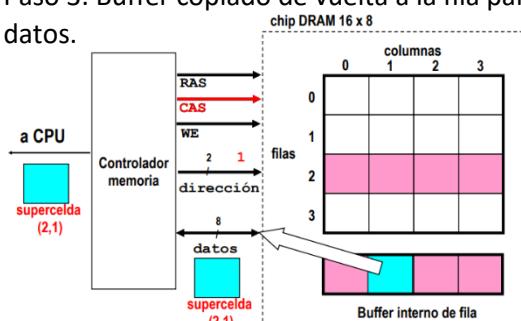
Paso 1(b): Fila 2 copiada de array DRAM a buffer de fila



Paso 2(a): CAS (Column address strobe, comando Read) selecciona columna 1

Paso 2(b): Supercelda (2,1) copiada de buffer a líneas bus datos → CPU.

Paso 3: Buffer copiado de vuelta a la fila para refrescar datos.



Resumen SRAM vs DRAM

	Trans. por bit	tiempo acceso	necesita refresco?	EDC†?	Coste	Aplicaciones
SRAM	6 ú 8	1x	No	Quizás	100x	memoria cache
DRAM	1	10x	Sí	Sí	1x	memoria principal, frame buffers†

EDC: detección y corrección de errores

La SRAM escala con la tecnología de semiconductores, pero está llegando a sus límites. El escalado de la DRAM está limitado por mínima capacidad necesaria C (condensador) razón de aspecto limita cómo de profundo se puede hacer el C, también está llegando a su límite.

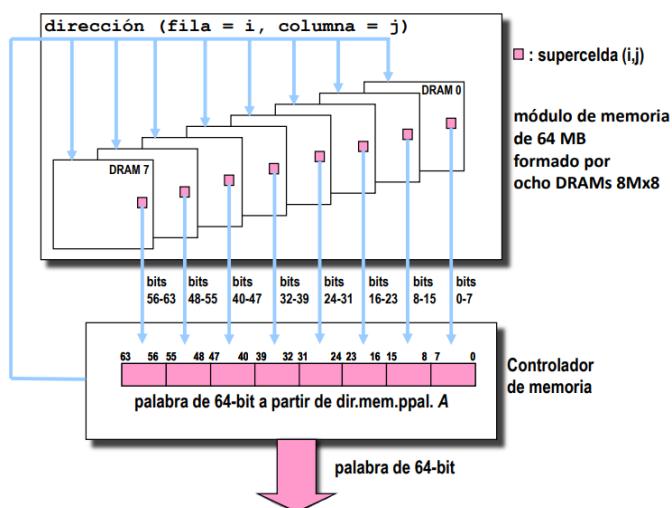
DRAMs mejoradas

El funcionamiento de La celda DRAM no ha cambiado desde su invención; comercializada por Intel en 1970 .

Núcleos DRAM con mejor lógica de interfaz y E/S más rápida:

- DRAM síncrona (SDRAM): usa una señal de reloj convencional en lugar de control asíncrono, puede aceptar un comando y transferir una palabra en cada ciclo reloj y hacer pipeline accediendo concurrentemente a distintos bancos. Reinterpreta señales RAS/CAS/WE/A10 como comando (Activate, Read...). Añade de 1 a 3 bits selección banco (BA0-2) (2,4,8 “Memory Arrays”).
- DRAM síncrona a doble velocidad datos (double data-rate, DDR SDRAM): temporización a doble flanco envía 2 bits por ciclo por pin, diferentes tipos según el tamaño de un pequeño buffer precaptación: DDR (2 bits), DDR2 (4 bits), DDR3 (8 bits), DDR4 (8 bits). DDR4 cambia formato comandos (ACT, RAS/CAS/WE=A₁₆₋₁₄, BC/AP=A_{12,10}) para 2010, estándar para mayoría sistemas sobremesa y servidor. Intel Core i7 admite SDRAM DDR3 y/o DDR4 (según modelo).

Módulos de Memoria

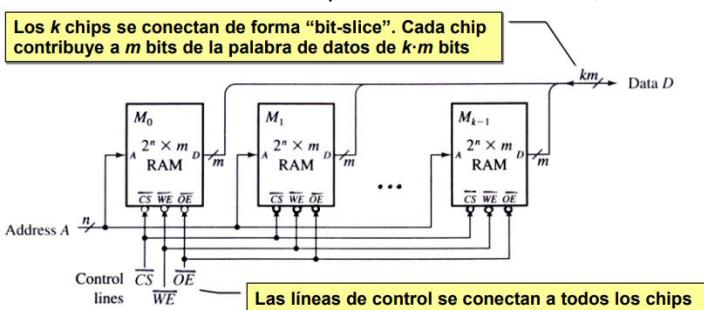


3. Configuración y diseño de memorias utilizando varios chips

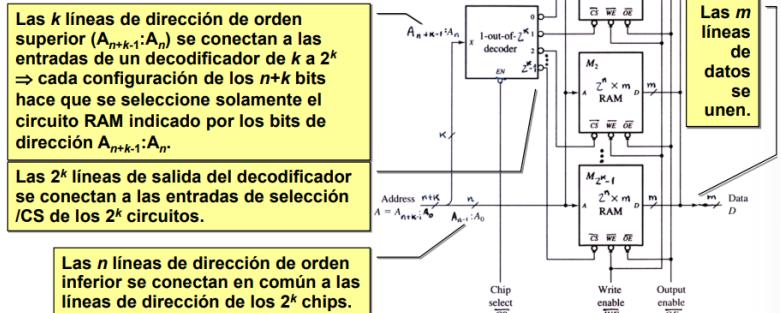
Ampliación de memoria

Problema: Construir una memoria de 2^N palabras de M bits a partir de chips de 2^n palabras de m bits.

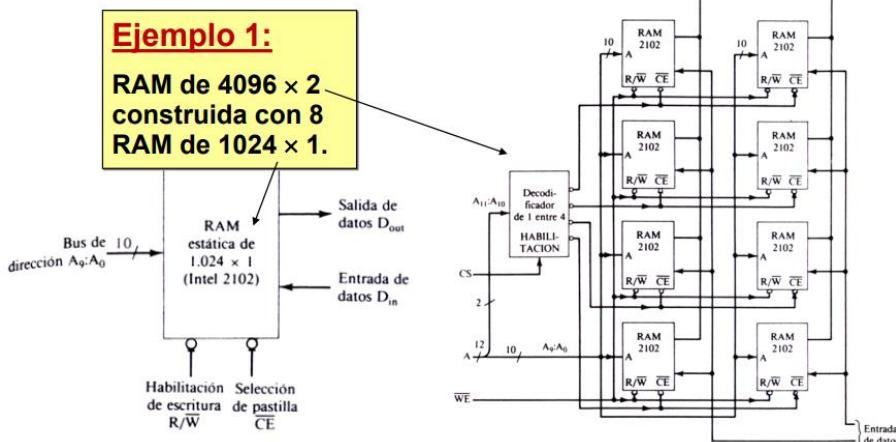
1. Incrementar el ancho de palabra de m a $k \cdot m = M$, necesitamos k circuitos de tamaño de palabra m:



2. Incrementar el número de palabras de 2^n a $2^{n+k} = 2^N$. Necesitamos 2^k circuitos de 2^n palabras y un decodificador de 1 entre 2^k .



3. Incrementar simultáneamente el número de palabras y el ancho de palabra: combina las dos técnicas anteriores.



Módulos de memoria en línea

En los 80, la memoria solía soldarse directamente en la placa madre del ordenador. Pero a medida que aumentaron los requisitos de memoria, esta técnica resultó poco factible. SIMM, DIMM, SODIMM: varios chips DRAM en una pequeña placa de circuito impreso (PCB) que calza en un conector en la placa madre. Es un método flexible para actualizar la memoria y ocupa menos espacio en la placa madre. Normalmente sólo se ampliaba el bus de datos, no el de direcciones (no había necesidad de decodificador).

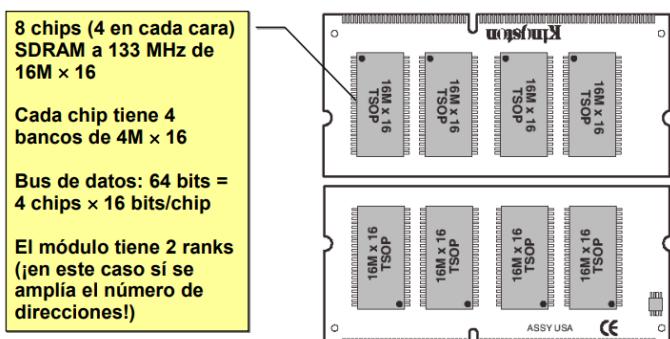
Módulos de memoria SIMM

SIMM (Single In-line Memory Module), cada contacto de una cara está conectado con el alineado en la otra cara para formar un único contacto. 80's y 90's, FPM y EDO-RAM. 30 contactos: 8 bits de datos ⇒ 4 SIMM para bus 32 bits y 12 pines dirección ⇒ 24 bits dir ⇒ 16MB máx. 72 contactos: 32 bits de datos (24 bits dir) ⇒ 64MB máx. y dos rangos (con /RAS1, /RAS3) ⇒ 128MB máx.

Módulos de memoria DIMM

DIMM (Dual In-line Memory Module), los contactos opuestos están aislados eléctricamente para formar dos contactos separados. 90's-hoy, SDRAM y DDR hasta DDR5. 64 bits de datos (ó 72 con ECC), incremento progresivo pins: 168-pin: FPM, EDO y SDRAM 184-pin: DDR 240-pin: DDR2, DDR3 288-pin: DDR4, DDR5.

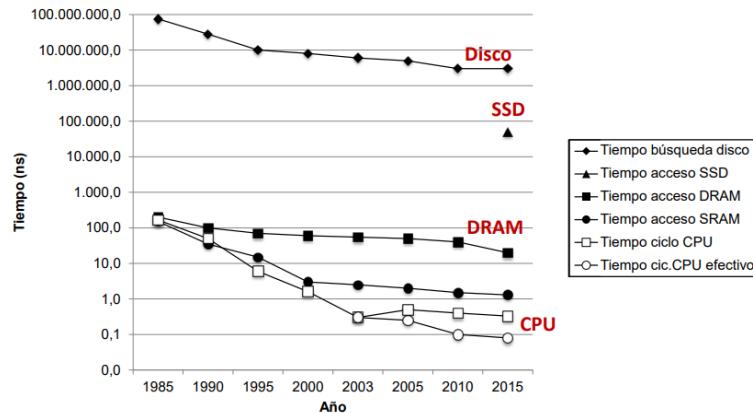
Módulos de memoria SODIMM



SODIMM (Small Outline DIMM), menor tamaño que un DIMM, menos contactos. Uso en portátiles. (100/144-pin SDR, 200-pin DDR-DDR2, 204-pin DDR3, 260-pin DDR4). Ej: SODIMM SDR (144-pin) de 256 MB (2 × 16M × 64) a 133 MHz.

4. Localidad de las referencias

La brecha entre velocidades de disco, DRAM y CPU se ensancha.



La clave para salvar esta brecha CPU-Memoria es una propiedad fundamental conocida como localidad.

PRINCIPIO DE LOCALIDAD: Los programas tienden a usar datos e instrucciones con direcciones iguales o cercanas a las que han usado recientemente.

Localidad temporal: elementos referenciados hace poco pueden ser referenciados de nuevo en un futuro próximo.

Localidad espacial: elementos con direcciones cercanas tienden a ser referenciados muy juntos en el tiempo.

Expresión matemática de localidad

Si en instante tiempo t se accede al dato/posición memoria $d(t)$...

Temporal: $d(t + n) = d(t)$ con n pequeño.

Espacial: $d(t + n) = d(t) + k$ con n, k pequeños.

Ejemplo

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Referencias a datos

Referenciar elementos array en sucesión (patrón de referencias de paso-1): espacial.

Referenciar variable sum cada iteración: temporal.

Referencias a instrucciones

Referenciar instrucciones en secuencia: espacial.

Iterar bucle repetidamente: temporal.

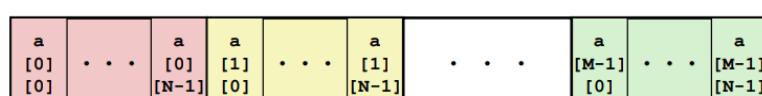
Estimaciones cualitativas de localidad

Afirmación: Ser capaz de mirar un código y sacar una idea cualitativa de su localidad es una habilidad clave para un programador profesional.

Pregunta: ¿Tiene esta función buena localidad respecto al array a ? Si

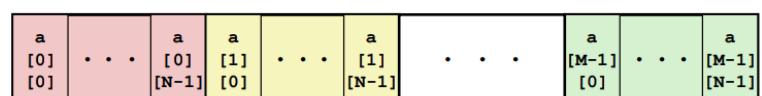
Pista: almacenamiento por filas (row-major order)

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```



Pregunta: ¿Tiene esta función buena localidad respecto al array a ? No, salvo si N es muy pequeño o M muy muy pequeño y N no muy grande

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



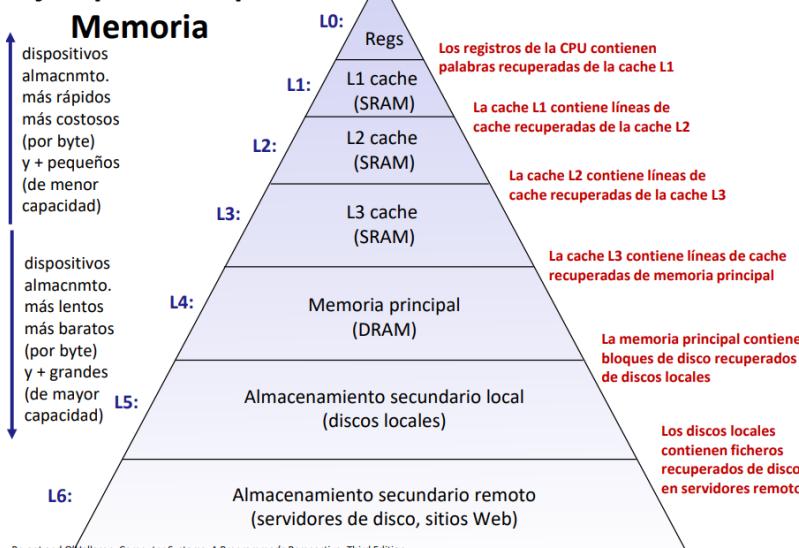
5. Jerarquía de memoria

Algunas propiedades fundamentales y perdurables del hardware y software:

- Las tecnologías de almacenamiento rápidas cuestan más por byte, tienen menor capacidad y requieren más potencia (¡calor!).
- La brecha velocidad CPU-memoria principal se está ampliando.
- Los programas bien escritos tienden a exhibir buena localidad.

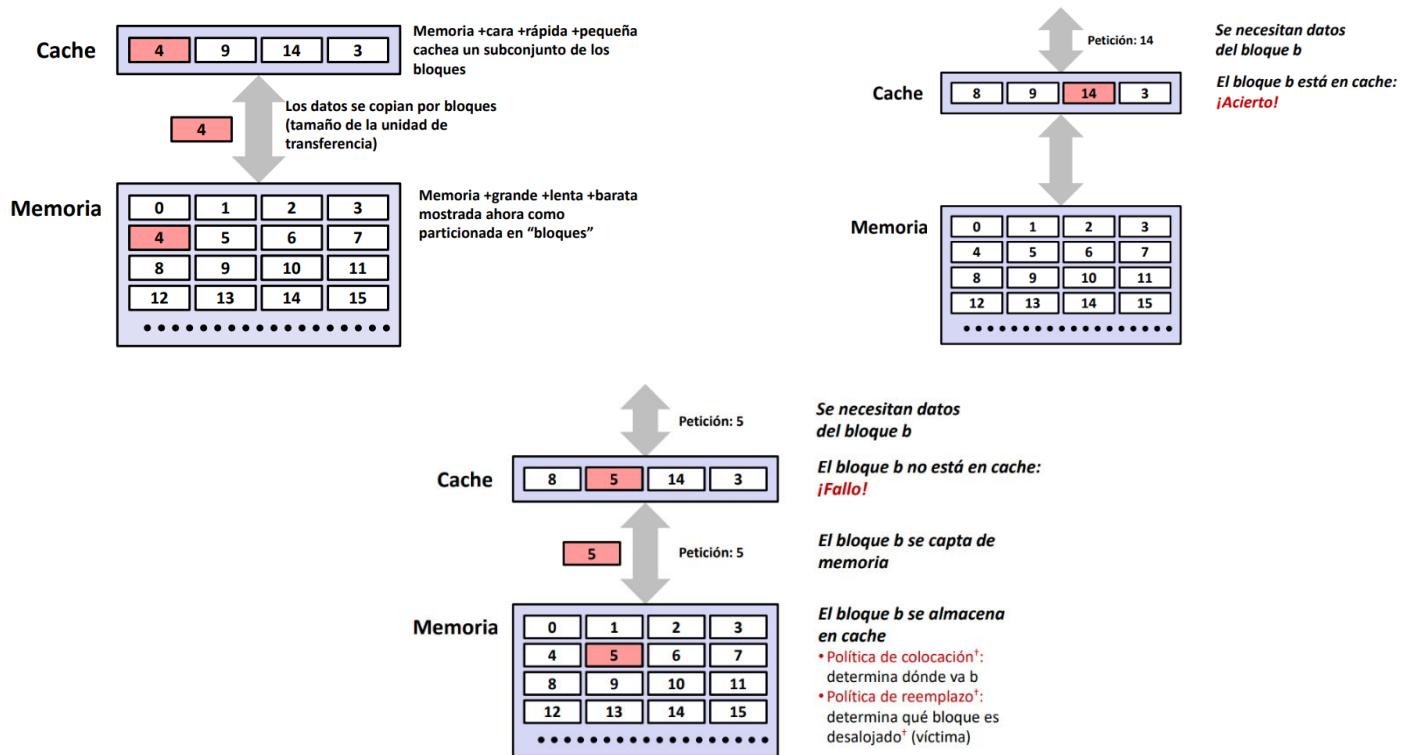
Estas propiedades fundamentales se complementan muy convenientemente. Sugieren un enfoque para organizar sistemas de memoria y almacenamiento conocido como jerarquía de memoria.

Ejemplo Jerarquía



Caches

Un dispositivo de almacenamiento más rápido y pequeño que funciona como zona de trabajo temporal para un subconjunto de los datos de otro dispositivo mayor y más lento. Idea fundamental de una jerarquía de memoria: $\forall k$, el dispositivo a nivel k (+rápido, +pequeño) sirve de cache para el dispositivo a nivel $k+1$ (+lento, +grande). Las jerarquías de memoria funcionan bien debido a la localidad; los programas suelen acceder a los datos a nivel k más a menudo que a los datos a nivel $k+1$ así, el almacenamiento a nivel $k+1$ puede ser más lento, y por tanto más barato (por bit) y más grande. Idea Brillante (ideal): la jerarquía conforma un gran conjunto de almacenamiento que cuesta como el más barato pero que proporciona datos a los programas a la velocidad del más rápido.



3 Tipos de Fallo de Cache

- Fallos en frío (obligados): ocurren porque la cache empieza vacía y esta es la primera referencia al bloque.
- Fallos por capacidad: el conjunto de bloques activos (conjunto de trabajo) es más grande que la cache.
- Fallos por conflicto: mayoría caches limitan que los bloques a nivel $k+1$ puedan ir a pequeño subconjunto (a veces unitario) de las posiciones de bloque a nivel k . *Ej:* Bloque i a nivel $k+1$ debe ir a bloque ($i \bmod 4$) a nivel k (corr. directa). Ocurren cuando cache nivel k suficientemente grande, pero a varios datos les corresponde ir al mismo bloque a nivel k . *Ej.* Referenciar bloques 0, 8, 0, 8, 0, 8, ... fallaría continuamente (ejemplo anterior con correspondencia directa).

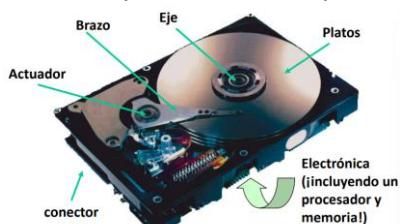
Ejemplos de cacheado en Jerarquía Memoria

Tipo de Cache	Qué se cachea?	Dónde se cachea?	Latencia (ciclos)	Gestionado por
Registros	Palabras de 4-8 B	CPU (on-chip)	0	Compilador
TLB [†]	Trad. de direcciones	TLB (on-chip)	0	MMU [†] (hardw)
cache L1	Bloques de 64 bytes	L1 (on-chip)	4	Hardware
cache L2	Bloques de 64 B	L2 (on-chip)	10	Hardware
cache L3	Bloques de 64 B	L3 (on-chip)	50	Hardware
Memoria Virtual	Páginas de 4 KB	Memoria principal	200	Hardware + SO
Buffer de disco	Partes de ficheros	Memoria principal	200	SO
cache de disco	Sectores de disco	Controladora de disco	100,000	Firmware disco
Buffer disco red	Partes de ficheros	Disco local	10,000,000	Cliente NFS [†]
cache Navegador	Páginas Web	Disco local	10,000,000	Navegador web
cache Web	Páginas Web	Discos de servidores remotos	1,000,000,000	Servidor web proxy [†]

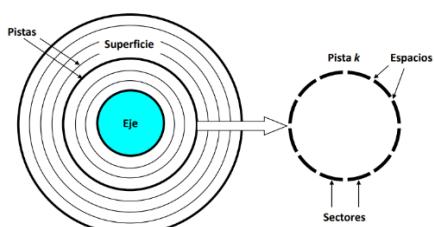
6. Tecnologías de almacenamiento, y tendencias

Discos Magnéticos: almacenamiento en medio magnético y acceso electromecánico.

Memoria No-volátil (Flash): Almacenamiento como carga persistente, implementado con estructura 3-D 100 niveles de celdas y 3 bits de datos por celda.



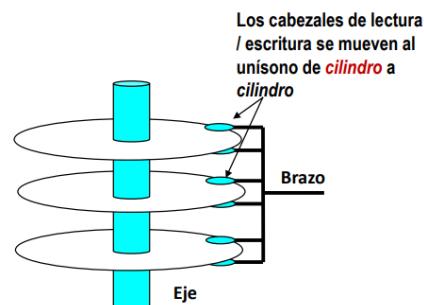
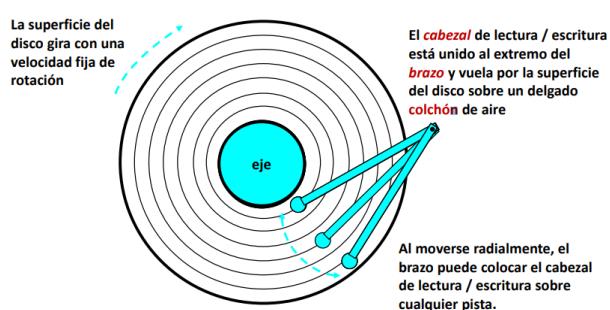
Geometría de un disco



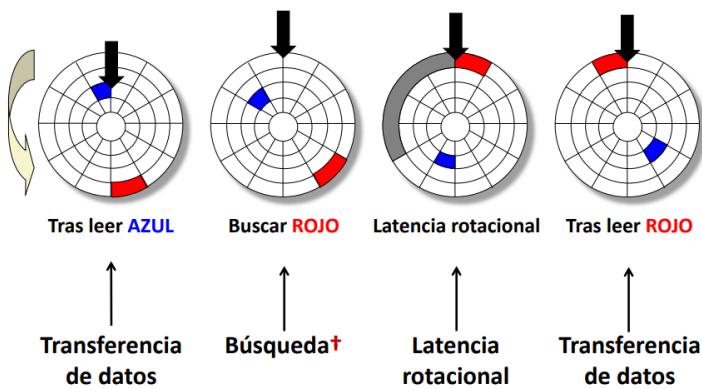
Los discos consisten en platos con dos superficies (caras), cada cara consiste en anillos concéntricos llamados pistas y cada pista consiste en sectores separados por espacios. La capacidad es el máximo número de bits que se pueden almacenar se expresa en GB o TB. Queda determinada por los siguientes factores tecnológicos: densidad de grabación (bits/pulgada): nº de bits que se pueden comprimir en un tramo de pista de 1 pulgada. Densidad de pistas (radial) (pistas/pulgada): nº de pistas que se pueden comprimir en un tramo radial de 1 pulgada y densidad superficial (bits/pulg²): producto de ambas densidades (grabación y radial).

Funcionamiento de un disco (Visto en un solo plato)

(Visto en varios platos)



Acceso a disco: componentes del tiempo de servicio



Tiempo de acceso a disco

Tiempo promedio acceso algún sector determinado, aprox:

$$T_{\text{acceso}} = T_{\text{prom b\u00f3squeda}} + T_{\text{prom rotaci\u00f3n}} + T_{\text{prom transferencia}}$$

Tiempo de b\u00f3squeda ($T_{\text{prom b\u00f3squeda}}$): tiempo para colocar cabezales sobre el cilindro que contiene el sector. Valores t\u00edpicos: 3–9 ms.

Latencia rotacional ($T_{\text{prom rotaci\u00f3n}}$): tiempo esperando a que pase bajo cabezales el primer bit del sector.

$$T_{\text{prom rotaci\u00f3n}} = 1/2 \times 1/\text{RPMs} \times 60 \text{ s}/1 \text{ min. Velocidad rotacional t\u00edpica} = 7,200 \text{ RPMs} (\Rightarrow 4.17 \text{ ms}).$$

Tiempo de transferencia ($T_{\text{prom transferencia}}$): tiempo para leer los bits del sector.

$$T_{\text{prom transferencia}} = 1/\text{RPMs} \times 1/(\#\text{ sectores/pista prom}) \times 60 \text{ s}/1 \text{ min}$$

Tiempo una rotaci\u00f3n (minutos) ; fracci\u00f3n de rotaci\u00f3n a leer

Ej: Datos: Velocidad rotacional = 7,200 RPM; tiempo b\u00f3squeda promedio = 9 ms; # sectores/pista promedio = 400.

Calcular: $T_{\text{prom rotaci\u00f3n}}$, $T_{\text{prom transferencia}}$ y T_{acceso} .

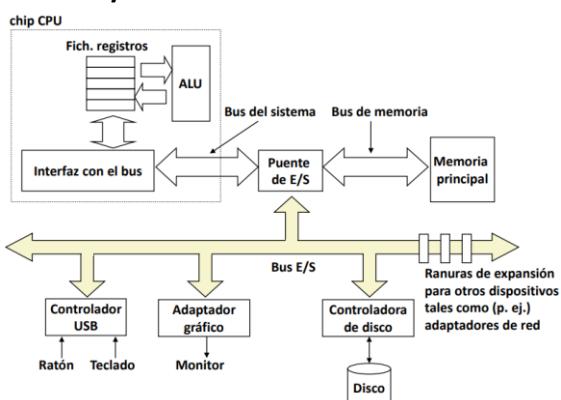
$$T_{\text{prom rotaci\u00f3n}} = 1/2 \times (60 \text{ s}/7200 \text{ RPM}) \times 1000 \text{ ms/s} = 4.17 \text{ ms}$$

$$T_{\text{prom transferencia}} = 60/7200 \times 1/400 \times 1000 \text{ ms/s} = 0.02 \text{ ms}$$

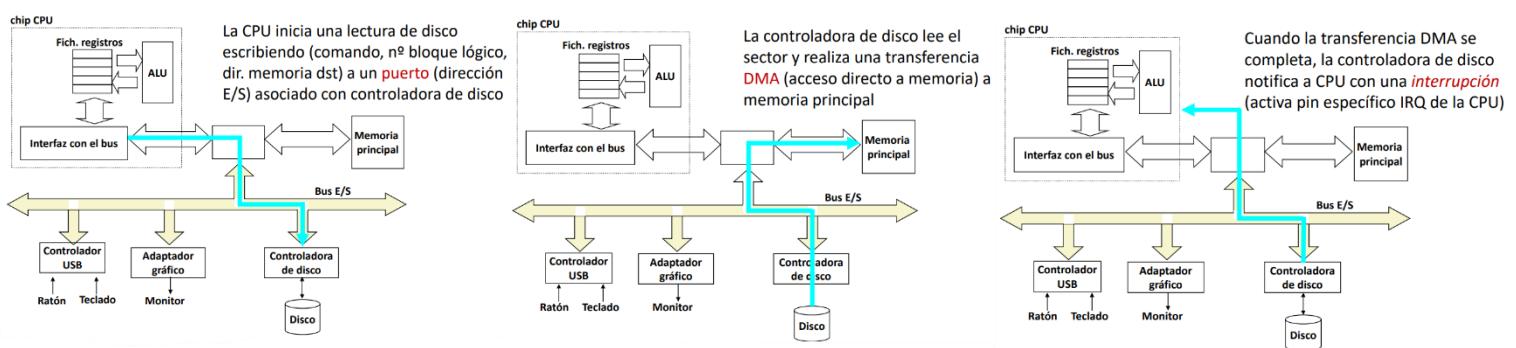
$$T_{\text{acceso}} = 9 \text{ ms} + 4.17 \text{ ms} + 0.02 \text{ ms} = 13.19 \text{ ms}$$

Puntos importantes: tiempo acceso dominado por tiempo b\u00f3squeda y latencia rotacional, el primer bit en un sector es el m\u00e1s caro (lento), el resto sale gratis. Tiempo acceso SRAM aprox. 4ns/palabra 64b, DRAM aprox. 60ns, disco es aprox. 40.000 veces m\u00e1s lento que SRAM y 2.500 veces m\u00e1s lento que DRAM.

Bus de E/S



Lectura de un sector de disco



Memorias no volátiles

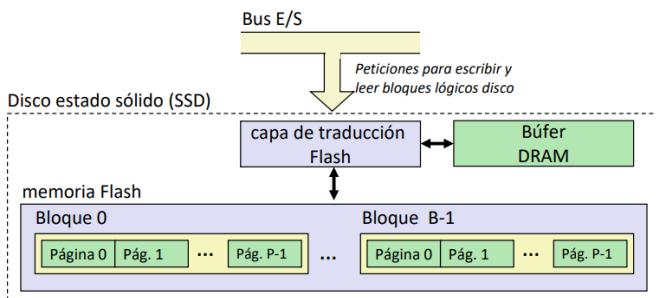
SRAM y DRAM son memorias volátiles: pierden información si se apagan.

Las memorias no volátiles retienen valores incluso si se apagan:

- Memoria de sólo lectura (ROM): programada durante su fabricación.
- PROM progr. usr. Irreversible.
- EEPROM: PROM borrible eléctricamente.
- Memorias Flash: EEPROMs capacidad borrado parcial (bloques), se desgastan tras aprox. 100.000 borrados.
- 3D XPoint[†] (Intel Optane) & NVMs emergentes; nuevos materiales.

Aplicaciones de las memorias no volátiles: programas firmware almacenados en una ROM (BIOS, controladoras de disco, tarjetas de red, aceleradores gráficos, subsistemas seguridad...), discos de estado sólido (reemplazando discos giratorios) y caches de disco.

Discos de estado sólido (SSDs)



Páginas: de 4KB a 512KB, Bloques: de 32 a 128 páginas.

Datos escritos/leídos en unidades de páginas.

Una página se puede escribir sólo tras borrar su bloque.

Un bloque se desgasta tras unas 100.000 escrituras

Prestaciones características SSD

Benchmark‡ de un Samsung 970 EVO Plus

Rendimiento lectura secuencial 2.126 MB/s Rendmto. escritura sec. 1.880 MB/s

Rendimiento lectura aleatoria 140 MB/s Rendmto. escritura aleat. 59 MB/s

Acceso secuencial mucho más rápido que aleatorio, tema omnipresente en jerarquías de memoria.

Escrituras aleatorias son especialmente lentas: borrar un bloque lleva mucho tiempo (~1ms), modificar una página de un bloque requiere que todas las otras se copien a un nuevo bloque y la capa de traducción Flash permite acumular una serie de pequeñas escrituras antes de realizar una escritura de bloque.

SSD frente a Discos giratorios

Ventajas: no partes móviles → más rápidos, menor consumo, más resistentes.

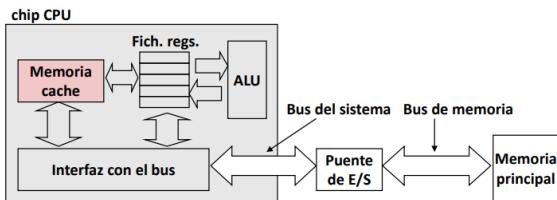
Desventajas : eventual desgaste mitigado por "lógica nivelado desgaste" en capa traducción flash; ej. Samsung 970 EVO Plus garantiza que se pueda escribir 600x la capacidad del disco antes de desgastarse, controladora migra datos para repartir/minimizar nivel desgaste. En 2019, aprox. 4x más caro por byte (que giratorios) y seguirá cayendo ese coste relativo.

Aplicaciones: reproductores MP3, smartphones, portátiles; cada vez más común en servidores y PC sobremesa.

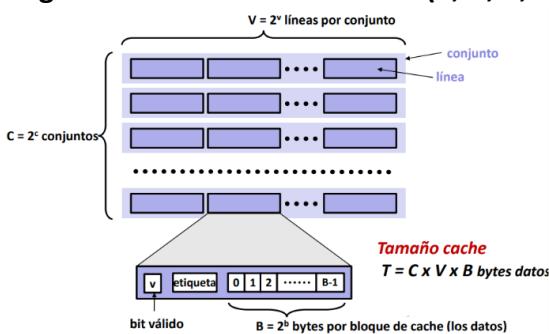
CACHE

1. Organización y Funcionamiento de la memoria cache

Las memorias cache son memorias pequeñas y rápidas basadas en SRAM gestionadas automáticamente por hardware, retiene bloques de memoria principal accedidos frecuentemente. La CPU busca los datos primero en caché, estructura típica del sistema:

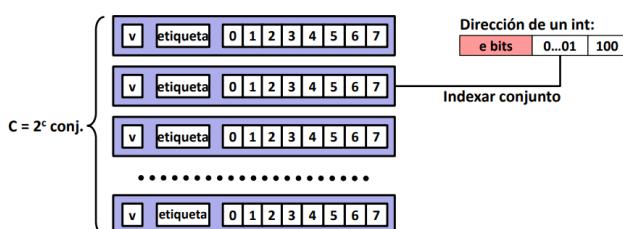


Organización General de Cache (C, V, B, m)



Cache con Correspondencia Directa (V = 1)

Correspondencia directa: Una línea por conjunto
Suponer: tamaño bloque cache B=8 bytes



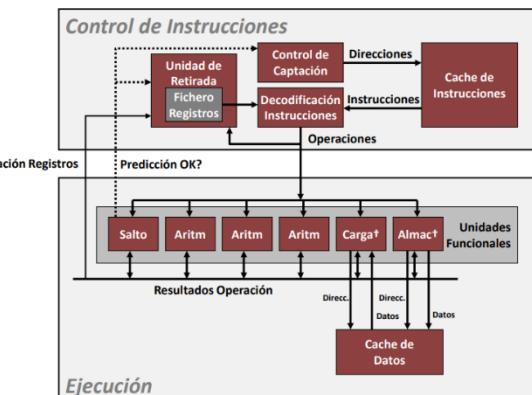
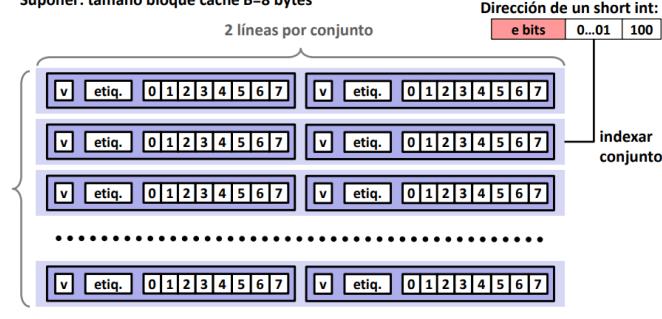
Simulación cache correspondencia directa

e=1 c=2 b=1
x xx x
direcciones 4 bits (espacio dirccnmt tam M=16 bytes)
C=4 conjuntos, V=1 vía (bloq./conj.), B=2 bytes/bloque

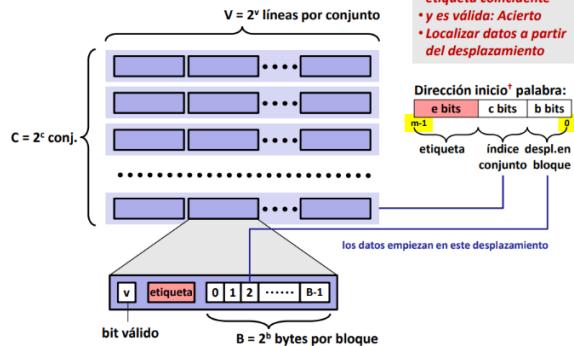
Traza de direcciones (lecturas, un byte por lectura):
0 [0000]₂, fallo
1 [0011]₂, acierto
7 [0111]₂, fallo
8 [1000]₂, fallo
0 [0000]₂, fallo

Cache Asociativa por Conjuntos de V vías (con V=2)

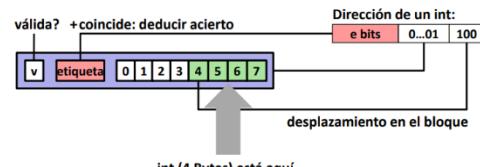
V = 2: Dos líneas por conjunto
Suponer: tamaño bloque cache B=8 bytes



Lectura de Cache



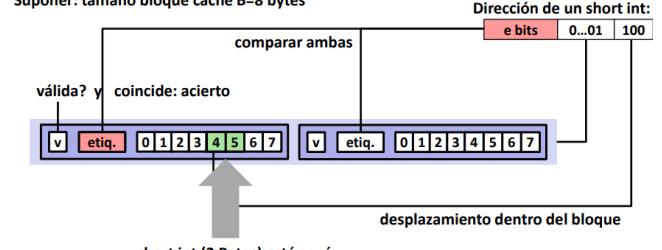
Correspondencia directa: Una línea por conjunto
Suponer: tamaño bloque cache B=8 bytes



Si etiqueta no coincide (= fallo): vieja línea desalojada y reemplazada
(nuevos datos y nueva etiqueta)

	v	Etiq.	Bloque
Conj. 0	1	0	M[0-1]
Conj. 1	0		
Conj. 2	0		
Conj. 3	1	0	M[6-7]

V = 2: Dos líneas por conjunto
Suponer: tamaño bloque cache B=8 bytes



Si ninguna coincide o no válida (= fallo):

- Se escoge una línea del conjunto para desalojar y reemplazar
- Políticas de reemplazamiento: aleatoria, menos rec.uso (LRU), ...

Simulación cache asociativa conjuntos 2-vías

e=2 c=1 b=1
 xx x x

direcciones 4 bits (M=16 bytes)
 C=2 conjuntos, V=2 vías (bloq./conj.), B=2 bytes/bloque

Traza de direcciones (lecturas, un byte por lectura):
 0 [0000]₂, fallo
 1 [0001]₂, acierto
 7 [0111]₂, fallo
 8 [1000]₂, fallo
 0 [0000]₂, acierto

	v	Etiq.	Bloque
Conj. 0	1	00	M[0-1]
	1	10	M[8-9]

	v	Etiq.	Bloque
Conj. 1	1	01	M[6-7]

¿Qué hay de las escrituras?

Múltiples copias de los datos: L1, L2, L3, Mem. principal, Disco.

¿Qué hacer en acierto escritura? Write-through (escribir inmediatamente en la memoria) o Write-back (diferir escritura hasta reemplazo de la línea), cada línea caché necesita un bit sucio (=1 si línea difiere de bloque M)

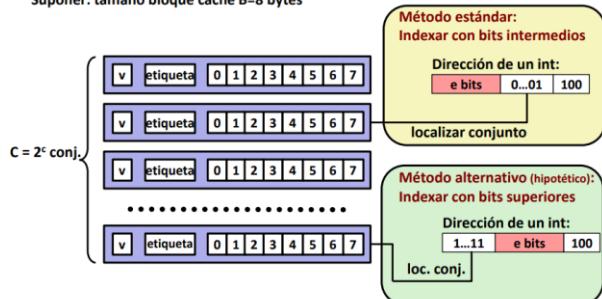
¿Qué hacer en fallo escritura? Write-allocate (cargar y actualizar línea en cache), conveniente si van a haber más escrituras a ese bloque o No-write-allocate (escribir directo a memoria, sin cargar en cache)

Usual: Write-through + No-write-allocate (Escritura directa sin Asignación en Escritura)

Write-back + Write-allocate (Post-Escritura con Asignación en Escritura)

¿Por qué indexar con los bits intermedios?

Correspondencia directa: Una línea por conjunto
 Suponer: tamaño bloques cache B=8 bytes



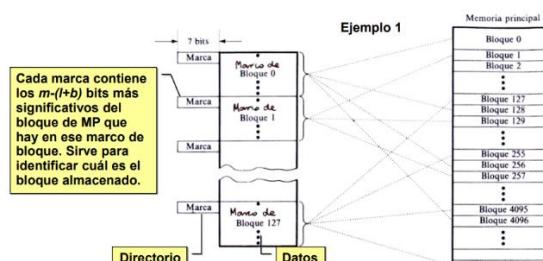
Políticas de colocación

Organización (C,V,B,m), la caché tiene $2^c \times 2^v = 2^{c+v}$ líneas. Un bloque tiene 2^b bytes. Una dirección física tiene m bits. La MP tiene 2^m bytes (2^{m-b} bloques).

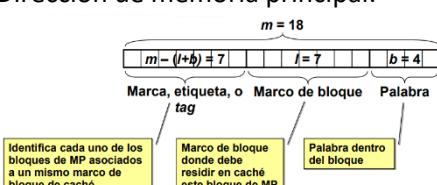
Correspondencia directa.

Bloque i de MP \Rightarrow línea i mod 2^l de cache (L líneas = $2^l = 2^{c+v} = 2^c$ con v=0). A cada línea le corresponde sólo un subconjunto de bloques de MP.

- ✓ Simplicidad y bajo coste
- ✗ Si dos o más bloques, utilizados alternativamente, corresponden al mismo marco de bloque \Rightarrow el índice de aciertos se reduce drásticamente.



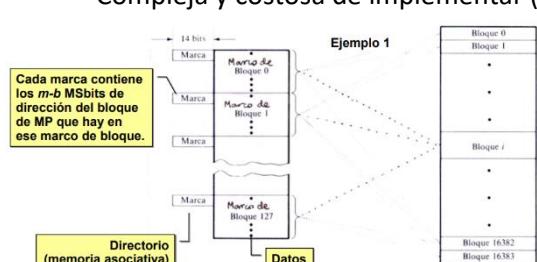
Dirección de memoria principal:



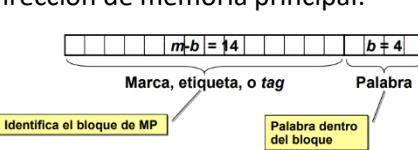
Correspondencia totalmente asociativa.

Un bloque de MP puede residir en cualquier marco de bloque de caché. Cuando se presenta una solicitud a la caché, todas las marcas se comparan simultáneamente para ver si el bloque está en la caché.

- ✓ Flexible, permite cualquier combinación de bloques de MP en la caché y elimina conflictos entre bloques.
- ✗ Compleja y costosa de implementar (por la memoria asociativa).



Dirección de memoria principal:



Correspondencia asociativa por conjuntos.

La caché se subdivide en 2^c conjuntos disjuntos, 2^v marcos de bloque/conjunto.

Bloque i de MP \Rightarrow conjunto $i \bmod 2^c$ de caché. Dentro de ese conjunto puede estar en cualquier marco de bloque.

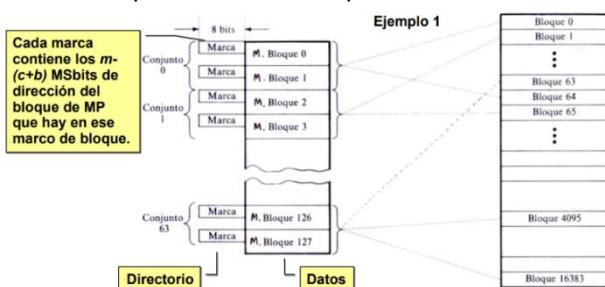
Hay dos fases en el acceso a caché: selección directa del conjunto donde puede estar ese bloque y búsqueda asociativa (dentro del conjunto) de la marca. A la correspondencia asociativa por conjuntos con 2^v marcos de bloque/conjunto también se le llama correspondencia asociativa de 2^v vías; la vía i está formada por todos los marcos de bloque de la caché que ocupan el lugar i -ésimo dentro de su conjunto.

$c = 0$ (1 conjunto) \Rightarrow correspondencia totalmente asociativa.

$c = n$ (1 marco de bloque/conjunto) \Rightarrow correspondencia directa.

$0 < c < n \Rightarrow$ se pretende reducir el coste de la totalmente asociativa manteniendo un rendimiento similar \Rightarrow es la técnica más utilizada.

- ✓ Resultados experimentales demuestran que un tamaño de conjunto de 2 a 16 marcos de bloque funciona casi tan bien como una correspondencia totalmente asociativa con un incremento de coste pequeño respecto de la correspondencia directa.



Dirección de memoria principal:



Métricas para prestaciones de cache

Tasa de Fallo: fracción de referencias a memoria no encontradas en caché (fallos / accesos) = $1 - \text{tasa de acierto}$.

Valores típicos (en porcentaje): 3-10% para L1 puede ser bastante pequeño (por ejemplo, < 1%) para L2, dependiendo del tamaño...

Tiempo en Acierto: (tiempo de acceso si acierto): tiempo para entregar una línea de cache al procesador incluye el tiempo para determinar si la línea está en cache. Valores típicos : 4 ciclos reloj para L1, 10 ciclos reloj para L2.

Penalización por Fallo: tiempo adicional requerido debido a un fallo, típicamente 50-200 ciclos para M.principal (Tendencia: aumentando!).

Enorme diferencia entre acierto y fallo podría llegar a 100x, si solo L1 y Memoria principal. ¿Verosímil que 99% acierto es doble de bueno que 97%? considerar este ejemplo simplificado: tiempo en acierto cache de 1 ciclo penalización por fallo de 100 ciclos, tiempo medio de acceso: 97% aciertos: 1 ciclo + 0.03×100 ciclos = 4 ciclos 99% aciertos: 1 ciclo + 0.01×100 ciclos = 2 ciclos. Por eso se usa "tasa de fallo" en lugar de "tasa de acierto".

2. Impacto de la cache en el rendimiento

MODELO DE EVALUACIÓN

Jerarquía de memoria

Parámetros que caracterizan cada nivel i: los dispositivos de almacenamiento se caracterizan por:

- Tiempo de acceso (t_i): desde que se inicia una lectura hasta que llega la palabra deseada.
- Tamaño de la memoria (s_i): nº bytes, palabras, sectores... que se pueden almacenar.
- Coste por bit o por byte (c_i).
- Ancho de banda (b_i): velocidad a la que se transfiere información desde un dispositivo.
- Unidad de transferencia (x_i): tamaño de la unidad de información que se transfiere entre el nivel i y el $i+1$.

Se verifica que:

$$\begin{aligned} t_i &< t_{i+1} \\ s_i &< s_{i+1} \\ c_i &> c_{i+1} \\ b_i &> b_{i+1} \\ x_i &< x_{i+1} \end{aligned}$$

Propiedad de inclusión: $M_1 \subset M_2 \subset M_3 \subset \dots \subset M_n$. Si una palabra se encuentra en $M_i \Rightarrow$ copias de esa palabra

también se encuentran en $M_{i+1}, M_{i+2}, \dots, M_n$. Sin embargo, una palabra almacenada en M_{i+1} puede no estar en M_i , y si es así, tampoco estará en $M_{i-1}, M_{i-2}, \dots, M_1$.

Modelo de evaluación del rendimiento de una jerarquía de memoria

Se asume que caches son inclusivas ($\sum_{i=1}^j a_i = A_j$) y que el compilador no reutiliza registros ($A_0 = 0$).

Tasa de aciertos A_i (hit ratio). Porcentaje de información buscada que está en el nivel i , en una jerarquía con n niveles: $A_0 = 0$ y $A_n = 1$. A_i depende de: capacidad del nivel i (S_i), granularidad de la transferencia de información y estrategia de administración de memoria.

Tasa de fallos F_i : (miss ratio), $F_i = 1 - A_i$, $i=0, \dots, n$. $F_0 = 1$ y $F_n = 0$.

Tasa de aciertos específica del nivel i: a_i : frecuencia de accesos con primer éxito al nivel i, probabilidad de acceder con éxito a una información en nivel i y que esa información no se encuentre en los niveles 0 a i-1. $\sum_{i=1}^n a_i = 1$. Dado que la información de M_j está también en M_k ; $k > j$: $a_i = A_i - A_{i-1}$, $i = 1, \dots, n$, $a_1 = A_1$ y $a_n = 1 - A_{n-1}$.

Los **objetivos** al diseñar una memoria con n niveles son: obtener un rendimiento cercano al de la memoria M_1 (la más rápida) y obtener un coste por bit cercano al de la memoria M_n (la más barata).

1. Rendimiento: Cuantificable con el tiempo medio de acceso a la jerarquía (T).

Tiempo de acceso efectivo al nivel i-ésimo de la jerarquía (T_i):

$$T_i = \sum_{j=1}^i t_j \quad \bar{T} = \sum_{i=1}^n a_i T_i \quad a_i = A_i - A_{i-1}$$

Sustituyendo...

$$\bar{T} = \sum_{i=1}^n [(A_i - A_{i-1}) \sum_{j=1}^i t_j] = \sum_{i=1}^n (A_i - A_{i-1}) t_i = \sum_{i=1}^n (1 - A_{i-1}) t_i = \sum_{i=1}^n F_{i-1} t_i$$

2. Coste por bit: el coste por bit promedio $c(n)$ de un sistema de n niveles es:

$$c(n) = \frac{\sum_{i=1}^n c_i s_i}{\sum_{i=1}^n s_i} + c_0$$

coste total
coste de interconexión entre niveles
tamaño total

Modelo de evaluación del rendimiento de una memoria caché

Modelo aplicado a un sistema de solo 2 niveles (memoria cache y memoria principal).

$t_1 = t_c$: tiempo de acceso de la memoria cache (específico).

$a_1 = A_1 = A$: tasa de acierto (hit ratio) de la memoria cache.

$t_2 = t_m$: tiempo de acceso a la memoria principal (específico).

Tiempo medio de acceso: $\bar{T} = At_C + (1 - A)(t_C + t_m)$

γ : razón entre los tiempos de acceso a la MP y a la cache. $\gamma = \frac{t_m}{t_c}$

Eficiencia de un sistema que emplea memoria cache: $E = \frac{t_C}{\bar{T}} = \frac{1}{1 + \gamma(1 - A)}$; $0 < E \leq 1$.

E máxima cuando $A=1$ (todas las referencias en cache).

$\gamma \uparrow \Rightarrow E \downarrow$; $A \downarrow \Rightarrow E \downarrow$. Interesa γ baja y A alta.

Caché separada/unificada

Es común particionar la caché en dos módulos diferentes:

Caché de datos

La localidad de los datos no es tan buena como la de las instrucciones, es menos eficiente. Es más compleja por la posibilidad de modificación de los datos; por lo que no la admiten muchos sistemas.

Caché de instrucciones

Es fácil que bucles y pequeñas rutinas entren totalmente en caché, permitiendo su ejecución sin necesidad de acceder a MP, se simplifica la caché, ya que es habitual que se prohíba escribir en las localizaciones donde se encuentran las instrucciones \Rightarrow caché de sólo lectura.

- ✓ Permite emitir direcciones de instrucción y dato a la vez, doblando el ancho de banda entre caché y procesador.
- ✓ Se puede optimizar cada caché por separado: diferentes capacidades, tamaños de bloque, asociatividades...

Las caches de instrucciones tienen menor frecuencia de fallos que las de datos.

¿Cuál tiene una frecuencia de fallos menos: una caché de instrucciones de 16 KB + una caché de datos de 16 KB o una caché unificada de 32 KB?

Frecuencia de fallos para la caché particionada:

$$53\% \cdot 3,6\% + 47\% \cdot 5,3\% = 4,4\%$$

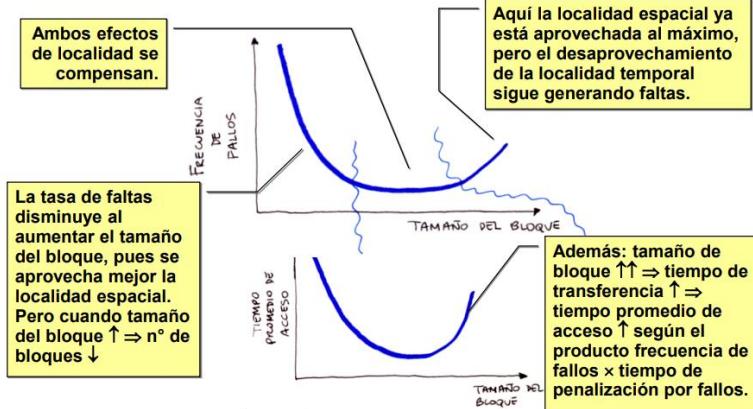
Una caché unificada de 32 KB tiene una frecuencia de fallos similar (4,3%).

Tamaño	Instrucción sólo	Sólo datos	Unificada
0,25 KB	22,2 %	26,8 %	28,6 %
0,50 KB	17,9 %	20,9 %	23,9 %
1 KB	14,3 %	16,0 %	19,0 %
2 KB	11,6 %	11,8 %	14,9 %
4 KB	8,6 %	8,7 %	11,2 %
8 KB	5,8 %	6,8 %	8,3 %
16 KB	3,6 %	5,3 %	5,9 %
32 KB	2,2 %	4,0 %	4,3 %
64 KB	1,4 %	2,8 %	2,9 %
128 KB	1,0 %	2,1 %	1,9 %
256 KB	0,9 %	1,9 %	1,6 %

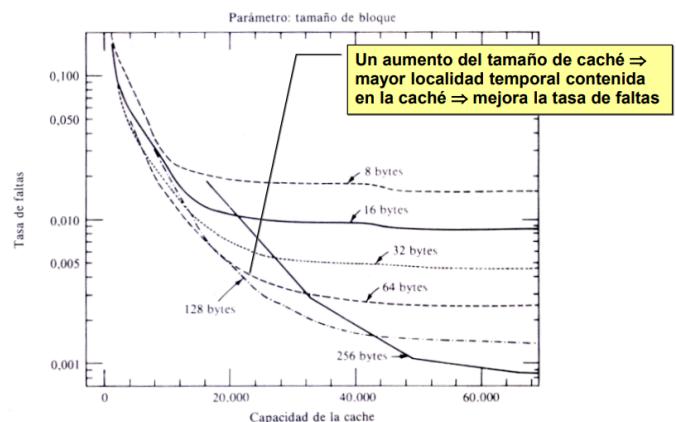
Frecuencia de fallos para caches de distintos tamaños de sólo datos, sólo instrucciones, y unificadas. Los datos son para una cache asociativa de 2 vías utilizando reemplazo LRU con bloques de 16 bytes para un promedio de trazas de usuario/sistema en la VAX-11 y trazas de sistema en el IBM 370 [Hill 1987]. El porcentaje de referencias a instrucciones en estas trazas es aproximadamente del 53 por 100.

Tamaño caché

Tamaños del bloque y de la caché para un tamaño de caché fijo:



tamaño de bloque fijo:



LA MONTAÑA DE MEMORIA

Rendimiento de lectura (ancho de banda de lectura): número de bytes leídos de memoria por segundo (MB/s).

Montaña de memoria: rendimiento de lectura medido en función de la localidad espacial y temporal, es la manera compacta de caracterizar el rendimiento del sistema de memoria.

Función test

```
long data[MAXELEMS]; /* Array global a recorrer */

/* test - Iterar sobre los primeros "elems" elementos
 *       del array "data" con paso de "stride",
 *       usando desenrollado de bucles x4
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combinar 4 elementos a la vez */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Terminar con los elementos restantes */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}                                mountain/mountain.c
```

Lamar a `test()` con muchas combinaciones de `elems` y `stride`

Para cada `elems` y `stride`:

1. Lamar una vez a `test()` para calentar las caches
2. Lamar a `test()` otra vez y medir el rendimiento de lectura (MB/s)

+ for (...) ; i<=limit; ...) :

3. Programación de código aprovechando la cache

Para escribir código amigable con cache:

- Hacer que el caso más común vaya rápido, centrarse en los bucles internos de las funciones básicas.
- Minimizar los fallos en los bucles internos: referencias repetidas a variables son buenas (localidad temporal) y patrones de referencia de paso-1 son buenos (localidad espacial).

Idea clave: nuestra noción cualitativa de localidad se cuantifica a través de nuestra comprensión de las memorias caché.