UNIVERSIDAD DE GRANADA

Estructura de Computadores

Temario de la asignatura

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Estructura de Computadores

Ismael Sallami Moreno

https://elblogdeismael.github.io

Índice general

Capítulo 1

Introducción

Tema 1 del programa. Las unidades funcionales de un computador, cómo se relacionan y qué ocurre en el intervalo que va desde que una instrucción se lee hasta que su efecto es visible.

1.1 Unidades funcionales

Un computador se descompone en cinco bloques. La descomposición es la de von Neumann y sigue describiendo cualquier máquina de propósito general, por debajo de las variaciones de implementación.

Unidad	Función
Unidad aritmético-lógica	opera sobre los datos
Unidad de control	interpreta las instrucciones y gobierna al resto
Memoria	guarda instrucciones y datos
Entrada	introduce información desde el exterior
Salida	entrega información al exterior

Las dos primeras forman el **procesador**. La memoria y el procesador se comunican por buses; la entrada y la salida, a través de controladores que el tema 5 desarrolla.

1.1.1 El principio de programa almacenado

La decisión que define la arquitectura: **instrucciones y datos comparten la misma memoria y el mismo formato**. Un programa es un dato más, lo que permite que un programa genere otro programa, que un compilador escriba código y que el sistema operativo cargue un ejecutable como quien copia bytes.

La alternativa, la **arquitectura Harvard**, separa la memoria de instrucciones de la de datos. Sobrevive en dos sitios muy concretos, y por razones distintas:

- En microcontroladores, porque el programa vive en memoria no volátil y los datos en RAM, y la separación es física de todas formas.
- Dentro de los procesadores modernos, en el primer nivel de caché, que sí está dividido en caché de instrucciones y caché de datos. Hacia fuera la máquina sigue siendo von Neumann; hacia dentro, Harvard.

La contrapartida del programa almacenado es el **cuello de botella de von Neumann**: instrucciones y datos compiten por el mismo camino hacia la memoria. Casi todo lo que aparece en los temas 4 y 6 —segmentación, cachés, prebúsqueda— existe para mitigarlo.

1.2 Niveles de abstracción

Un computador se puede describir en varios niveles, cada uno con su vocabulario:

Nivel	Qué se ve	Quién trabaja aquí
Lenguaje de alto nivel	tipos, funciones, objetos	el programador de aplicaciones
Ensamblador	instrucciones y registros con nombre	el compilador y esta asignatura
Lenguaje máquina	secuencias de bits	el procesador
Microarquitectura	camino de datos, unidad de control	el diseñador del procesador
Lógica digital	puertas y biestables	el diseñador del circuito

La frontera que importa aquí es la del nivel de lenguaje máquina, porque es donde se define el contrato entre hardware y software: la **arquitectura del repertorio de instrucciones**, o ISA.

La ISA es lo que un programa ve; la microarquitectura es cómo se implementa. Dos procesadores con la misma ISA ejecutan el mismo binario aunque por dentro no se parezcan en nada: uno puede ejecutar en orden y otro reordenar cien instrucciones. Esa separación es lo que permite que un binario compilado en 2005 siga funcionando, y es la razón económica por la que las ISA cambian tan despacio.

1.3 Conceptos básicos de funcionamiento

1.3.1 El ciclo de instrucción

El procesador repite indefinidamente la misma secuencia:

1. **Búsqueda.** Se lee de memoria la instrucción cuya dirección está en el contador de programa, y este se incrementa.
2. **Decodificación.** Se interpretan los campos de la instrucción: qué operación es, qué operandos usa, dónde deja el resultado.
3. **Lectura de operandos.** Del banco de registros o de memoria.
4. **Ejecución.** La ALU opera, o se calcula una dirección.
5. **Escritura del resultado.** En un registro o en memoria.

El incremento del contador de programa en el paso 1, **antes** de ejecutar, no es un detalle de orden: es lo que hace que una instrucción de salto relativo se calcule respecto a la instrucción siguiente, y explica los desplazamientos que aparecen al desensamblar.

Al final del ciclo el procesador comprueba si hay una interrupción pendiente. Es el único punto donde puede atenderla sin dejar una instrucción a medias, y por eso una instrucción muy larga aumenta la latencia de interrupción.

1.3.2 Registros

La memoria más rápida y más escasa. Los que toda máquina tiene, con nombre propio o sin él:

Registro	Contenido
Contador de programa (PC)	dirección de la instrucción siguiente

Registro	Contenido
Registro de instrucción (IR)	la instrucción que se está ejecutando
Registro de direcciones de memoria (MAR)	dirección del acceso en curso
Registro de datos de memoria (MDR)	dato que se lee o se escribe
Puntero de pila (SP)	cima de la pila
Registros generales	operandos y resultados
Registro de estado	banderas de cero, signo, acarreo y desbordamiento

El **registro de estado** es la pieza sobre la que se construye todo el control de flujo: una comparación no salta, solo deja banderas; el salto condicional las lee. Separar las dos cosas permite que una comparación sirva a varios saltos y que el compilador reordene el código entre ambas.

1.3.3 Medida de prestaciones

El tiempo de ejecución de un programa es el producto de tres factores:

$$T = N \times \mathrm{CPI} \times T_{ciclo}$$

donde N es el número de instrucciones ejecutadas, CPI los ciclos por instrucción y T_{ciclo} el periodo de reloj. Cada factor lo determina alguien distinto:

Factor	Lo fija
N	la ISA, el compilador y el algoritmo
CPI	la microarquitectura y la mezcla de instrucciones
T_{ciclo}	la tecnología del circuito y la profundidad del cauce

De aquí sale el error clásico de comparar máquinas por la frecuencia de reloj: un procesador a 3 GHz con CPI 2 es más lento que uno a 2 GHz con CPI 1. Y el otro error, comparar por MIPS: la medida ignora qué hace cada instrucción, así que favorece a la ISA que necesita más instrucciones para el mismo trabajo.

1.3.4 La ley de Amdahl

Cuánto mejora el conjunto al acelerar una parte. Si una fracción f del tiempo se acelera un factor k, la ganancia global es

$$S = \frac{1}{(1-f) + \dfrac{f}{k}}$$

El límite cuando $k \to \infty$ es $1/(1-f)$. Con $f = 0{,}9$, hacer esa parte infinitamente rápida solo multiplica por 10 el rendimiento total. Es el argumento cuantitativo de por qué se optimiza lo que más se ejecuta y no lo que más llama la atención, y reaparece en el tema 6 al medir el efecto de la caché.

1.3.5 Tipos de arquitectura del repertorio

	CISC	RISC
Número de instrucciones	grande, y complejas	reducido y regular
Formato	longitud variable	longitud fija
Acceso a memoria	casi cualquier instrucción	solo carga y almacenamiento
Modos de direccionamiento	muchos	pocos
Registros	pocos, especializados	muchos, generales
Ejemplos	x86, VAX, 68000	RISC-V, ARM, MIPS

La comparación limpia entre las dos familias dejó de existir hace décadas: los x86 actuales decodifican sus instrucciones complejas en microoperaciones de tipo RISC y ejecutan esas. Lo que sobrevive es la diferencia visible al programar en ensamblador, y esa es la que interesa en el tema siguiente. El desarrollo completo de esta comparación está en Stallings, 2022 y en Patterson y Hennessy, 2021.

1.3.6 Buses

Un bus es un conjunto de líneas compartidas por varias unidades. Tres grupos:

Bus	Qué transporta	Sentido
Direcciones	la posición a la que se accede	del procesador hacia fuera
Datos	el contenido	bidireccional
Control	lectura o escritura, sincronización, peticiones	mixto

El ancho del bus de direcciones fija el espacio direccionable: con 32 líneas, 4 GiB. El ancho del bus de datos fija cuánto se transfiere por operación.

Como el bus es compartido, hace falta **arbitraje** cuando varias unidades quieren usarlo. Puede ser centralizado —un árbitro decide— o distribuido, y la política va desde la prioridad fija hasta el turno rotatorio.

Y una transferencia puede ser **síncrona**, gobernada por un reloj común, o **asíncrona**, con un protocolo de acuse en el que el emisor espera confirmación. La síncrona es más rápida y obliga a que todos los dispositivos vayan al ritmo del más lento; la asíncrona se adapta a dispositivos de velocidades distintas a costa de las señales de acuse.

El bus compartido clásico ha desaparecido de las máquinas de propósito general. PCI Express, que lo sustituyó, no es un bus sino una red de enlaces punto a punto conmutados: no hay arbitraje porque no hay medio compartido. El vocabulario —«bus PCI»— se quedó, y conviene no dejarse llevar por él.

Capítulo 2

Representación de programas a nivel máquina

Tema 2 del programa, y el más extenso. Cómo se codifica un programa en instrucciones, qué ofrece la arquitectura del repertorio y cómo se traducen las construcciones de un lenguaje de alto nivel: control de flujo, procedimientos, vectores y estructuras.

2.1 Codificación de programas

Una instrucción máquina es una palabra de bits con campos. El primero es el **código de operación**; el resto identifica los operandos.

```
|<-- codigo de operacion -->|<-- operando 1 -->|<-- operando 2 -->|
```

Dos decisiones de formato:

- **Longitud fija.** Todas las instrucciones ocupan lo mismo. La decodificación es trivial y se puede empezar la siguiente sin saber qué es la actual, que es justo lo que la segmentación necesita. A cambio, se desperdician bits.
- **Longitud variable.** Cada instrucción ocupa lo que necesita. El código es más denso y la decodificación deja de ser posicional: hay que decodificar una instrucción para saber dónde empieza la siguiente.

x86 llega a instrucciones de quince bytes; RISC-V usa cuatro, con una extensión comprimida de dos. La densidad importaba cuando la memoria era cara; hoy importa por otra razón, la ocupación de la caché de instrucciones.

2.1.1 El número de operandos

Direcciones	Forma	Ejemplo conceptual
Tres	destino, fuente 1, fuente 2	`add r1, r2, r3`
Dos	destino y fuente, el destino se sobrescribe	`add r1, r2`
Una	acumulador implícito	`add r1`
Cero	operandos en una pila	`add`

x86 usa dos direcciones, y de ahí que `add %rbx, %rax` destruya el valor anterior de `%rax`. RISC-V usa tres, y por eso no hace falta copiar antes de operar. La diferencia se nota al leer código desensamblado: en x86 abundan las copias que en RISC-V no existen.

2.1.2 Ordenación de bytes

Un dato de varios bytes se puede guardar de dos formas:

	Extremo menor (*little-endian*)	Extremo mayor (*big-endian*)
En la dirección más baja	el byte menos significativo	el byte más significativo
`0x12345678` en `0x100`	`78 56 34 12`	`12 34 56 78`
Ejemplos	x86, RISC-V, ARM por omisión	SPARC, redes TCP/IP

No es una diferencia académica: leer un dato escrito por una máquina de la otra convención lo entrega con los bytes al revés. Por eso los protocolos de red fijan un orden —el mayor, llamado orden de red— y las bibliotecas ofrecen funciones de conversión.

También explica un efecto visible al depurar: en una máquina de extremo menor, un volcado hexadecimal de memoria muestra los enteros «al revés» respecto a como se escriben.

2.1.3 Alineamiento

Un acceso está alineado si la dirección es múltiplo del tamaño del dato. Un acceso no alineado cuesta más —puede necesitar dos accesos a memoria— y en algunas arquitecturas produce una excepción.

El compilador alinea insertando relleno, y eso tiene una consecuencia práctica en las estructuras que aparece más abajo: el orden de los campos cambia el tamaño.

2.2 Arquitectura del repertorio

2.2.1 Modos de direccionamiento

Cómo se especifica dónde está un operando:

Modo	Notación	Operando efectivo
Inmediato	`$5`	la constante, dentro de la instrucción
Registro	`%rax`	el contenido del registro
Directo	`dir`	el contenido de esa dirección
Indirecto de registro	`(%rax)`	lo que hay en la dirección que contiene el registro
Con desplazamiento	`8(%rax)`	dirección `%rax + 8`
Indexado con escala	`(%rax,%rbx,4)`	dirección `%rax + 4·%rbx`
Relativo al PC	`etiqueta`	dirección calculada respecto al contador de programa

Los dos últimos no son adornos. El indexado con escala existe **porque existen los vectores**: el factor de escala es el tamaño del elemento, así que `base + índice · tamaño` se calcula en la propia instrucción. Y el relativo al PC es lo que permite código reubicable: un salto no nombra una dirección absoluta, sino una distancia, y el código funciona cargado en cualquier sitio.

2.2.2 Registros de x86-64

La asignatura trabaja sobre x86-64 en Linux, que es lo que las prácticas usan. Dieciséis registros generales de 64 bits, con nombres heredados de la evolución de la arquitectura:

64 bits	32	16	8 bajos	Uso convenido
`%rax`	`%eax`	`%ax`	`%al`	valor de retorno
`%rbx`	`%ebx`	`%bx`	`%bl`	preservado por el llamado
`%rcx`	`%ecx`	`%cx`	`%cl`	cuarto argumento
`%rdx`	`%edx`	`%dx`	`%dl`	tercer argumento
`%rsi`	`%esi`	`%si`	`%sil`	segundo argumento
`%rdi`	`%edi`	`%di`	`%dil`	primer argumento
`%rsp`	—	—	—	puntero de pila
`%rbp`	—	—	—	puntero de marco
`%r8`–`%r15`	`%r8d...`	`%r8w...`	`%r8b...`	quinto y sexto argumento, y temporales

Una regla que sorprende y hay que retener: **escribir en un registro de 32 bits pone a cero los 32 bits altos del registro de 64**, mientras que escribir en uno de 16 u 8 deja intacto el resto. Es la razón por la que el compilador emite `mov %eax, %eax` de vez en cuando, y por la que `movl` y `movq` no son intercambiables.

2.2.3 Tipos de instrucción

Transferencia de datos. `mov` en sus variantes por tamaño (`movb`, `movw`, `movl`, `movq`), más las de extensión: `movzx` rellena con ceros y `movsx` extiende el signo. Elegir mal entre estas dos convierte un `-1` de 32 bits en un 4294967295 de 64.

`lea` merece un párrafo aparte. Su nombre dice que carga una dirección efectiva, pero **no accede a memoria**: calcula la dirección y la guarda. Por eso el compilador la usa como una instrucción aritmética barata: `lea (%rdi, %rdi, 2), %rax` multiplica por tres en una sola operación.

Aritmético-lógicas. `add`, `sub`, `imul`, `idiv`, `neg`, `and`, `or`, `xor`, `not`, y los desplazamientos `sal`, `sar` y `shr`. Dos detalles con consecuencias:

- `sar` es desplazamiento aritmético y conserva el signo; `shr` es lógico e introduce ceros. Desplazar un número negativo con `shr` produce un positivo enorme.
- `idiv` es la instrucción más cara del repertorio, decenas de ciclos, y exige extender el dividendo a 128 bits con `cqto` antes de llamarla. El compilador la evita siempre que puede: una división por una constante se convierte en una multiplicación por su inverso y un desplazamiento.

Comparación y banderas. `cmp` resta sin guardar el resultado y `test` hace un `and` sin guardarlo. Las dos existen solo por sus banderas. `test %rax, %rax` es el modismo para comprobar si un registro es cero, y es más corto que compararlo con la constante.

Control. `jmp` incondicional, y los condicionales que leen las banderas:

Instrucción	Salta si	Para operandos
`je`, `jne`	igual, distinto	cualquiera
`js`, `jns`	negativo, no negativo	con signo
`jg`, `jge`, `jl`, `jle`	mayor, mayor o igual, menor, menor o igual	**con signo**
`ja`, `jae`, `jb`, `jbe`	por encima, por debajo	**sin signo**

Confundir `jg` con `ja` es el error silencioso de este tema: el programa funciona con datos pequeños y falla cuando un valor tiene el bit alto puesto.

2.3 Instrucciones de control de flujo

Toda estructura de un lenguaje de alto nivel se reduce a comparaciones y saltos.

2.3.1 Condicional

```
if (a > b) { c = a; } else { c = b; }
```

```
        cmpq    %rsi, %rdi        # compara a con b
        jle     .Lelse
        movq    %rdi, %rdx        # c = a
        jmp     .Lfin
.Lelse: movq    %rsi, %rdx        # c = b
.Lfin:
```

La condición se invierte al traducir: el código salta cuando la condición del `if` es **falsa**, porque el camino verdadero es el que sigue en línea. Es la observación que hace legible cualquier desensamblado.

Para condicionales cortos el compilador prefiere `cmov`, que copia solo si se cumple la condición y no rompe el flujo. Es más rápido porque evita un salto —y con él el riesgo de control del tema 4—, pero **evalúa las dos ramas**, así que no sirve si una de ellas puede fallar, como una lectura por un puntero que podría ser nulo.

2.3.2 Bucles

Un bucle `while` se traduce con la comprobación al final y un salto de entrada, o directamente con la comprobación al final si el compilador puede demostrar que se ejecuta al menos una vez:

```
int suma = 0;
for (int i = 0; i < n; i++) { suma += v[i]; }
```

```
        xorl    %eax, %eax        # suma = 0
        xorl    %ecx, %ecx        # i = 0
        testl   %esi, %esi        # si n <= 0, no se entra
        jle     .Lfin
.Lbucle:
        addl    (%rdi,%rcx,4), %eax
        incl    %ecx
        cmpl    %esi, %ecx
        jl      .Lbucle
.Lfin:
```

El acceso `v[i]` es una sola instrucción gracias al direccionamiento indexado con escala 4, que es el

tamaño de un `int`. Ahí se ve por qué ese modo existe.

2.3.3 Selección múltiple

Un `switch` con casos densos no se traduce como una cadena de comparaciones, sino con una **tabla de saltos**: un vector de direcciones indexado por el valor, y un salto indirecto. El coste pasa a ser constante en vez de lineal en el número de casos. Con casos dispersos el compilador vuelve a las comparaciones, o construye un árbol de decisión.

2.4 Procedimientos y subrutinas

Llamar a una función exige acordar tres cosas: dónde van los argumentos, dónde vuelve el resultado y quién conserva qué. Ese acuerdo es el **convenio de llamada**, y no es parte de la ISA sino del sistema operativo.

2.4.1 El convenio System V de x86-64

- Los seis primeros argumentos enteros van en `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`. Los siguientes, a la pila.
- El valor de retorno, en `%rax`.
- **Preservados por el llamado**: `%rbx`, `%rbp`, `%r12`–`%r15`. Quien los use los guarda y los restaura.
- **Preservados por el llamante**: el resto. Quien los tenga vivos los salva antes de llamar.
- La pila queda alineada a 16 bytes en el punto de llamada.

El reparto entre las dos clases de registros es un compromiso: si todos fueran preservados por el llamado, una función hoja pagaría por registros que nadie tenía vivos; si todos fueran del llamante, se salvaría todo en cada llamada.

2.4.2 La pila

`call` apila la dirección de retorno y salta; `ret` desapila y salta ahí. La pila crece hacia direcciones **bajas**, así que apilar resta de `%rsp`.

El marco de una función:

```
direcciones altas
+-------------------------+
| argumentos del septimo  |
| en adelante             |
+-------------------------+
| direccion de retorno    |  <- apilada por call
+-------------------------+
| %rbp del llamante       |  <- si se usa puntero de marco
+-------------------------+  <- %rbp
| variables locales       |
+-------------------------+
| area para argumentos    |
+-------------------------+  <- %rsp
direcciones bajas
```

El prólogo y el epílogo típicos:

```
funcion:
```

```
pushq   %rbp
movq    %rsp, %rbp
subq    $32, %rsp          # espacio para locales
...
leave                      # equivale a movq %rbp,%rsp ; popq%rbp
ret
```

El puntero de marco `%rbp` no es obligatorio. Compilando con `-fomit-frame-pointer` el compilador lo libera como registro general y accede a las locales por desplazamiento sobre `%rsp`. El precio es que reconstruir la pila de llamadas deja de ser trivial, y por eso los perfiladores piden que se conserve.

La **zona roja** son los 128 bytes por debajo de `%rsp` que una función hoja puede usar sin restar del puntero de pila, porque el convenio garantiza que nada los pisa. Es una optimización específica de este convenio.

2.4.3 Recursión y variables locales

Cada llamada tiene su propio marco, y por eso las variables locales de cada activación son independientes. Es la razón por la que la recursión funciona sin que el lenguaje haga nada especial.

Y también la razón por la que devolver la dirección de una variable local es un error: el marco desaparece al volver, y el puntero queda apuntando a memoria que la siguiente llamada reutilizará. El programa suele funcionar al probarlo, que es lo que hace grave al fallo.

2.5 Vectores y estructuras de datos heterogéneas

2.5.1 Vectores

Un vector de n elementos de tamaño t ocupa $n \cdot t$ bytes contiguos, y la dirección del elemento i es $base + i \cdot t$. El compilador emite esa expresión con el direccionamiento indexado, sin multiplicaciones, siempre que t sea 1, 2, 4 u 8.

Los vectores multidimensionales se linealizan. En C el orden es **por filas**: para `int m[F][C]`, la dirección de `m[i][j]` es

$$base + (i \cdot C + j) \cdot 4$$

De ahí un resultado con consecuencias medibles: recorrer una matriz por filas es mucho más rápido que por columnas, porque el recorrido por filas accede a posiciones consecutivas y aprovecha la localidad espacial que el tema 6 desarrolla. Es el mismo programa con los bucles intercambiados, y la diferencia puede ser de un orden de magnitud.

Un vector de punteros no es lo mismo que una matriz: `int *m[F]` son F punteros a bloques que pueden estar en cualquier sitio, así que el acceso cuesta dos lecturas y pierde la contigüidad.

2.5.2 Estructuras

Los campos se colocan en orden de declaración, con relleno para que cada uno quede alineado, y la estructura entera se alinea al mayor de sus campos. Consecuencia práctica:

```
struct A { char c; int i; char d; };   /* 12 bytes */
struct B { int i; char c; char d; };   /* 8 bytes  */
```

Los mismos campos, cuatro bytes de diferencia. Declarar de mayor a menor tamaño minimiza el

relleno, y en un vector de un millón de estructuras eso son cuatro megabytes y una tasa de aciertos de caché distinta.

Una **unión** hace que todos los campos empiecen en el mismo desplazamiento, así que ocupa lo que el mayor. No hay ningún mecanismo que registre qué campo está vivo: eso corre por cuenta del programador, casi siempre con una etiqueta aparte.

2.6 Combinar ensamblador y alto nivel

Dos formas de mezclar, y una de leer lo que el compilador hizo:

Ensamblador en línea. Con `asm` y sus restricciones de entrada, salida y registros destruidos. Es la vía habitual para instrucciones que el lenguaje no expone: leer un contador de ciclos, emitir una barrera de memoria, usar una instrucción vectorial concreta.

Módulos separados. Un `.s` ensamblado por separado y enlazado con el resto. Basta con respetar el convenio de llamada y declarar el símbolo global.

Leer la salida del compilador. `gcc -S -O2` produce el ensamblador, y `objdump -d` desensambla el binario. Comparar `-O0` con `-O2` sobre el mismo programa es el ejercicio más instructivo del tema: se ve desaparecer las variables en registros, el bucle desenrollado y la división convertida en multiplicación.

Escribir ensamblador a mano para ganar velocidad casi nunca compensa hoy: el compilador conoce la microarquitectura mejor que el programador y programa las instrucciones para el cauce del tema 4. Se escribe ensamblador cuando no hay alternativa, no cuando se cree que se puede mejorar la salida del compilador. El tratamiento completo de este tema, con x86-64 como máquina de referencia, está en Bryant y O'Hallaron, 2016.

Capítulo 3

Unidad de control

Tema 3 del programa. El camino de datos sobre el que se ejecutan las instrucciones, y las dos formas de construir la unidad que lo gobierna.

3.1 El camino de datos

El camino de datos es el conjunto de elementos que transforman y transportan información: banco de registros, ALU, memorias, sumadores y los multiplexores que eligen entre alternativas. No decide nada; ejecuta lo que las señales de control le indican.

3.1.1 Elementos

Elemento	Función	Señales que recibe
Banco de registros	dos lecturas y una escritura simultáneas	número de registro, permiso de escritura
ALU	opera sobre dos entradas	código de operación
Memoria de instrucciones	entrega la instrucción de una dirección	—
Memoria de datos	lee y escribe	permiso de lectura, permiso de escritura
Extensor de signo	amplía un inmediato a la anchura completa	—
Multiplexores	eligen entre entradas	señal de selección

El banco de registros con dos puertos de lectura y uno de escritura no es arbitrario: es exactamente lo que una instrucción de tres direcciones necesita. La estructura del camino de datos se deduce del repertorio, no al revés.

3.1.2 Recorrido de una instrucción

Para una instrucción aritmética entre registros:

1. El contador de programa direcciona la memoria de instrucciones.
2. Los campos de la instrucción seleccionan dos registros fuente.
3. La ALU opera con lo leído.
4. El resultado se escribe en el registro destino.
5. El contador de programa se incrementa.

Para una carga desde memoria, el paso 3 calcula la dirección sumando el registro base y el desplazamiento extendido, el resultado direcciona la memoria de datos y lo leído va al registro

destino. Para un salto condicional, la ALU compara y un sumador aparte calcula la dirección destino; el multiplexor del contador de programa elige entre esa dirección y la siguiente instrucción.

El mismo hardware sirve a las tres, y los multiplexores son los que deciden qué camino se usa. **La unidad de control es el circuito que gobierna esos multiplexores y los permisos de escritura**, a partir del código de operación.

3.1.3 Ejecución monociclo y multiciclo

	Monociclo	Multiciclo
Duración del ciclo	la de la instrucción más lenta	la de la etapa más lenta
CPI	1	variable, según la instrucción
Recursos	duplicados; hace falta una memoria de instrucciones y otra de datos	compartidos entre etapas
Registros intermedios	no	sí, entre etapas

El diseño monociclo es sencillo y desperdicia: una instrucción aritmética, que no toca la memoria de datos, dura lo mismo que una carga. El multiciclo divide la ejecución en pasos y solo paga los que cada instrucción necesita, a costa de una unidad de control con estados. Ninguno de los dos se construye hoy: el punto de partida real es el cauce segmentado del tema 4, que es un multiciclo con instrucciones solapadas.

3.2 Señales de control

La unidad de control genera un vector de señales por instrucción. Para un repertorio reducido de tipo RISC-V:

Señal	Efecto
`RegWrite`	permite escribir en el banco de registros
`ALUSrc`	el segundo operando de la ALU es un registro o el inmediato
`ALUOp`	qué operación hace la ALU
`MemRead`, `MemWrite`	permisos de la memoria de datos
`MemToReg`	lo que se escribe en el registro viene de la ALU o de la memoria
`Branch`	la instrucción es un salto condicional

Y su valor por tipo de instrucción:

Instrucción	`RegWrite`	`ALUSrc`	`MemRead`	`MemWrite`	`MemToReg`	`Branch`
Aritmética registro-registro	1	0	0	0	0	0
Carga	1	1	1	0	1	0
Almacenamiento	0	1	0	1	X	0

Instrucción	RegWrite	ALUSrc	MemRead	MemWrite	MemToReg	Branch
Salto condicional	0	0	0	0	X	1

Las X son indiferentes: la señal no afecta al resultado porque no se escribe en ningún registro. Aprovecharlas es lo que permite simplificar el circuito, y es la razón por la que una tabla de verdad con indiferencias produce menos puertas.

3.3 Unidad de control cableada

La unidad se construye como un circuito combinacional —o secuencial, si hay estados— cuya entrada es el código de operación y cuya salida son las señales. Se diseña como cualquier circuito: tabla de verdad, simplificación, síntesis.

- **A favor:** es lo más rápido posible. La señal se propaga por unas pocas puertas.
- **En contra:** cambiar el repertorio obliga a rediseñar el circuito. Con cientos de instrucciones, el diseño se vuelve inabordable y la depuración, peor.

Es la solución de las arquitecturas RISC, donde el repertorio es pequeño y regular precisamente para que quepa en un circuito manejable.

3.4 Unidad de control microprogramada

La idea de Wilkes, de 1951: en vez de un circuito, **un programa**. Cada instrucción máquina se implementa como una secuencia de microinstrucciones guardadas en una memoria de control, y cada microinstrucción contiene directamente las señales de control de un ciclo.

3.4.1 Estructura

Elemento	Función
Memoria de control	guarda las microinstrucciones
Registro de dirección de microinstrucción	apunta a la actual
Registro de microinstrucción	la que se está ejecutando
Lógica de secuenciamiento	decide cuál es la siguiente

El ciclo de la unidad de control es una copia en miniatura del ciclo de instrucción: buscar la microinstrucción, ejecutarla —es decir, emitir sus señales— y calcular la siguiente dirección. La secuencia de cada instrucción empieza en una dirección obtenida a partir del código de operación.

3.4.2 Formatos de microinstrucción

	Horizontal	Vertical
Codificación	un bit por señal	campos codificados
Anchura	grande	pequeña
Decodificación	ninguna	hace falta decodificar
Paralelismo	máximo, varias señales a la vez	limitado

	Horizontal	Vertical
Microprogramas	cortos	largos

Es el mismo compromiso entre espacio y tiempo que aparece en toda la asignatura. Los diseños reales usan formatos mixtos: horizontal en las señales que necesitan concurrencia, codificado en las que son mutuamente excluyentes.

3.4.3 Comparación

	Cableada	Microprogramada
Velocidad	mayor	menor
Coste de diseño	alto si el repertorio es grande	manejable
Modificar el repertorio	rediseñar el circuito	reescribir el microprograma
Corregir un error tras fabricar	imposible	posible, si la memoria es escribible
Encaja con	RISC	CISC

La última fila de la tabla es la que sigue teniendo efecto hoy. Los procesadores x86 actuales tienen unidad de control híbrida: las instrucciones frecuentes y simples se decodifican en hardware cableado, y las complejas y raras se traducen mediante microcódigo. Y ese microcódigo es **actualizable**: los parches contra Spectre y contra errores de fabricación se distribuyen como actualizaciones de microcódigo que el firmware carga al arrancar. Un procesador ya vendido cambia de comportamiento sin tocar el silicio, y eso es exactamente lo que la unidad microprogramada permite. Los dos enfoques se desarrollan con detalle en Stallings, 2022 y en Hamacher et al., 2012.

3.5 Excepciones e interrupciones

La unidad de control tiene que atender sucesos que rompen la secuencia. El tratamiento:

1. Terminar o abortar la instrucción en curso, según el suceso.
2. Guardar el contador de programa y el registro de estado.
3. Determinar la causa, por vector o por consulta.
4. Saltar a la rutina de tratamiento.
5. Al terminar, restaurar el estado y volver.

El paso 1 es el difícil. Una excepción **precisa** deja la máquina en un estado en el que todas las instrucciones anteriores han terminado y ninguna posterior ha empezado, así que el tratamiento puede reanudar la ejecución. Conseguirlo es trivial en una máquina monociclo y deja de serlo en cuanto hay solapamiento, que es el tema siguiente: con varias instrucciones en vuelo, la que falla no es la última que se empezó.

Sin excepciones precisas no hay memoria virtual: un fallo de página tiene que poder reanudarse en el punto exacto. Es la razón por la que las arquitecturas que las implementaron mal quedaron limitadas.

Capítulo 4

Segmentación de cauce

Tema 4 del programa. Cómo se solapa la ejecución de varias instrucciones, qué situaciones impiden ese solapamiento y con qué se resuelven.

4.1 Conceptos básicos

La segmentación divide la ejecución en etapas y permite que en cada instante haya una instrucción distinta en cada etapa. No acelera una instrucción: aumenta el número de instrucciones terminadas por unidad de tiempo.

El cauce clásico de cinco etapas:

Etapa	Nombre	Qué hace
IF	búsqueda	lee la instrucción de memoria
ID	decodificación	interpreta y lee los registros fuente
EX	ejecución	opera, o calcula una dirección
MEM	memoria	accede a la memoria de datos
WB	escritura	escribe el resultado en el registro

Entre cada par de etapas hay un registro que retiene el resultado parcial y el estado de control. Sin esos registros no habría cauce: cada etapa pisaría a la siguiente.

```
ciclo:    1    2    3    4    5    6    7    8    9
inst 1:   IF   ID   EX   MEM  WB
inst 2:        IF   ID   EX   MEM  WB
inst 3:             IF   ID   EX   MEM  WB
inst 4:                  IF   ID   EX   MEM  WB
inst 5:                       IF   ID   EX   MEM  WB
```

4.1.1 Ganancia

Con n instrucciones y k etapas, un cauce ideal tarda $k + (n - 1)$ ciclos frente a los $n \cdot k$ de la ejecución secuencial. La ganancia es

$$S = \frac{n \cdot k}{k + (n - 1)}$$

que tiende a k cuando n crece. La cota es el número de etapas, y solo se alcanza si el cauce nunca se detiene.

Tres cosas impiden llegar:

- **El llenado y el vaciado.** Los primeros $k-1$ ciclos el cauce no está lleno. Con muchas instrucciones el efecto se diluye.
- **El desequilibrio entre etapas.** El ciclo lo fija la etapa más lenta, así que las demás desperdician tiempo. Dividir en más etapas reduce el ciclo pero añade el retardo de los registros intermedios, que no baja.
- **Los riesgos**, que son el resto del tema.

Más etapas no es siempre mejor. El Pentium 4 llegó a 31 y el diseño se abandonó: el coste de vaciar un cauce tan profundo en cada salto mal predicho superaba la ganancia de frecuencia. Los diseños actuales se mueven entre 14 y 20 etapas.

4.2 Riesgos

Situaciones en las que la instrucción siguiente no puede ejecutarse en el ciclo que le tocaría. Tres clases.

4.3 Riesgos estructurales

Dos instrucciones necesitan el mismo recurso a la vez. El caso típico: la etapa IF de una instrucción y la etapa MEM de otra acceden a memoria en el mismo ciclo.

La solución es duplicar el recurso, y es la razón de que el primer nivel de caché esté dividido en instrucciones y datos. La alternativa —detener el cauce un ciclo— convierte un problema de diseño en una pérdida de rendimiento permanente.

Otro caso frecuente es la unidad de división, que no se segmenta porque duplicarla no compensa: dos divisiones seguidas se serializan.

4.4 Riesgos de datos

Una instrucción necesita un resultado que otra anterior todavía no ha escrito.

```
addq    %rbx, %rax        # escribe %rax en WB, ciclo 5
subq    %rax, %rcx        # lee %rax en ID, ciclo 3
```

La segunda lee en el ciclo 3 lo que la primera escribe en el 5. Sin nada que lo evite, lee el valor antiguo.

4.4.1 Los tres tipos

Tipo	Nombre	Descripción	En un cauce en orden
RAW	lectura tras escritura	se lee lo que aún no se ha escrito	**el único real**
WAR	escritura tras lectura	se escribe antes de que otro haya leído	imposible: se lee en ID y se escribe en WB
WAW	escritura tras escritura	dos escrituras en orden incorrecto	imposible si todas escriben en WB

WAR y WAW son **falsas dependencias**: no hay flujo de información, solo reutilización de un nombre de registro. Aparecen en cuanto las instrucciones pueden completarse fuera de orden, y se eliminan con renombrado de registros, que es lo que hacen los procesadores superescalares.

4.4.2 Anticipación

La solución principal. El resultado existe al final de EX, aunque no se escriba hasta WB: basta con encaminarlo directamente a la entrada de la ALU de la instrucción siguiente, sin pasar por el banco de registros.

```
addq    %rbx, %rax
subq    %rax, %rcx
         ^
         |  el resultado de EX de la primera entra
         |  como operando de EX de la segunda
```

Con anticipación desde EX y desde MEM, la mayoría de los riesgos RAW desaparecen sin perder un solo ciclo. El precio es hardware: comparadores que detectan la coincidencia de registros y multiplexores en las entradas de la ALU.

4.4.3 El caso que la anticipación no resuelve

Una carga seguida de un uso inmediato:

```
movq    (%rdi), %rax      # el dato sale de MEM, ciclo 4
addq    %rax, %rbx        # lo necesita en EX, ciclo 4
```

El dato no existe hasta el final de MEM, y la instrucción siguiente lo necesita al principio de su EX, que ocurre en ese mismo ciclo. **No hay forma de adelantarlo**: haría falta enviarlo hacia atrás en el tiempo.

La única salida es detener el cauce un ciclo, insertando una **burbuja**. Es el riesgo *load-use*, y es el motivo por el que el compilador reordena las instrucciones para colocar algo útil entre la carga y su uso. Cuando no encuentra nada, emite un `nop` o deja que el hardware detenga.

4.4.4 Detección y detención

El hardware compara los registros fuente de la instrucción en ID con los registros destino de las que van por delante. Si coinciden y no se puede anticipar, congela IF e ID e inyecta una burbuja en EX. Las etapas posteriores siguen avanzando, así que la burbuja se propaga y sale por el final.

4.5 Riesgos de control

Un salto no se resuelve hasta una etapa avanzada, y mientras tanto el cauce ya ha leído instrucciones que quizá no haya que ejecutar.

Si el salto se resuelve en MEM, hay tres instrucciones dentro del cauce que pueden ser incorrectas. Si el salto se toma, hay que descartarlas, y esos tres ciclos se pierden. Con saltos en el 20 % de las instrucciones y una penalización de tres ciclos, el CPI sube de 1 a 1,6.

4.5.1 Soluciones

Resolver antes. Adelantar la comparación y el cálculo de la dirección a la etapa ID reduce la penalización a un solo ciclo. Es la primera medida, y la más barata.

Salto retardado. La arquitectura declara que la instrucción siguiente al salto se ejecuta siempre, y el compilador coloca ahí algo útil. Funcionó en MIPS con un cauce de cinco etapas, y no escala: con cauces profundos harían falta muchas ranuras de retardo, y llenarlas es imposible. Es una decisión de ISA que resultó ser un error a largo plazo, porque el contrato queda grabado en el repertorio.

Predicción. Suponer el resultado y seguir. Si se acierta no se pierde nada; si se falla, se descarta lo especulado y se paga la penalización completa.

4.5.2 Predicción de saltos

Esquema	Cómo decide	Acierto típico
Estático, no tomado	siempre sigue en línea	~50 %
Estático por dirección	los saltos hacia atrás se toman, los de adelante no	~65 %
Dinámico de 1 bit	repite lo que ocurrió la última vez	~80 %
Dinámico de 2 bits	necesita dos fallos seguidos para cambiar	~90 %
Correlado y de torneo	combina historia local y global	>95 %

La heurística de la segunda fila funciona porque **los saltos hacia atrás son bucles**, y un bucle se toma casi siempre.

El predictor de 2 bits merece detalle porque su ventaja es concreta. Con un solo bit, un bucle de n iteraciones falla **dos veces** por ejecución: al salir, y la primera vez de la siguiente entrada. Con dos bits y un contador de saturación, el estado no cambia hasta el segundo fallo consecutivo, así que la salida del bucle no destruye la predicción y solo se falla una vez.

Los predictores actuales son mucho más elaborados y superan el 99 % en código regular. Ese acierto es la base de la ejecución especulativa, y también de Meltdown y Spectre: la especulación deja huella en la caché aunque el resultado se descarte.

4.6 Más allá del cauce simple

Tres extensiones que las máquinas reales combinan:

- **Superescalar.** Varias instrucciones por ciclo, con unidades funcionales duplicadas. El CPI baja de 1, y por eso se mide su inverso, el IPC.
- **Ejecución fuera de orden.** Las instrucciones se ejecutan cuando sus operandos están listos, no en orden de programa, y se retiran en orden para mantener las excepciones precisas. Necesita renombrado de registros, que es lo que elimina las dependencias WAR y WAW.
- **Multihilo simultáneo.** Varios hilos comparten las unidades funcionales, así que cuando uno se detiene el otro las aprovecha. Duplica los registros de estado, no las unidades: dos hilos hardware no son dos procesadores.

Ninguna cambia lo que el programa ve. La ISA sigue siendo la misma y el programador razona en orden secuencial; el hardware se encarga de que el resultado sea indistinguible del de una ejecución en orden. El tratamiento cuantitativo de todo esto está en Patterson y Hennessy, 2021 y en Harris y Harris, 2022.

Capítulo 5

Entrada/Salida

Tema 5 del programa. Cómo se conecta el procesador con los dispositivos, y las tres técnicas con las que se transfiere información entre ellos.

5.1 Funciones del sistema de entrada/salida

Los dispositivos no se pueden conectar al bus del procesador tal cual. Entre medias hace falta un **módulo de entrada/salida** que resuelva cuatro desacoplos:

Desacoplo	Ejemplo
Velocidad	un teclado entrega unos bytes por segundo; la memoria, gigabytes
Formato de datos	serie frente a paralelo, anchuras distintas
Códigos y unidades	el dispositivo trabaja con sectores; el programa, con bytes
Sincronización	el dispositivo va a su ritmo, no al del reloj del procesador

Las funciones del módulo: control y temporización, comunicación con el procesador, comunicación con el dispositivo, almacenamiento temporal y detección de errores.

El **almacenamiento temporal** es el que más consecuencias tiene. Sin un búfer en el módulo, la transferencia iría a la velocidad del dispositivo y ocuparía el bus todo ese tiempo. Con búfer, el bus se ocupa solo mientras dura la transferencia entre módulo y memoria, que va a velocidad de memoria.

5.2 Interfaces de entrada/salida

El módulo expone al procesador un conjunto de registros:

Registro	Sentido	Contenido
Datos	ambos	lo que se transfiere
Estado	lectura	listo, ocupado, error
Control	escritura	qué operación iniciar

Y hay dos formas de direccionar esos registros:

	Entrada/salida aislada	Proyectada en memoria
Espacio de direcciones	uno propio	el mismo de la memoria
Instrucciones	específicas, `in` y `out`	las de acceso a memoria
Líneas de control	una señal distingue memoria de E/S	ninguna adicional
Protección	inmediata, son instrucciones privilegiadas	por la unidad de gestión de memoria
Direcciones disponibles	todas, no restan a la memoria	restan al espacio direccionable

x86 tiene las dos y la segunda domina. La razón es práctica: con proyección en memoria, un manejador se escribe en C sin recurrir a ensamblador, y el repertorio completo de instrucciones de acceso a memoria queda disponible para operar sobre los registros del dispositivo.

Su precaución obligada: esas direcciones **no se pueden cachear ni reordenar**. El valor cambia sin que el procesador escriba, y un compilador que elimine una lectura «redundante» rompe el manejador. De ahí `volatile` y las barreras de memoria.

5.3 Entrada/salida programada

El procesador ejecuta la transferencia entera. La secuencia:

1. Escribe la orden en el registro de control.
2. Lee el registro de estado en un bucle hasta que el dispositivo está listo.
3. Transfiere una palabra entre el registro de datos y un registro del procesador.
4. Repite desde 2 hasta terminar.

```
espera: inb     $ESTADO, %al
        testb   $LISTO, %al
        jz      espera
        inb     $DATOS, %al
```

- **A favor:** simplicidad total y la latencia mínima posible.
- **En contra:** el procesador consume el 100 % de su tiempo esperando. Con un dispositivo lento, se desperdician millones de ciclos por byte.

El bucle de la línea 2 es la **espera activa**. Solo se justifica cuando la espera es brevísima —del orden de decenas de ciclos, donde el coste de una interrupción sería mayor— o cuando el sistema no tiene nada más que hacer, como en el arranque o en un microcontrolador de función única.

5.4 Interrupciones

El dispositivo avisa cuando está listo, y mientras tanto el procesador trabaja en otra cosa.

5.4.1 Secuencia

1. El módulo activa la línea de petición de interrupción.
2. El procesador termina la instrucción en curso.
3. Reconoce la petición.
4. Guarda el contador de programa y el registro de estado.
5. Identifica la causa y salta a la rutina de tratamiento.
6. La rutina salva los registros que vaya a usar, transfiere el dato y los restaura.

7. Una instrucción de retorno de interrupción restaura el estado y reanuda.

El paso 2 explica la **latencia de interrupción**: el procesador no interrumpe a mitad de instrucción, así que la instrucción más larga del repertorio fija el peor caso. Las instrucciones de cadena de x86, que pueden copiar miles de bytes, son interrumpibles precisamente por esto: guardan su progreso en registros y se reanudan.

5.4.2 Identificación de la fuente

Método	Cómo	Coste
Consulta por *software*	la rutina lee el estado de cada módulo hasta encontrar el que pidió	lineal en el número de dispositivos
Consulta encadenada	los módulos se conectan en cadena y el más cercano responde primero	la prioridad la fija la posición física
Vectorizada	el módulo envía un identificador y el procesador indexa una tabla	constante, y es lo que se usa
Arbitraje por bus	el módulo compite por el bus antes de responder	permite prioridades dinámicas

Con interrupciones vectorizadas, la tabla de vectores traduce el identificador en la dirección de la rutina. Es la misma tabla que el sistema operativo programa al arrancar.

5.4.3 Prioridades y anidamiento

Varias peticiones simultáneas se resuelven por prioridad, y una interrupción de prioridad alta puede interrumpir a la rutina de una más baja. El anidamiento exige que el estado se guarde en pila, no en un registro fijo: con un solo registro, la segunda interrupción destruiría el contexto de la primera.

Un procesador puede además **enmascarar** interrupciones, inhibiéndolas durante las secciones en las que su tratamiento causaría problemas. La regla es que esas secciones sean lo más cortas posible: con las interrupciones inhibidas, la latencia de todo el sistema crece.

La **interrupción no enmascarable** queda fuera de ese mecanismo y se reserva para sucesos que no admiten espera, como un fallo de alimentación inminente.

5.4.4 El límite

Una interrupción por dato transferido es aceptable con un teclado y ruinoso con un disco: transferir un sector de 4 KiB costaría cuatro mil interrupciones. La sobrecarga de guardar y restaurar el contexto acabaría dominando, y de ahí la tercera técnica.

5.5 DMA

Un controlador de acceso directo a memoria transfiere bloques entre el dispositivo y la memoria sin intervención del procesador.

5.5.1 Funcionamiento

1. El procesador programa el controlador: dirección de memoria, número de palabras, sentido y dispositivo.
2. El procesador sigue con otra cosa.
3. El controlador toma el control del bus y transfiere palabra a palabra.
4. Al terminar el bloque entero, genera **una sola** interrupción.

El procesador interviene dos veces —al programar y al recibir el aviso— en vez de una vez por palabra. La ganancia crece con el tamaño del bloque.

5.5.2 Cómo se comparte el bus

Modo	Qué hace	Efecto sobre el procesador
Robo de ciclo	toma el bus un ciclo por palabra	lo frena, no lo detiene
Ráfaga	toma el bus hasta terminar el bloque	lo detiene mientras dura
Transparente	usa solo los ciclos en que el procesador no accede al bus	ninguno, pero es más lento

El robo de ciclo es el compromiso habitual. La ráfaga se usa cuando la transferencia tiene que completarse en un plazo, como en la captura de vídeo.

5.5.3 Coherencia de caché

El controlador escribe en memoria sin que las cachés del procesador se enteren. Si una línea de caché contiene una copia de esa zona, queda obsoleta; y si está sucia, la escritura posterior de la caché pisará lo que el dispositivo escribió.

Tres soluciones, de más a menos automática:

1. **Coherencia por hardware.** El controlador de caché espía el bus e invalida las líneas afectadas. Es lo normal en máquinas de propósito general.
2. **Marcar la región como no cacheable.** Sencillo y lento.
3. **Invalidar y vaciar a mano** antes y después de la transferencia. Es lo que hacen los manejadores en arquitecturas empotradas sin coherencia, y una fuente clásica de errores intermitentes.

5.5.4 Seguridad

Un dispositivo con DMA puede leer y escribir **toda** la memoria física, incluida la del sistema operativo. Un puerto externo con acceso directo al bus —Thunderbolt, PCIe expuesto— convierte eso en una vía de ataque.

La respuesta es la **IOMMU**, que traduce y comprueba las direcciones que emiten los dispositivos igual que la MMU hace con las de los procesos. Cada dispositivo recibe su propio espacio de direcciones y solo alcanza lo que se le ha asignado.

5.6 Comparación de las tres técnicas

	Programada	Interrupciones	DMA
Quién transfiere	el procesador	el procesador	el controlador
Ocupación del procesador	total	por dato	por bloque
Latencia	mínima	media	mayor, por la programación inicial
Adecuada para	esperas brevísimas	dispositivos lentos y esporádicos	bloques grandes

Las tres conviven en una máquina real: el teclado por interrupciones, el disco por DMA, y el arranque por entrada/salida programada, porque todavía no hay sistema que atienda una interrupción. El desarrollo completo, con los diagramas de tiempos de cada técnica, está en Stallings, 2022 y en Hamacher et al., 2012.

Capítulo 6

Memoria

Tema 6 del programa. Por qué la memoria se organiza en niveles, qué propiedad de los programas hace que esa organización funcione, y cómo se diseña el nivel que más rendimiento decide: la caché.

6.1 Jerarquía de memoria

No existe una tecnología que sea a la vez rápida, grande y barata. Las tres propiedades se contraponen:

Tecnología	Tiempo de acceso	Coste por bit	Capacidad típica
Registros	menos de 1 ciclo	altísimo	cientos de bytes
Caché SRAM	1 a 40 ciclos	alto	de KiB a decenas de MiB
Memoria principal DRAM	200 a 300 ciclos	medio	GiB
Disco de estado sólido	decenas de miles de ciclos	bajo	TiB
Disco magnético	millones de ciclos	muy bajo	TiB

La jerarquía apila los niveles de forma que el procesador vea, aproximadamente, la velocidad del más rápido y la capacidad del más lento. Cada nivel guarda un subconjunto del siguiente.

Los números de la tabla son la clave del tema: entre un registro y la DRAM hay dos órdenes de magnitud, y ese hueco —la **brecha de memoria**— ha crecido durante décadas, porque la velocidad del procesador mejoró más rápido que la latencia de la memoria. La caché no es una optimización opcional; es lo que hace utilizable a un procesador moderno.

6.1.1 Por qué funciona

Solo funciona porque los programas no acceden a la memoria al azar. Cumplen el **principio de localidad**:

- **Localidad temporal.** Lo accedido hace poco se vuelve a acceder pronto. La causan los bucles, las variables de trabajo y las llamadas repetidas.
- **Localidad espacial.** Lo próximo a lo accedido se accede pronto. La causan el recorrido secuencial de vectores, la ejecución en línea del código y los campos contiguos de una estructura.

Sobre localidad temporal se apoya la decisión de **qué guardar**; sobre localidad espacial, la de **traer bloques y no palabras sueltas**.

El caso que lo demuestra por contraste es el recorrido de una matriz por columnas. El programa es el mismo con dos bucles intercambiados, y la diferencia de tiempo puede ser de un orden de magnitud, porque el recorrido por columnas salta una fila entera entre accesos y no aprovecha ni un byte de los que la caché trajo.

6.2 Memoria caché

6.2.1 Funcionamiento

La memoria se divide en **bloques** del mismo tamaño, y la caché en **líneas** que alojan un bloque cada una. Ante un acceso:

- **Acierto.** El bloque está en la caché. Se sirve a velocidad de caché.
- **Fallo.** No está. Se trae el bloque entero del nivel siguiente, se coloca en una línea y se sirve.

El tiempo medio de acceso, con tasa de aciertos h, tiempo de acierto T_a y penalización de fallo T_f:

$$T_{medio} = T_a + (1 - h) \cdot T_f$$

Con $T_a = 1$ ciclo, $T_f = 200$ y $h = 0{,}95$, el acceso medio son 11 ciclos. Subir la tasa al 99 % lo baja a 3. **Un punto porcentual de aciertos vale más que duplicar la velocidad de la caché**, y por eso el diseño se orienta a fallar menos y no a acertar más rápido.

Con varios niveles la fórmula se anida, y conviene distinguir dos tasas: la **local**, referida a los accesos que llegan a ese nivel, y la **global**, referida a todos los del procesador. La local del segundo nivel parece mala —muchos aciertos ya los absorbió el primero— y no lo es.

6.2.2 Organización

Dónde puede colocarse un bloque. Tres esquemas, y los dos extremos son casos particulares del intermedio.

Directa. Cada bloque tiene una única línea posible, la de índice *bloque* mód L, con L líneas.

- Búsqueda inmediata: se calcula el índice y se compara una etiqueta.
- Y el problema: dos bloques que compitan por la misma línea se expulsan mutuamente aunque el resto de la caché esté vacía. Es el **fallo por conflicto**, y produce el patrón patológico de recorrer dos vectores separados por una potencia de dos.

Totalmente asociativa. Cualquier bloque en cualquier línea. Elimina los conflictos y obliga a comparar todas las etiquetas a la vez, lo que solo es viable con pocas líneas. Se usa en las TLB.

Asociativa por conjuntos de n vías. La caché se divide en conjuntos de n líneas; un bloque va a un conjunto fijo y a cualquier línea dentro de él. Se comparan n etiquetas.

	Directa	4 vías	Totalmente asociativa
Comparadores	1	4	uno por línea
Fallos por conflicto	muchos	pocos	ninguno
Tiempo de acierto	el menor	intermedio	el mayor
Uso real	cachés grandes de último nivel	primer y segundo nivel	TLB

Entre 4 y 8 vías se obtiene casi toda la ventaja; a partir de ahí el rendimiento apenas mejora y el tiempo de acierto empeora.

6.2.3 Formato de la dirección

Una dirección se descompone en tres campos:

```
|<---- etiqueta ---->|<--- indice --->|<-- desplazamiento -->|
```

- **Desplazamiento**: $\log_2$(tamaño de bloque) bits. Selecciona el byte dentro del bloque.
- **Índice**: $\log_2$(número de conjuntos) bits. Selecciona el conjunto.
- **Etiqueta**: el resto. Se compara para saber si el bloque es el buscado.

Ejemplo: caché de 32 KiB, asociativa de 4 vías, bloques de 64 bytes, direcciones de 32 bits.

Magnitud	Cálculo	Valor
Líneas	32768/64	512
Conjuntos	512/4	128
Desplazamiento	$\log_2 64$	6 bits
Índice	$\log_2 128$	7 bits
Etiqueta	$32 - 7 - 6$	19 bits

Cada línea guarda además un bit de validez y, si la escritura es diferida, un bit de modificado. El coste de esos metadatos no es despreciable: con etiquetas de 19 bits y bloques de 64 bytes, el 4 % de la caché no guarda datos.

6.2.4 Tamaño de bloque

Aumentarlo aprovecha mejor la localidad espacial y reduce el número de etiquetas. Pero pasado un punto lo empeora todo: hay menos bloques distintos en la caché, la penalización por fallo crece porque hay que traer más bytes, y se transfieren datos que no se van a usar. La curva de fallos frente a tamaño de bloque tiene un mínimo, típicamente entre 32 y 128 bytes.

6.2.5 Estrategias

Extracción. Cuándo se trae un bloque.

- *Por demanda*: cuando se falla. Es lo básico.
- *Anticipada* (prebúsqueda): se traen bloques que se predice que harán falta. Acierta con recorridos secuenciales y, si falla, contamina la caché expulsando algo útil.

Colocación. Dónde se coloca. La determina la organización.

Reemplazo. Qué línea se expulsa cuando el conjunto está lleno. En una caché directa no hay elección; en las demás:

Política	Criterio	Coste
LRU	la que lleva más tiempo sin usarse	exacto solo con pocas vías
Pseudo-LRU	aproximación con un árbol de bits	lo que se implementa
FIFO	la más antigua	barato y peor
Aleatoria	al azar	sorprendentemente competitiva con muchas vías

Actualización. Qué hacer al escribir.

	Escritura inmediata	Escritura diferida
Cuándo se propaga	en cada escritura	al expulsar la línea
Tráfico al nivel siguiente	alto	bajo
Coherencia entre niveles	siempre	solo al expulsar
Metadatos	ninguno	bit de modificado
Fallo de lectura	no requiere escribir nada	puede exigir volcar la línea sucia

La escritura inmediata se acompaña de un **búfer de escritura** que absorbe las escrituras y deja seguir al procesador; sin él, cada escritura costaría un acceso a memoria principal.

Y ante un fallo de escritura hay dos opciones: traer el bloque y escribir sobre él, o escribir directamente al nivel siguiente sin traerlo. La primera se combina con escritura diferida y la segunda con escritura inmediata, porque son las parejas coherentes.

6.2.6 Clasificación de los fallos

Las tres «C», que dicen qué hacer con cada tipo:

Tipo	Causa	Se reduce con
Forzosos	primer acceso al bloque	bloques mayores, prebúsqueda
De capacidad	la caché no cabe el conjunto de trabajo	más capacidad
De conflicto	varios bloques compiten por el mismo conjunto	más asociatividad

Los forzosos son inevitables por definición. Distinguir capacidad de conflicto es lo que dice si conviene una caché mayor o más asociativa, y se mide comparando con una caché totalmente asociativa del mismo tamaño: lo que esta evita son los de conflicto.

En multiprocesadores se añade una cuarta: los fallos de **coherencia**, causados por invalidaciones de otro procesador. De ahí el *falso compartimiento*, donde dos núcleos escriben variables distintas que caen en la misma línea y se invalidan mutuamente sin compartir nada en realidad.

6.3 Memorias con múltiples módulos

La memoria principal se construye con varios módulos que pueden trabajar en paralelo, lo que multiplica el ancho de banda sin bajar la latencia de un acceso suelto.

- **Entrelazado de orden bajo.** Los bits bajos de la dirección seleccionan el módulo, así que direcciones consecutivas caen en módulos distintos. Un acceso secuencial —traer una línea de caché— se reparte entre todos y va n veces más rápido. Es lo que se usa.
- **Entrelazado de orden alto.** Los bits altos seleccionan el módulo, así que cada uno cubre un rango contiguo. No acelera el acceso secuencial; sirve para aislar módulos o poder ampliar la memoria.

La DRAM añade su propia estructura: los accesos se hacen por fila, y leer varias palabras de la misma fila abierta es mucho más barato que cambiar de fila. Los canales, rangos y bancos de un módulo moderno son la misma idea de entrelazado llevada a varios niveles.

6.4 Influencia en las prestaciones

El efecto de la memoria sobre el CPI se suma al CPI de ejecución:

$$\text{CPI} = \text{CPI}_{ejec} + \text{accesos por instrucción} \times (1 - h) \times \text{penalización}$$

Con CPI de ejecución 1, 1,3 accesos por instrucción, 3 % de fallos y 200 ciclos de penalización, el CPI resultante es $1 + 1{,}3 \cdot 0{,}03 \cdot 200 = 8{,}8$. **El procesador pasa casi el 90 % del tiempo esperando a la memoria.** El número es el argumento de todo el tema.

Cómo lo reduce el programador, sin tocar el hardware:

- **Recorrer en el orden en que los datos están en memoria**, que en C es por filas.
- **Bloquear** los algoritmos: dividir una multiplicación de matrices en submatrices que quepan en la caché, para reutilizar cada bloque mientras está cargado.
- **Compactar las estructuras**, ordenando los campos de mayor a menor tamaño para minimizar el relleno, y separando los campos que se recorren de los que no.
- **Alinear** los datos al tamaño de línea cuando el acceso es concurrente, para evitar el falso compartimiento.

Ninguna de esas medidas cambia el algoritmo ni su complejidad asintótica, y todas pueden multiplicar la velocidad por varias veces. Es la razón por la que este tema cierra la asignatura: explica por qué dos programas con el mismo número de operaciones tardan tiempos muy distintos. El análisis cuantitativo completo está en Patterson y Hennessy, 2021, y el enfoque desde el punto de vista del programador, en Bryant y O'Hallaron, 2016.

Capítulo 7

Temario práctico

Las seis prácticas del programa, con su seminario correspondiente. Todas sobre x86-64 en Linux con las herramientas GNU.

7.1 Práctica 1. Entorno de desarrollo GNU

Las cuatro etapas que `gcc` encadena, y cómo detenerse en cada una:

Etapa	Herramienta	Entrada	Salida	Cómo pararse
Preprocesado	`cpp`	`.c`	`.i`	`gcc -E`
Compilación	`cc1`	`.i`	`.s`	`gcc -S`
Ensamblado	`as`	`.s`	`.o`	`gcc -c`
Enlazado	`ld`	`.o`	ejecutable	`gcc`

`gcc -S -O2 prog.c` es la orden que más se usa en toda la asignatura: produce el ensamblador que el compilador genera, y compararlo con `-O0` enseña más sobre el tema 2 que cualquier explicación.

Herramientas de inspección:

Orden	Para qué
`objdump -d`	desensambla las secciones de código
`objdump -h`	lista las secciones y sus tamaños
`readelf -a`	cabeceras, símbolos y reubicaciones del ELF
`nm`	tabla de símbolos
`size`	tamaño de texto, datos y BSS
`strings`	cadenas imprimibles del binario
`ldd`	bibliotecas dinámicas de las que depende

Las secciones de un ELF, que se corresponden con las regiones que el tema 2 describía:

Sección	Contenido	Permisos
`.text`	código	lectura y ejecución
`.rodata`	constantes y literales de cadena	solo lectura
`.data`	variables globales con valor inicial	lectura y escritura
`.bss`	variables globales a cero	lectura y escritura, no ocupa el fichero

Sección	Contenido	Permisos
`.symtab`	símbolos	—

Que `.bss` no ocupe espacio en el ejecutable es comprobable: declarar un vector global de un millón de enteros no cambia el tamaño del fichero, y declararlo inicializado sí.

`make`. Un objetivo, sus prerrequisitos y las órdenes que lo construyen. La regla que evita la mayor parte de los problemas es declarar bien las dependencias: un `.o` depende de su `.c` **y de las cabeceras que incluye**, o cambiar una cabecera no reconstruye nada.

7.2 Práctica 2. Programación en ensamblador: programas aritméticos

Un programa completo en ensamblador de GNU, sintaxis AT&T:

```
        .section .rodata
fmt:    .asciz "El resultado es %ld\n"

        .text
        .globl  main
main:
        pushq   %rbp
        movq    %rsp, %rbp

        movq    $10, %rdi
        call    factorial

        movq    %rax, %rsi
        leaq    fmt(%rip), %rdi
        xorl    %eax, %eax          # printf es variadica: 0 registros XMM
        call    printf

        xorl    %eax, %eax
        leave
        ret

# long factorial(long n)
factorial:
        movq    $1, %rax
        testq   %rdi, %rdi
        jle     .Lfin
.Lbucle:
        imulq   %rdi, %rax
        decq    %rdi
        jnz     .Lbucle
.Lfin:
        ret
```

Cinco cosas de este programa que se pasan por alto y rompen la ejecución:

1. **La sintaxis AT&T pone el destino a la derecha**, al revés que Intel. `movq $1, %rax` carga 1 en `%rax`. Leer una salida de `objdump` con la convención contraria produce razonamientos exactamente invertidos.
2. **`%rax` a cero antes de llamar a `printf`.** El convenio exige que indique cuántos registros vectoriales llevan argumentos. Sin eso, `printf` puede leer registros que no se han preparado.
3. **`leaq fmt(%rip), %rdi`**, no `movq $fmt, %rdi`. El código de posición independiente direcciona relativo al contador de programa, y es lo que el enlazador espera por omisión.
4. **La pila alineada a 16 bytes** en el punto de la llamada. Desalinearla no falla de inmediato: falla dentro de una función de biblioteca que use instrucciones vectoriales, y el error apunta a un sitio que no es el culpable.
5. **Los sufijos de tamaño** (`b`, `w`, `l`, `q`) son obligatorios cuando el tamaño no se deduce de los operandos.

Directivas de datos que se usan en la práctica:

Directiva	Reserva
`.byte`, `.word`, `.long`, `.quad`	1, 2, 4 y 8 bytes con valor
`.asciz`	cadena terminada en cero
`.space n`	`n` bytes a cero
`.align n`	alinea la posición actual

7.3 Práctica 3. Programación mixta C y ensamblador: optimización

Dos formas de mezclar los dos lenguajes.

Módulo separado. Se escribe la función en un `.s`, se declara `.globl` y se enlaza. Desde C basta con declarar el prototipo. Lo único que hay que respetar es el convenio de llamada del tema 2.

```
gcc -c -O2 principal.c -o principal.o
as rutina.s -o rutina.o
gcc principal.o rutina.o -o programa
```

Ensamblador en línea. Con `asm` extendido, sus operandos de salida, de entrada y su lista de registros destruidos:

```
static inline long multiplicar(long a, long b) {
    long r;
    asm ("imulq %2, %1\n\t"
         "movq  %1, %0"
         : "=r" (r)          /* salida */
         : "r" (a), "r" (b)  /* entradas */
         : "cc");            /* destruye las banderas */
    return r;
}
```

Omitir la lista de destruidos es el error grave: el compilador supone que las banderas y los registros que no se declaran siguen intactos, y genera código alrededor que da resultados incorrectos **solo al optimizar**. Un programa que funciona con `-O0` y falla con `-O2` casi siempre tiene aquí su causa.

Medir antes de optimizar. El ejercicio de la práctica es comparar versiones, y para eso hace falta medir bien:

```
gcc -O2 -o prog prog.c
perf stat -e cycles,instructions,cache-misses ./prog
```

`perf` da ciclos, instrucciones y fallos de caché, que son los tres números que los temas 4 y 6 explican. Un cambio que reduce instrucciones y aumenta fallos de caché puede ser más lento, y sin medir no se nota.

Las optimizaciones que el compilador ya hace, y que no hay que escribir a mano: propagación de constantes, eliminación de subexpresiones comunes, desenrollado de bucles, conversión de división por constante en multiplicación, y colocación de variables en registros. Lo que el compilador **no** puede hacer es cambiar el orden de los bucles cuando no puede demostrar que es seguro, ni reorganizar las estructuras de datos. Ahí es donde el programador aporta.

7.4 Práctica 4. Depuradores, desensambladores y editores hexadecimales

GDB. Las órdenes que se usan a bajo nivel:

Orden	Efecto
`layout asm`, `layout regs`	ventanas de ensamblador y registros
`break *0x401136`	punto de ruptura en una dirección
`stepi`, `nexti`	avanzar una instrucción máquina
`info registers`	todos los registros
`x/16xb $rsp`	volcar 16 bytes en hexadecimal desde la pila
`x/8i $rip`	desensamblar 8 instrucciones desde el PC
`p/x $rax`	imprimir un registro en hexadecimal
`set $rax = 5`	modificar un registro
`watch var`	detener cuando una variable cambie

`x/16xb $rsp` es la orden con la que se comprueba de verdad el marco de pila del tema 2: se ve la dirección de retorno, el `%rbp` salvado y las locales, en el orden en que están.

Y el efecto del extremo menor se ve aquí sin ambigüedad: `x/4xb` sobre un entero de valor `0x12345678` muestra `78 56 34 12`.

Desensamblar con `objdump -d -M att`, o `gdb` sobre el binario. Comparar el desensamblado con el `.s` que produjo `gcc -S` cierra el círculo: se ve qué directivas se convirtieron en bytes y cuáles eran solo información para el enlazador.

Editor hexadecimal. `xxd`, `hexdump -C` o `ghex` para inspeccionar y modificar un binario byte a byte. Cambiar un `jle` por un `jmp` en el ejecutable —dos bytes— y ver cómo cambia el comportamiento es el ejercicio que demuestra que el programa es un dato, que era el principio del tema 1.

7.5 Práctica 5. Funcionamiento de la entrada/salida con microcontrolador

Sobre una placa con microcontrolador, donde los registros del tema 5 son accesibles directamente y sin sistema operativo por medio.

Entrada/salida programada. Configurar un puerto como entrada o salida escribiendo en su

registro de dirección, y leer o escribir el registro de datos en un bucle de sondeo.

```
#define PORT_DIR  (*(volatile uint8_t *)0x40020000)
#define PORT_DAT  (*(volatile uint8_t *)0x40020004)

PORT_DIR = 0xFF;                  /* los ocho bits, como salida */
while (1) {
    PORT_DAT = 0x01;
    retardo();
    PORT_DAT = 0x00;
    retardo();
}
```

`volatile` no es opcional. Sin él, el compilador ve dos escrituras al mismo sitio sin lectura entre medias y elimina la primera; el bucle deja de hacer nada visible. Es el ejemplo más limpio de por qué la memoria proyectada de dispositivo no se puede tratar como memoria normal.

Interrupciones. Habilitar la fuente, escribir la rutina, colocar su dirección en la tabla de vectores y habilitar las interrupciones globalmente. La rutina tiene que **reconocer** la interrupción borrando la bandera del periférico; si no lo hace, vuelve a entrar en cuanto sale y el programa se queda dentro de ella para siempre.

Y la regla que atraviesa el tema: las variables compartidas entre la rutina y el programa principal se declaran `volatile`, y los accesos de más de un byte se protegen inhibiendo interrupciones, o se leen a medio actualizar.

Comparación medida. El ejercicio final de la práctica es comparar el tiempo de CPU disponible con sondeo y con interrupciones para el mismo periférico. El número que sale es el argumento del tema 5, comprobado en vez de contado.

7.6 Práctica 6. Análisis de una jerarquía de memoria

Medir el efecto de la caché sobre programas reales.

Los parámetros de la máquina se consultan sin adivinar:

```
lscpu | grep -i cache
getconf -a | grep -i cache
```

Recorrido por filas frente a por columnas. El mismo programa con los dos bucles intercambiados:

```
/* por filas: aprovecha la localidad espacial */
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        s += m[i][j];

/* por columnas: un salto de una fila entre accesos consecutivos */
for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
        s += m[i][j];
```

Con una matriz que no quepa en la caché de último nivel, la diferencia se mide en veces, no en porcentajes. `perf stat -e cache-misses,cache-references` da el número que lo explica.

Multiplicación de matrices por bloques. Dividir en submatrices que quepan en la caché y operar bloque a bloque. El número de operaciones aritméticas es el mismo; lo que cambia es cuántas veces se recorre la memoria.

Recorrido con zancada creciente. Medir el tiempo por acceso al recorrer un vector con zancadas de 1, 2, 4… elementos. La curva tiene escalones, y cada escalón marca un límite: el tamaño de línea de caché primero, y el tamaño de cada nivel después. Es la forma de **deducir la jerarquía midiendo**, sin consultar la documentación.

Falso compartimiento. Dos hilos que incrementan variables distintas situadas en la misma línea de caché. No comparten ningún dato y aun así se frenan mutuamente, porque la línea viaja de un núcleo a otro en cada escritura. Separarlas al tamaño de línea elimina el efecto por completo, y el diagnóstico es puramente el del tema 6.

Los guiones de laboratorio de estas prácticas y los problemas resueltos que las acompañan están en García Carballeira et al., 2015 y en Ortega Lopera et al., 2006; el enfoque de medición desde el programa, en Bryant y O'Hallaron, 2016.

Bibliografía

Bryant, R. E., & O'Hallaron, D. R. (2016). *Computer Systems: A Programmer's Perspective* (3.ª ed.). Pearson.

García Carballeira, F., et al. (2015). *Problemas resueltos de Estructura de Computadores*. Paraninfo.

Hamacher, C., Vranesic, Z., Zaky, S., & Manjikian, N. (2012). *Computer Organization and Embedded Systems* (6.ª ed.). McGraw-Hill.

Harris, S. L., & Harris, D. (2022). *Digital Design and Computer Architecture: RISC-V Edition*. Elsevier.

Ortega Lopera, J., Anguita López, M., & Prieto Espinosa, A. (2006). *Arquitectura de Computadores*. Paraninfo.

Patterson, D. A., & Hennessy, J. L. (2021). *Computer Organization and Design: The Hardware-Software Interface. RISC-V Edition* (2.ª ed.). Morgan Kaufmann.

Stallings, W. (2022). *Computer Organization and Architecture: Designing for Performance* (11.ª ed.). Pearson.