

Estructura de Computadores

Curso 2024-2025

Miguel Ángel De la Vega Rodríguez



Apuntes de Estructura de Computadores



Facultad de Ciencias UGR
Escuela Técnica Ingeniería Informática UGR
Granada

Contents

1	Ensamblador	5
1.1	Historia	5
1.2	Lenguaje C, ensamblador y máquina	5
1.2.1	Proceso de compilación	6
1.2.2	Representación Datos C, IA32, x86-64	8
1.3	Registros, operandos y operaciones	8
1.3.1	Registros	8
1.3.2	Operandos	9
1.3.3	Operaciones	11
1.4	Control	12
1.4.1	Codigos de condición	12
1.4.2	Bucles	15
1.4.3	Secuencias Switch	16
1.5	Procedimientos	17
1.5.1	Stack (Pila)	18
1.5.2	Pasando el Control	18
1.5.3	Pasando los datos	19
1.5.4	Gestionando datos locales	20
1.5.5	Recursividad	20
1.6	Datos	21
1.6.1	Arrays	21
1.6.2	Estructuras	23
1.6.3	Uniones	24
2	Unidad de Control	26
2.1	Introducción	26
2.1.1	Unidad de procesamiento con un bus	28
2.1.2	Unidad de procesamiento múltiples buses	31
2.2	Unidades de control cableadas y microprogramadas	32
2.2.1	Diseño UC cableada	32
2.2.2	Unidad de control microprogramada	34
2.3	Control microprogramado	34
2.3.1	Formato de las microinstrucciones	34
2.3.2	Nanoprogramación	35
2.3.3	Secuenciamiento de microinstrucciones	36

2.3.4	Control residual	39
2.3.5	Camino de datos	40
2.3.6	Diseño Horizontal	41
2.3.7	Diseño Vertical	44
3	Segmentación de Cauce	45
3.1	Concepto de Segmentación	45
3.2	Aceleración	46
3.3	Riesgos	47
3.3.1	Riesgos estructurales	47
3.3.2	Riesgos por dependencias de datos	48
3.3.3	Riesgos de control	48
3.4	Influencia en el repertorio de instrucciones	50
3.5	Funcionamiento superescalar	50
4	Jerarquía de memoria	52
4.1	Abstracción de memoria	52
4.1.1	Lectura y escritura	52
4.2	RAM	53
4.2.1	SRAM	53
4.2.2	DRAM	54
4.3	Configuración y diseño de memorias usando varios chips	55
4.3.1	Ampliación de la memoria	55
4.3.2	Incrementar simultáneamente el número de palabras y el ancho de palabra	56
4.3.3	Módulos de memoria en línea	57
4.3.4	Localidad	58
4.4	Jerarquía de memoria	59
4.4.1	Caché	59
4.4.2	Tipos de Fallos	60
4.5	Tecnologías de almacenamiento	60
4.5.1	Geometría de un disco	61
4.5.2	Capacidad de un disco	61
4.5.3	Funcionamiento de un disco	61
4.5.4	Tiempo de acceso	62
4.5.5	Bus de E/S	63
4.5.6	Memorias no volátiles	63
4.5.7	Discos de estado sólido	64
5	Caché	65
5.0.1	Organización general de Cache	66
5.0.2	Escrituras	69
5.0.3	Por qué indexar los bits intermedios	69
5.1	Políticas de Colocación	71
5.1.1	Correspondencia Directa	71
5.1.2	Correspondencia Totalmente Asociativa	71
5.1.3	Correspondencia Asociativa por Conjuntos	72
5.1.4	Política de reemplazo	72
5.1.5	Métricas para prestaciones de caché	72
5.1.6	Jerarquía de memoria	73

5.1.7	Modelo de evaluación de la jerarquía	73
5.1.8	Caché separada o unificada	75
5.1.9	Impacto del tamaño	75
5.1.10	La montaña de memoria	75
5.2	Código amigable con caché	76
6	Relaciones de problemas	79
6.1	Tema 5-6	79

Ensamblador

1.1 Historia

La línea de procesadores Intel, comúnmente conocida como *x86* ha seguido un largo proceso de evolución, manteniendo compatibilidad hasta 8086, introducido en 1978. La familia de procesadores *x86* ha seguido una evolución hasta el día de hoy:

- 8086 (1978) → 29K transistores, 16 bits, 5 MHz. [Intel 16-bit, IBM PC]
- 386 (1985) → 275K transistores, 32 bits, 16 MHz. [Intel 32-bit]
- Pentium 4E (2004) → 125M transistores, 64 bits, 3.4 GHz. [Intel 64-bit, Hyperthreading, Implementación de *x86-64* de AMD]
- Core i7 Haswell (2013) → 1.4G transistores, 64 bits, 3.9 Ghz [Multicore, AVX2]
- Core i9 Raptor Lake (2022) → +30G transistores, 64 bits, 5.2 GHz [DDR5, PCIe 5.0, AVX-512]

A lo largo de los años, muchas compañías han creado procesadores que se puedan comparar con los de Intel. Entre ellas, AMD ha sido la más destacada, cayendo ligeramente detrás de Intel en cuanto a rendimiento y ofreciendo un precio más competitivo. Sin embargo, en los últimos años, AMD ha sido capaz de ofrecer procesadores con un rendimiento similar o superior a los de Intel, concretamente, en 2024, Intel ha sufrido una gran decadencia por problemas de estabilidad en los últimos lanzamientos. Nosotros nos centraremos en la arquitectura *x86-64*, que es la que se utiliza en el libro de clase, sin embargo, haremos pequeñas menciones a *IA32*, cuando sea conveniente.

1.2 Lenguaje C, ensamblador y máquina

Las computadoras modernas utilizan distintas capas de abstracción, ocultando la complejidad de la implementación. Dos de estas capas son realmente importantes para nosotros, el formato y comportamiento de las instrucciones de máquina, que viene determinado por la ISA (Instruction Set Architecture), esta define el formato de las instrucciones, y el resultado que cada una de estas tendrá sobre el estado de la máquina. La mayoría de ISAs, incluyendo *x86-64*, describen el comportamiento de un programa de forma secuencial, como si se ejecutase una instrucción tras otra, sin embargo, en realidad, el hardware de la CPU es mucho más complejo, ejecutando instrucciones de forma paralelizada o en otro orden, sin embargo, esto se verá con más detenimiento en la asignatura *Arquitectura de Computadores*. La otra capa de abstracción ocurre sobre las direcciones de memoria, se nos presentan como *virtual addresses*, proporcionando un modelo de memoria que aparenta ser un

gran array de bytes, sin embargo, en realidad, la memoria se maneja de otra forma, como se ve en la asignatura *Sistemas Operativos*. Recalcamos aunque no vayan a ser estudiadas en profundidad, que existen otras ISA, como ARM (utilizada en dispositivos móviles), RISC-V (Nueva ISA open-source), o la misma IA32. Aunque no se mencione más, se recomienda al lector la profundización sobre ARM, que actualmente está ampliando su uso en el mercado de portátiles. (Intel Ultra, M1, etc.) Recordamos que podemos clasificar en general las distintas ISAs en dos tipos:

- **CISC** (Complex Instruction Set Computer): Conjunto de instrucciones complejas.
- **RISC** (Reduced Instruction Set Computer): Conjunto de instrucciones reducidas.

Realmente la mayoría de ISAs actuales son híbridas, convirtiendo las instrucciones de CISC en instrucciones de RISC, para mejorar el rendimiento.

El código x86-64 difiere en gran medida del código C, destacamos las siguientes partes del estado del procesador que normalmente no se ven en C:

- **PC (Program Counter)**: Registra la dirección de la siguiente instrucción a ejecutar. Se denomina *%rip* en x86-64.
- **Archivo de registros**: Contiene 16 registros etiquetados con valores de 64 bits. Pueden ser utilizados para almacenar valores temporales, punteros, etc.
- **Códigos de condición**: Contienen información sobre la instrucción más reciente. Se utilizan para realizar saltos condicionales.
- **Pila**: Se utiliza para almacenar enteros o flotantes. Es un array direccionable por bytes.

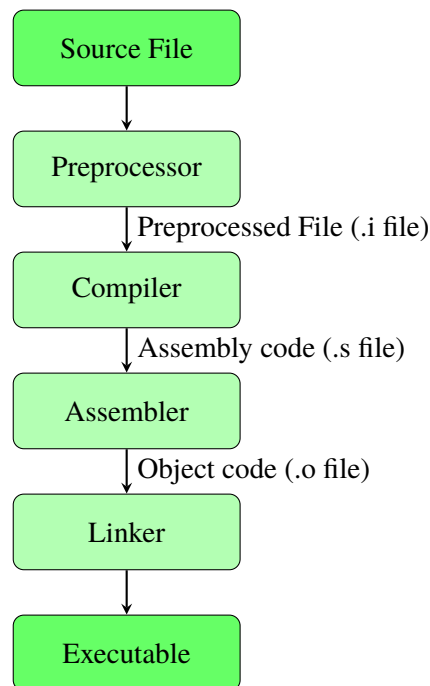
Ahora, vamos a ver cómo se relaciona el código C con el código ensamblador. Para ello, habrá que tener en cuenta que el código generado por el compilador puede variar según el compilador, la versión, las optimizaciones, etc.

1.2.1 Proceso de compilación

Partimos de un número arbitrario de ficheros *.c*, que contienen código C. Para compilarlos, se pueden utilizar distintos compiladores, nosotros usaremos *gcc*:

```
gcc -Og programa.c -o programa
```

Cuando lo ejecutemos:



En este proceso, gcc primero preprocesa el código, generando un archivo *.i*, que posteriormente compila, generando un archivo *.s*, que se ensambla para generar *.o*, finalmente, se enlazan los distintos archivos *.o* para generar el ejecutable. Vamos a explicar un poco más detenidamente cada uno de los pasos:

- **Preprocesador:** C proporciona ciertas herramientas del lenguaje mediante un preprocesador, que es el primer paso en la compilación. Los más utilizados son *#include*, para incluir el contenido de un archivo durante la compilación, y *#define*, para reemplazar un token por una secuencia arbitraria de texto o valores. Además, elimina los comentarios de un archivo fuente. En gcc, podemos generar el código preprocesado con el flag *-E*.
- **Compilación:** El compilador traduce el código preprocesado y genera código ensamblador, con gcc, podemos generar el código ensamblador con el flag *-S*.
- **Ensamblado:** El ensamblador traduce el código ensamblador a código de máquina, generando un archivo *.o*. En gcc, podemos generar el código de máquina con el flag *-c*.
- **Enlazado:** El enlazador combina los distintos archivos *.o* generados en un solo ejecutable. También se encarga de combinar las librerías que se hayan utilizado en el código. En gcc, podemos generar el ejecutable con el flag *-o*.

Ejemplo 1.1: Veamos un ejemplo de cómo se relaciona el código C con el código ensamblador (solo se muestra la parte interesante):

```
long plus(long x, long y);      sumstore:
                                pushq %rbx
void sumStore(long x, long y,   movq %rdx, %rbx
    long *dest) {              call plus
    long t = plus(x, y);        movq %rax, (%rbx)
    *dest = t;                  popq %rbx
                                ret
}
```

■

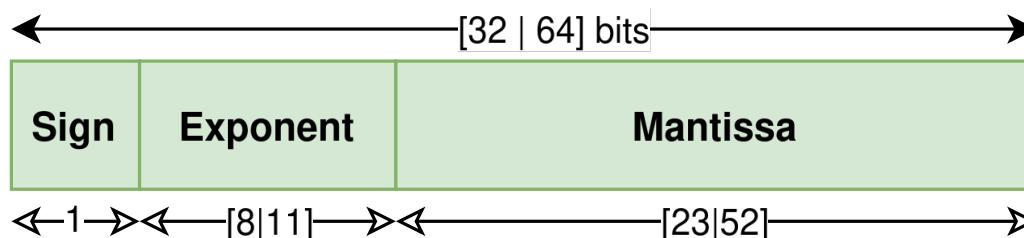
Si lo compilamos en nuestra propia máquina, encontraremos muchas directivas que comienzan por . (ignorar)

1.2.2 Representación Datos C, IA32, x86-64

A continuación, mostramos los distintos tamaños en Bytes para los tipos de dato mas comunes en C. Así como su representación en Assembly. Debido a sus orígenes como una arquitectura de 16 bits que expandió a 32 bits, Intel usa el termino *word* para referirse a un dato de 16 bits, y *doubleword* para referirse a una palabra de 32 bits. En x86-64, se ha añadido el termino *quadword* para referirse a una palabra de 64 bits.

Tipo de dato en C	Assembly sufijo	Normal 32 bits	IA32	x86-64
unsigned		4	4	4
int	l	4	4	4
long	q	4	4	8
char	b	1	1	1
short	w	2	2	2
float	s	4	4	4
double	l	8	8	8
long double		8	10/12	16
char *		4	4	8

Los números de punto flotante se representan en el formato IEEE 754, ya visto en TOC. Se incluye un diagrama de la representación de un número de punto flotante en simple y doble precision:



También cabe mencionar que existe un tipo de dato cuyo tamaño es 80 bits (long double) sin embargo su uso está completamente desaconsejado debido a su portabilidad entre sistemas.

1.3 Registros, operandos y operaciones

1.3.1 Registros

Un procesador x86-64 tiene 16 registros de propósito general, que almacenan valores de 64 bits. Estos registros se usan para almacenar valores temporales, punteros, etc. Sus nombres comienzan con *%r*, sin embargo, siguen convenciones de nombres debido a su legado. El primer 8086 tenía 8 registros de 16 bits, cada uno tenía un uso específico, de ahí su nombre, con la extensión a IA32, los registros se extendieron a 32 bits, y con x86-64, a 64 bits, incluyendo además 8 nuevos registros, cuyos nombres van desde *%r8* hasta *%r15*.

63	31	15	7	0	
%rax	%eax	%ax	%al		Return Value
%rbx	%ebx	%bx	%bl		Callee saved
%rcx	%ecx	%cx	%cl		4th argument
%rdx	%edx	%dx	%dl		3rd argument
%rsi	%esi	%si	%sil		2nd argument
%rdi	%edi	%di	%dil		1st argument
%rbp	%ebp	%bp	%bpl		Callee saved
%rsp	%esp	%sp	%spl		Stack pointer
%r8	%r8d	%r8w	%r8b		5th argument
%r9	%r9d	%r9w	%r9b		6th argument
%r10	%r10d	%r10w	%r10b		Caller saved
%r11	%r11d	%r11w	%r11b		Caller saved
%r12	%r12d	%r12w	%r12b		Callee saved
%r13	%r13d	%r13w	%r13b		Callee saved
%r14	%r14d	%r14w	%r14b		Callee saved
%r15	%r15d	%r15w	%r15b		Callee saved

1.3.2 Operandos

La mayoría de instrucciones tienen uno o más operandos que especifican el *source* y el destino de la operación. En x86-64, los operandos pueden ser de tres tipos, (1) Inmediatos, (2) Registros, (3) Memoria. Veamos la sintaxis para cada uno de ellos junto con la operación que se realiza:

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	(r_i, s)	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Los tipos de datos inmediatos se usan para valores constantes. En AT&T se representan con un signo de dolar, siguiendo la notación usual de C, por ejemplo $\$0xFF$ o $\$323$. Los registros se representan con un signo de porcentaje, por ejemplo, $\%rax$. Como ya hemos visto, los registros son de 64 bits, sin embargo, se pueden realizar operaciones de 32 bits, 16 bits o 8 bits, más adelante veremos cómo. Los operandos de memoria se representan con paréntesis, por ejemplo, $(\%rax)$, $0x8(\%rax)$, etc. Podemos interpretarlo como una desreferenciación de un puntero, es decir, acceder al valor almacenado en la dirección de memoria que se indica.

Ejemplo 1.2: Veamos un ejemplo de cómo se realiza un swap en x86-64:

```
void swap(long *xp, long *yp) swap:
{
    long x = *xp;
    *xp = *yp;
    *yp = x;
}
```

```
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rax, (%rsi)
    movq %rdx, (%rdi)
    ret
```

Podemos ver que inicialmente, la disposición era $\%rdi \rightarrow xp$, $\%rsi \rightarrow yp$. Posteriormente, se almacena el **contenido** de $\%rdi$ en el registro $\%rax$, y el contenido de $\%rsi$ en $\%rdx$. Finalmente, se almacena el contenido de $\%rax$ en la dirección de memoria que apunta $\%rsi$, y el contenido de $\%rdx$ en la dirección de memoria que apunta $\%rdi$. Veamos un ejemplo para entender mejor el funcionamiento de los registros, supongamos que tenemos los siguientes valores:

- **Registro $\%rdi$:** 0x120
- **Registro $\%rsi$:** 0x130
- **Memoria en 0x120:** 125
- **Memoria en 0x130:** 450

Veamos ahora paso por paso el contenido de cada registro(\rightarrow representa el contenido de la dirección de memoria):

Registro	Valor	movq ($\%rdi$), $\%rax$	movq ($\%rsi$), $\%rdx$	movq $\%rax$, ($\%rsi$)	movq $\%rdx$, ($\%rdi$)
$\%rdi$	0x120	0x120	0x120	0x120 \rightarrow 125	0x120 \rightarrow 450
$\%rsi$	0x130	0x130	0x130	0x130 \rightarrow 125	0x130 \rightarrow 125
$\%rax$	—	125	125	125	125
$\%rdx$	—	—	450	450	450

Es posible que el lector no entienda el por qué de las cuatro instrucciones en vez de tres como en C, esto se debe a que, como hemos visto antes, no se pueden realizar operaciones de memoria a memoria, por lo que se necesita un registro para almacenar el valor de una de las direcciones de memoria. ■

1.3.3 Operaciones

Veamos algunas de las operaciones más comunes en x86-64, notemos que algunas de ellas además pueden tener variantes indicando el tamaño de los operandos. Antes de ver las operaciones de forma general, recalamos la instrucción *leaq*:

leaq

La instrucción (*load effective address*) se utiliza como variante de la instrucción *movq*, la instrucción lee de la memoria la dirección efectiva de un operando y la almacena en un registro, sin acceder a la memoria en sí. Por ejemplo *leaq 0x8(%rax), %rdx* almacena en *%rdx* la dirección de memoria *%rax + 0x8*. Esta instrucción es a menudo utilizada por los compiladores para optimizaciones.

Ejemplo 1.3: Veamos una forma para multiplicar un numero por 12 usando *leaq*

```
void m12(long x)          swap:
{                          leaq (%rdi, %rdi, 2), %rax # t = x + 2x
    return x*12;          salq $2, %rax             # return t<<2
}
```

Veamos algunas de las operaciones del tipo aritmetico-lógicas

Instrucción	Efecto	Descripcion
lea S, D	$D \leftarrow \&S$	Load effective address
inc D	$D \leftarrow D + 1$	Increment
dec D	$D \leftarrow D - 1$	Decrement
neg D	$D \leftarrow -D$	Negate
not D	$D \leftarrow \sim D$	Complement
add S, D	$D \leftarrow D + S$	Add
sub S, D	$D \leftarrow D - S$	Subtract
imul S, D	$D \leftarrow D * S$	Multiply
xor S, D	$D \leftarrow D \wedge S$	Exclusive-or
or S, D	$D \leftarrow D \vee S$	Or
and S, D	$D \leftarrow D \& S$	And
sal k, D	$D \leftarrow D \ll k$	Left shift
shl k, D	$D \leftarrow D \ll k$	Left shift (same as sal)
sar k, D	$D \leftarrow D \gg_A k$	Arithmetic right shift
shr k, D	$D \leftarrow D \gg_L k$	Logical right shift

Destacamos una característica peculiar de las operaciones de shift, en el caso de que hagamos un shift a la derecha sobre un registro de 32 bits *shr \$16 %edx, %rdx* se rellenará con 0 en los primeros 32 bits también, de esta forma si teníamos *\$ - 1* por ejemplo y realizamos *shr \$16 %edx*, nos quedaremos con 0x0000 0000 0000 FFFF en vez de 0xFFFF FFFF 0000 FFFF.

Operaciones aritméticas especiales

Multiplicar dos enteros con/sin signo de 64 bits puede tener un resultado que necesite 128 bits para representarlo es para ello que se incluyen las siguientes instrucciones especiales:

Instruction	Effect	Description
<code>imulq S</code>	$R[\%rdx] : R[\%rax] \leftarrow S \times R[\%rax]$	Signed full multiply
<code>mulq S</code>	$R[\%rdx] : R[\%rax] \leftarrow S \times R[\%rax]$	Unsigned full multiply
<code>cqto</code>	$R[\%rdx] : R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Convert to oct word
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx] : R[\%rax] \div S$	Signed divide
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx] : R[\%rax] \div S$	Unsigned divide

Además de esto, añadimos ciertas peculiaridades que nos pueden ser útiles en la práctica:

- Las instrucciones `movzbl` y `movslq` (la *z* indica zero extension y la *s* sign extension) se utilizan para extender cantidades a registros, lo remarcable es que, mientras que `movslq` mueve 32 bits a 64 bits con extensión de signo, `movzbl` no existe ya que al mover 32 bits a 64 bits (`movl`) ya se realiza una extensión de ceros.
- Cuando se tiene el código desensamblado de un programa, podemos ver, la dirección en hexadecimal, junto con varios pares de números en hexadecimal y a la derecha las instrucciones, los pares de números en hexadecimal representan los bytes de la instrucción, por ejemplo si tenemos la línea siguiente:

400544 : e8 08 00 00 00 callq 400551 <f>

Podremos ver que, `e8` corresponde con la instrucción `callq` y los siguientes 4 bytes son la dirección a la que se llama. En este caso $400544 + 8 = 400551$ como se puede ver en la instrucción a la derecha.

1.4 Control

Hasta ahora hemos considerado como se comporta el código cuando se ejecuta de forma secuencial, sin embargo, algunas estructuras de *C* como los bucles, las estructuras de control, etc. requieren de una ejecución condicional, esto es, la secuencia de instrucciones a ejecutar depende de una condición. En x86-64, veremos dos formas de implementar estas estructuras de control más adelante, primero, veamos los códigos de condición que se utilizan en x86-64:

1.4.1 Códigos de condición

Además de los registros de propósito general, la CPU mantiene un conjunto de bits de estado que se utilizan para describir atributos de la última operación aritmético-lógica realizada. Veamos los códigos de condición que vamos a usar:

- CF (Carry Flag):** El flag de acarreo, se activa si hay un acarreo en la última operación. Se utiliza para detectar overflow en operaciones sin signo.
- ZF (Zero Flag):** El flag de cero, se activa si el resultado de la última operación es cero.
- SF (Sign Flag):** El flag de signo, se activa si el resultado de la última operación es negativo.
- OF (Overflow Flag):** El flag de desbordamiento, se activa si hay un desbordamiento en la última operación en complemento a 2.

Estos códigos de condición se ajustan implícitamente por las operaciones aritméticas, de esta forma, *leaq* no afecta a los códigos de condición. Además de ajustarse por las operaciones que ya conocemos, existen instrucciones que ajustan explícitamente los códigos de condición sin alterar los registros de propósito general.

- El conjunto de instrucciones **Compare** (*cmp*) ajustan los códigos de condición de acuerdo a las diferencias entre sus dos operandos. Debemos tener en cuenta que en *AT&T* la sintaxis es al revés: *cmpq S1, S2* compara *S1* con *S2* mediante $S2 - S1$.
- El conjunto de instrucciones **Test** (*test*) ajustan los códigos de condición de acuerdo a las operaciones lógicas entre sus dos operandos. Se realiza la operación $S1 \& S2$ y se ajustan los códigos de condición. Es común que se utilicen ambos operandos iguales, por ejemplo, para comprobar si un número es 0.

Consultar códigos de condición

En vez de consultarlos directamente, se utilizan mediante tres formas:

1. Se establece un único byte a 0 o 1 en función de una combinación de los códigos de condición.
2. Se utilizan para realizar saltos condicionales.
3. Se utilizan para transferir datos condicionalmente.

Para la primera forma, se utilizan las instrucciones *setcc* siguientes:

Instruction	Effect
<i>sete D</i>	$D \leftarrow ZF$ (Equal / zero)
<i>setne D</i>	$D \leftarrow \sim ZF$ (Not equal / not zero)
<i>sets D</i>	$D \leftarrow SF$ (Negative)
<i>setns D</i>	$D \leftarrow \sim SF$ (Nonnegative)
<i>setg D</i>	$D \leftarrow \sim (SF \oplus OF) \& \sim ZF$ (Greater, signed >)
<i>setge D</i>	$D \leftarrow \sim (SF \oplus OF)$ (Greater or equal, signed \geq)
<i>setl D</i>	$D \leftarrow SF \oplus OF$ (Less, signed <)
<i>setle D</i>	$D \leftarrow (SF \oplus OF) \vee ZF$ (Less or equal, signed \leq)
<i>seta D</i>	$D \leftarrow \sim CF \& \sim ZF$ (Above, unsigned >)
<i>setae D</i>	$D \leftarrow \sim CF$ (Above or equal, unsigned \geq)
<i>setb D</i>	$D \leftarrow CF$ (Below, unsigned <)
<i>setbe D</i>	$D \leftarrow CF \vee ZF$ (Below or equal, unsigned \leq)

Ejemplo 1.4: Veamos un ejemplo de su uso:

```

                                gt:
long gt(long a, long b)        cmpq %rsi, %rdi
{                               setg %al
    return a > b;              movzbq %al, %rax
}                               ret

```

Donde *movzbq* es una instrucción que significa *Move with Zero-extend Byte to Long*. ■

Veamos ahora cómo se realizan los saltos condicionales. Estos provocan que la ejecución del programa pueda saltar a otra parte del código en función de los códigos de condición, es muy posible que en algún momento nos encontremos una instrucción en C de tipo *goto*, su uso está desaconsejado, sin embargo, por su similitud con el ensamblador, puede resultar útil en ciertas situaciones. Veamos una tabla con las instrucciones de salto condicional y después un ejemplo de uso.

Instruction	Jump Condition	Description
jmp	Label	Direct jump
jmp	*Operand	Indirect jump
je	ZF	Equal / zero
jne	\sim ZF	Not equal / not zero
js	SF	Negative
jns	\sim SF	Nonnegative
jg	\sim (SF ^ OF) & \sim ZF	Greater (signed >)
jge	\sim (SF ^ OF)	Greater or equal (signed >=)
jl	SF ^ OF	Less (signed <)
jle	(SF ^ OF) ZF	Less or equal (signed <=)
ja	\sim CF & \sim ZF	Above (unsigned >)
jae	\sim CF	Above or equal (unsigned >=)
jb	CF	Below (unsigned <)
jbe	CF ZF	Below or equal (unsigned <=)

Ejemplo 1.5: Veamos ahora un ejemplo de cómo hacer por ejemplo un valor absoluto de una diferencia:

```

                                abs:
long abs(long x, long y)
{
    if (x > y)
        return x - y;
    else
        return y - x;
}

                                cmpq %rsi, %rdi
                                jle .L1
                                movq %rdi, %rax
                                subq %rsi, %rax
                                ret
                                .L1:
                                movq %rsi, %rax
                                subq %rdi, %rax
                                ret

```

Si tratamos de replicar el ejemplo anterior, veremos que en general, el compilador no produce este código, esto se debe a las optimizaciones que el compilador aplica, evitando en la medida de lo posible los saltos condicionales, ya que muchas veces, es mejor calcular ambas ramas, a tener que predecir y calcular cual de las ramas hay que seguir. Para obtener este mismo código, deberemos especificar la opción de compilación *-fno-if-conversion*. ■

Nos detenemos brevemente en las optimizaciones mencionadas, lo que hace el compilador en vez de saltar ramas es asignar valores a las variables en función de un test: `let val = test ? x : y`. gcc intenta utilizar este tipo de optimizaciones siempre que sea seguro. De esta forma, el código anterior se traduciría a:

```

abs:
    movq %rdi, %rax # x
    subq %rsi, %rax # x - y
    movq %rsi, %rdx
    subq %rdi, %rdx # y - x
    cmpq %rsi, %rdi # x : y
    cmovle %rdx, %rax # si x <= y, rax = rdx

```


Esto se verá como una optimización en la asignatura de *Arquitectura de Computadores*, sin embargo, adelantamos que por ejemplo `let val = test ? p : *q` no se puede optimizar de la misma forma, ya que pudiera tener efectos no deseables. Notemos además que este tipo de optimizaciones solo tienen sentido cuando los cálculos son sencillos, en cuyo caso es preferible calcular las dos ramas a predecir cual de ellas seguir. Es por este tipo de opciones que `gcc` es posible que genere un código mas lento cuando se activan las optimizaciones `-O3` ya que si la rama era muy predecible, la predicción de saltos es mas rápida que calcular las dos ramas. Tampoco se debe utilizar en casos donde una rama u otra tiene efectos colaterales, ya que eso pudiera cambiar el comportamiento del programa.

1.4.2 Bucles

La mayoría de lenguajes ofrecen varias formas de implementar bucles: *do-while*, *while*, *for*, *loop*, etc. En x86-64, no existen instrucciones similares, en su lugar, se combinan diferentes tests y saltos condicionales para implementar los bucles.

Bucles Do-While

En C, un bucle *do-while* se implementa de la siguiente forma:

```
long pcount(long x)
{
    long result = 0;
    do
    {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

Versión C

```
long pcount(long x)
{
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

Versión goto

En este ejemplo se ha hecho un clásico programa popcount que cuenta el número de bits a 1 en un número. El nombre *popcount* viene de *population count*, que es el nombre que se le da a esta operación. Como ya sabemos, el efecto del loop es ejecutar el cuerpo del bucle, evaluar la condición y si el resultado de la condición es distinto de 0, seguir ejecutando el bucle. Notemos que se ejecuta al menos una vez.

Bucles While

Ejemplo 1.6: Veamos ahora cómo se implementa un bucle *while*:

```

long pcount(long x)
{
    long result = 0;
    while(x) {
        result += x & 0x1;
        x >>= 1;
    }

    return result;
}

```

Versión C

```

long pcount(long x)
{
    long result = 0;
    goto test;
loop:
    result += x & 0x1;
    x >>= 1;
test:
    if (x) goto loop;
    return result;
}

```

Versión goto

■

En general, *while* evalúa la condición antes de ejecutar el bucle, por lo que si la condición no se cumple, el bucle no se ejecuta.

Bucles For

Sabemos ya que en general un bucle *for* se compone de tres partes, la inicialización, la condición y la actualización. No adjuntamos el código en C ya que es trivial sabiendo los anteriores, notemos que un bucle *for* es equivalente a un bucle *while* de la siguiente forma: *for (init; test; update) body* es equivalente a *init; while(test) body; update;* Por último, veamos las secuencias *switch*.

1.4.3 Secuencias Switch

Una secuencia *switch* proporciona una ramificación condicional múltiple basada en el valor de una expresión entera. Son muy útiles cuando trabajamos con tests que pueden tener una gran cantidad de resultados, haciendo el código más legible y eficiente. Este último punto puede sorprender, por ello, vamos a desarrollarlo.

Una *tabla de salto* es una estructura de datos en forma de array donde *i* es la dirección de un segmento de código que ejecuta la acción que el programa debería seguir cuando el valor de la expresión es *i*. La diferencia entre múltiples *if-else* y *switch* radica en que el tiempo que se tarda en ejecutar un *switch* es constante, mientras que en un *if-else* es lineal en función del número de casos.

Código en C

```
void switch_eg(long x) {
    switch (x) {
        case 1:
            printf("Code 1\n");
            break;
        case 2:
            printf("Code 2\n");
            break;
        case 3:
            printf("Code 3\n");
            break;
        default:
            printf("Code 4\n");
    }
}
```

Assembly Code

```
cmpq $3, %rdi
ja .L4 # Si x > 3, Default
jmp *.L1(,%rdi,8) # Dirección L1 + 8x
.L1:
.L2:
.L3:
.L4 # Default
```

Branch Table (Jump Table)

Índice	Dirección de Salto
1	Code Block 1
2	Code Block 2
3	Code Block 3
Default	Code Block 4

Como podemos ver, la tabla de saltos comienza en este caso en .L1, para acceder a el resto de bloques de código, se suma 8 veces el valor de x, ya que cada dirección ocupa 8 bytes. Si x es mayor que 3, se salta a la dirección .L4, que es el bloque por defecto. Si por ejemplo en vez de hacer break, se quisiera hacer un fall through, obtendríamos algo del siguiente estilo:

```
long w = 1;
switch(x) {
    ...
    case 2:
        w = y/z;
        /* Fall through */
    case 3:
        w += z;
        break;
    ...
}
```

$\%rdi \rightarrow x$ $\%rsi \rightarrow y$ $\%rdx \rightarrow z$

```
.L5: # Case 2
movq %rsi, %rax
cqto # Convert Quad To Oct
idivq %rdx
jmp .L6
.L9: # Case 3
movl $1, %eax
.L6: # Merge
addq %rcx, %rax # w += z
ret
```

1.5 Procedimientos

Los procedimientos son una abstracción clave en el desarrollo de software, permiten dividir un programa en pequeñas partes donde cada una de ellas proporciona una funcionalidad concreta al proporcionarle una serie de argumentos. Pueden ser llamados desde cualquier parte del programa, lo que permite reutilizar el código y hacerlo más modular. Los procedimientos vienen de distintas formas en cada lenguaje: funciones, métodos, subrutinas, lambdas, etc, pero todos ellos comparten ciertos elementos comunes.

Existen muchos atributos que hay que manejar cuando se proporciona soporte en lenguaje ensamblador para procedimientos, para verlos, supongamos un proceso P que llama a un procesador Q , que se ejecuta y tras esto vuelve a P , estas acciones incluyen:

- **Pasar Control:** El PC tiene que fijarse en la dirección de comienzo de Q , y tras esto, cuando Q termina, el PC tiene que volver a la dirección de la instrucción que sigue a la llamada.
- **Parámetros:** P tiene que pasar los parámetros a Q , y Q tiene que devolver un valor a P .
- **Reserva y Limpieza de Memoria:** Q puede necesitar reservar memoria para sus propias variables, y tras la ejecución, limpiarla.

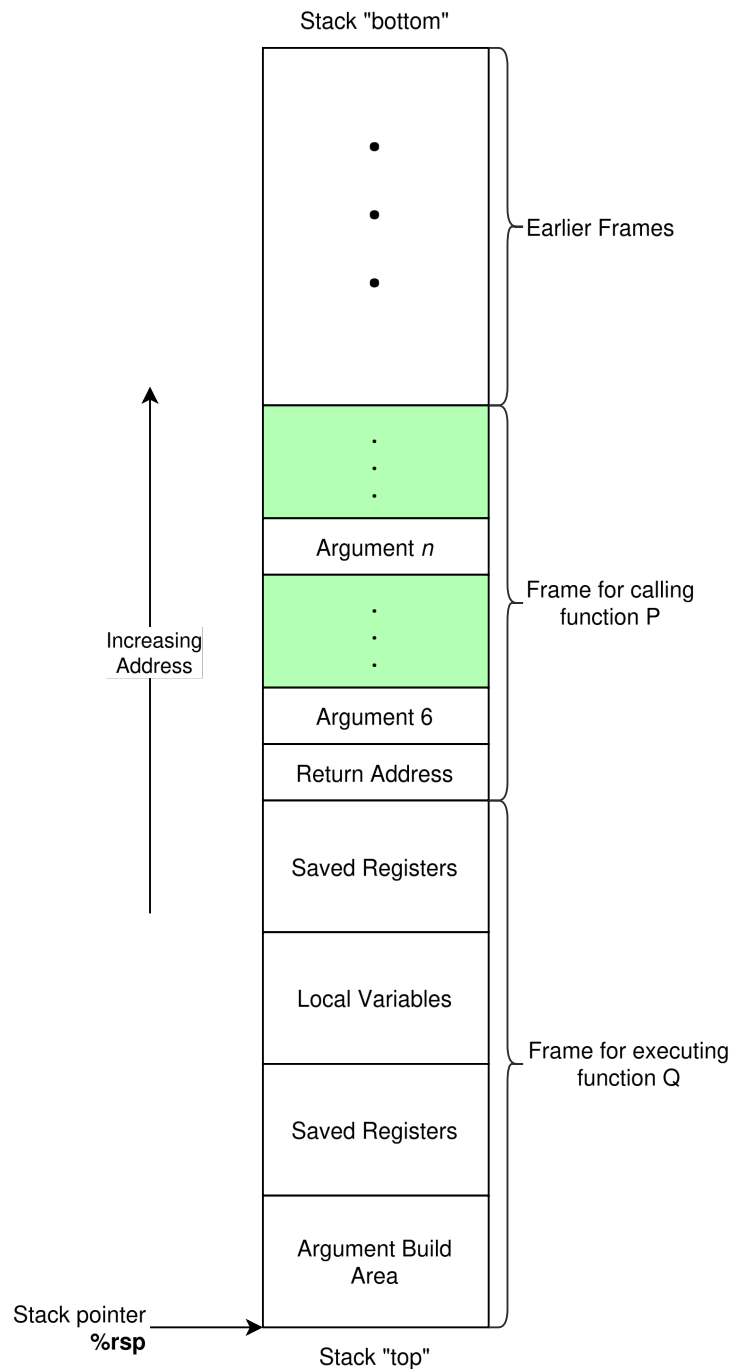
La implementación de estos procedimientos en x86-64 involucra una combinación de instrucciones especiales y convenciones de cómo usar los recursos de la máquina. Detrás de esto hay un gran esfuerzo por minimizar el overhead que involucra una llamada, esto provoca que al final, solo se ejecuta lo mínimo indispensable mencionado antes. Veamos ahora diferentes mecanismos paso por paso, comenzando por la pila:

1.5.1 Stack (Pila)

Es una región de memoria gestionada con aspectos de LIFO, es decir, el último elemento en entrar es el primero en salir, crece hacia direcciones inferiores, en x86-64 el registro `%rsp` contiene la dirección mas baja de la pila. Los datos, pueden ser accedidos y extraídos de la pila usando las instrucciones `pushq` y `popq`. Por ejemplo para reservar espacio para datos sin valor inicial, se puede decrementar la pila la cantidad necesaria, análogamente, se puede liberar el espacio incrementando el stack pointer. En x86-64, cuando un procedimiento requiere más espacio de lo que puede guardar en registros, reserva espacio en el stack. A esta región se la llama *procedure space* para las variables locales. En interés del espacio y la eficiencia, los procedimientos de x86-64 reservan solo el espacio necesario.

1.5.2 Pasando el Control

Pasar el control de una función P a otra Q requiere únicamente de poner el PC en la dirección de inicio de Q , sin embargo, cuando Q termina, el PC tiene que tener alguna manera de saber dónde debe reanudar la ejecución de P , esta información se guarda en las máquinas x86-64 usando la instrucción `call` Q . Esta instrucción añade una dirección A al stack y pone el PC en el comienzo de Q . La dirección A se denomina *return address* y se calcula como la instrucción que sigue inmediatamente a `call`. La instrucción contraria `ret`, elimina una dirección A de la pila y pone el PC en A . (Notemos que estas instrucciones nos las encontraremos como `callq` y `retq`, donde el sufijo q simplemente enfatiza que son versiones de x86-64 no de IA32). Veamos un esquema con la estructura general del stack:



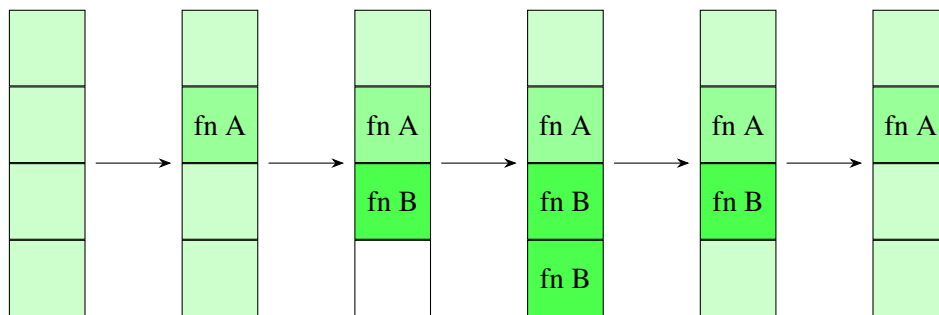
1.5.3 Pasando los datos

Además de pasar el control a un procedimiento cuando es invocado, y de vuelta cuando termina, las llamadas a procedimientos pueden involucrar pasar datos como argumentos, además un procedimiento puede devolver también un valor. En x86-64, la mayoría de este flujo de datos es llevada a cabo mediante registros, ya hemos visto numerosos ejemplos en los que `%rdi` y `%rsi` se pasan como parámetros y valores que se devuelven mediante `%rax`. En x86-64, se pueden pasar hasta 6 argumentos mediante registros, que se usan en un orden específico con el nombre del registro en función del tamaño del dato que contiene. Cuando una función tiene más de 6 argumentos, los argumentos restantes se pasan en la pila tal que el argumento 7 se coloca abajo, el 8 encima, y

así sucesivamente.

1.5.4 Gestionando datos locales

Hasta ahora, la mayoría de ejemplos que hemos visto no han requerido de variables locales, ya sea por optimización del compilador o por la sencillez del código, sin embargo, sabemos que muchas veces es necesario almacenar variables locales. Normalmente, en lenguajes basados en pila, o que soportan la recursión (e.g C, C++, Java, Rust, etc), las variables locales se almacenan en la pila. Esta pila, se reserva en marcos (allocated in frames), que son bloques de memoria que contienen las variables locales de un procedimiento, estos marcos de memoria se organizan tal que contienen información de retorno, almacenamiento local si es necesario y espacio temporal si es necesario, además hay un puntero de marco opcional: `%rbp`. Veamos un ejemplo de cadena de llamadas a procedimientos:



Donde en cada uno de los pasos, `%rsp` apunta en la función más abajo existente y `%rbp` apunta al marco de la función inmediata anterior. Los registros de programa actúan como un único recurso compartido por todos los procedimientos, aunque un solo procedimiento puede estar activo en un momento dado, tenemos que asegurarnos de que cuando un procedimiento (calle) llama otro procedimiento (callee), el calle no sobrescriba los registros del calle. Veamos un ejemplo de esto:

```
funcA:
    movq $15213, %rdx
    call funcB
    addq %rdx, %rax
    ret

funcB:
    subq $18213, %rdx
    ret
```

Como podemos ver, pudiera pasar que `%rdx` fuese sobrescrito en la función `funcB`, para evitar esto, existen convenciones para coordinar el uso de registros. Los registros `%rbx`, `%rbp`, `%r12` – `%r15` se denominan *callee-saved* ya que el calle debe guardar su valor si los modifica. Para este fin, el calle puede o bien no modificar los registros, o guardando su valor en la pila y restaurándolo antes de salir, a este espacio se le llama *Saved Registers*, podemos verlo en el diagrama 1.5.2. El resto de registros, excepto `%rsp`, se clasifican como *Caller-saved*, lo que significa que pueden ser modificados por cualquier función, el nombre viene de un procedimiento *P* que llama a un *Q*, como *Q* puede modificar libremente estos registros, *P* debe guardar su valor si los necesita.

1.5.5 Recursividad

Las convenciones que hemos visto hasta ahora son suficientes para permitir a los procedimientos de x86-64 llamarse a sí mismos. Cada llamada a procedimiento tiene su propio espacio en el stack, luego las variables locales de las múltiples llamadas no interfieren entre ellas. Veamos un ejemplo del código C y ensamblador generado para una función recursiva:


```

long pcount_r(unsigned long x) {
    if (x == 0) {
        return 0;
    } else {
        return (x & 1)
            + pcount_r(x >> 1);
    }
}

pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L2
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L2
    rep; ret # Para branch prediction

```

Veamos más detenidamente lo que está pasando:

- Las primeras tres instrucciones sirven para saber cuando acaba la recursión, si x es 0, se devuelve 0.
- El `pushq` está ya que se va a modificar `%rbx` y se necesita restaurar su valor al salir de la función.
- Desde `movq` hasta `shrq` se calcula $x \& 1$ y $x \gg 1$.
- $x \gg 1$ se pasa como argumento a la función `pcount_r` (llamada recursiva)
- Se suma el resultado de la llamada recursiva con el valor de $x \& 1$.

1.6 Datos

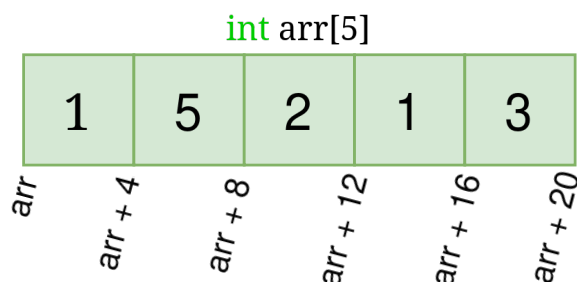
1.6.1 Arrays

Los arrays en C se usan para almacenar colecciones de datos escalares en variables más grandes, la implementación particular de C de los arrays es muy directa, una característica importante es que en C podemos manipular punteros a elementos de un array y realizar operaciones con ellos. Los compiladores son extremadamente buenos simplificando el cálculo de direcciones usado cuando accedemos a un array, es por ello que la correspondencia entre C y ensamblador puede llegar a ser difícil de descifrar.

Principios Básicos

Sea T un tipo de dato cualquiera, y N una constante, consideramos la declaración $TA[N]$, entonces A es un array de N elementos. La declaración reserva una región contigua de memoria de $T \cdot N$ bytes e introduce un identificador A que se usa como puntero al comienzo de esta región de memoria. Los elementos de este array se acceden mediante el uso de un índice entre 0 y $N-1$.

Ejemplo 1.7: Consideramos `int arr[5]`, veamos los siguientes ejemplos:



- `arr[4]` (int) se refiere al quinto elemento del array (valor 3).
- `arr` (int *) comienzo del array (posición `arr`)
- `arr+1` (int *) comienzo del array + 1 (posición `arr + 4`)
- `&arr[2]` (int *) dirección de `val[2]` (posición `arr + 8`)
- `arr[5]` (int) fuera de los límites del array, **undefined behavior**.
- `*(arr+1)` (int) valor de la dirección `val + 1` (valor 5)

■

Destacamos que declaramos arrays inmediatamente después de otro, no está garantizado que estos se almacenen en memoria de forma contigua. Así como hemos podido ver en el ejemplo anterior, que C permite operaciones con los punteros, así, si p es un puntero a un array de tipo T y la dirección de memoria de p es x_p , entonces la expresión $p + i$ tiene el valor $x_p + T \cdot i = i \cdot \text{sizeof}(T)$. Recordamos que los operadores `&` y `*` permiten la generación y dereferenciación de punteros respectivamente.

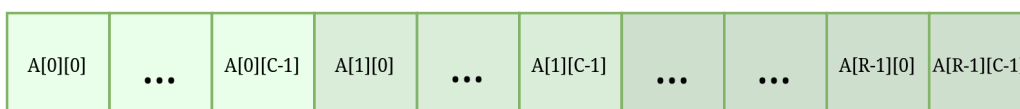
Ejemplo 1.8: Ejemplo de acceso a elementos de array:

```
int get_num(int *arr, int i)           get_num:
{                                     movslq %esi, %rsi
    return arr[i];                   movl (%rdi, %rsi, 4), %eax
}
```

Recordamos que `movslq` es una instrucción que significa *Move with Sign-extend Long to Quad*, se utiliza ya que para hacer operaciones con la dirección de memoria, es necesario que los registros sean de 64 bits. ■

Arrays Anidados

Los principios de los arrays multidimensionales son los mismos que los de los arrays unidimensionales, de esta manera, la declaración `T A[R][C]` declara un array de R elementos, cada uno de los cuales es un array de C elementos. En general, un array bidimensional de elementos de tipo T . De otra forma, podemos decir que tiene R filas y C columnas. Notemos que si el tamaño de T es K bytes, entonces el tamaño de A es $C \cdot R \cdot K$ bytes. La disposición de los elementos en memoria es por filas, (*row-major-order*):



De esta forma, si tenemos un array `T A[R][C]` el elemento `T[i][j]` está en la posición de memoria `&A[i][j]` $= x_A + L(C \cdot i + j)$.

Ejemplo 1.9: Veamos un ejemplo en el que el flag `-no-pie` debe ser considerado

```
#define ROWS 4
// Zip_dig -> Array de 5 elementos
int *get_row(int index)
{
    return arr[index];
}

zip_dig arr[ROWS] = {
    {1, 2, 7, 4, 5},
    {6, 7, 2, 9, 1},
    {1, 9, 3, 1, 5},
    {1, 7, 8, 9, 2}};
```

```

get_num:
    leaq (%rdi, %rdi, 4), %rdx
    leaq arr(, %rdx, 4), %rax
    ret

get_num:
    movslq %edi, %rdi
    leaq arr(%rdi, %rdi, 4), %rdx
    leaq arr(%rip), %rax
    leaq (%rax, %rdx, 4), %rax
    ret

```

A la izquierda, como se usa `–no-pie` que desactiva el *Position Independent Code*, la dirección de memoria es conocida en tiempo de compilación, por lo que se puede acceder directamente con `5 · index` por ser 5 elementos por fila y `4 · index` por ser un tipo de dato entero de 4 bytes. En el segundo caso, el direccionamiento es relativo a contador de programa `%rip`. Notemos que `–no-pie` resulta más eficiente sin embargo es más vulnerable a ataques de buffer overflow. ■

Array Multinivel

De manera similar a los arrays bidimensionales, tenemos los arrays multinivel, que tienen la peculiaridad de que se tiene un array de punteros a arrays, de esta forma, para acceder a un dato son necesarios dos accesos a memoria, uno para acceder al puntero y otro para acceder al dato.

Ejemplo 1.10: Un ejemplo de array multinivel en C:

```

zip_dig cmu = {0, 5, 2, 7, 3};
zip_dig mit = {1, 2, 3, 4, 5};
zip_dig ucb = {9, 3, 7, 5, 1};

# define UCOUNT 3
int *univ[UCOUNT] = {cmu, mit, ucb};

```

1.6.2 Estructuras

C permite dos formas de definir tipos de datos compuestos: *struct* que agrupan múltiples datos en un solo objeto y *union* que permiten que un objeto sea referenciado usando diferentes tipos de datos. Veamos en detalle las estructuras.

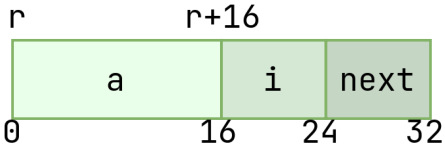
El struct en C crea un tipo de dato que agrupa múltiples variables que pueden ser de diferentes tipos en un solo objeto, los distintos elementos de la estructura se referencian mediante nombres, con la sintaxis `struct.field`, `struct->field`. El struct se parece a los arrays en el sentido de que se garantiza que los datos de un mismo struct están contiguos en memoria. Es el compilador el que mantiene información sobre cada estructura para indicar el offset de cada campo. El tamaño de los structs es el tamaño de todos sus campos sumados, teniendo en cuenta que es posible que el compilador reserve espacio adicional tras un tipo de dato para alinear la memoria.

Ejemplo 1.11: Veamos un ejemplo de struct en C y cómo se distribuye su memoria:

```

struct rec {
    int a[4];
    size_t i;
    struct rec *next;
}

```



Notemos que si en el struct el array de enteros fuese de tamaño 3, el tamaño del struct se mantendría en 32 B por el alineamiento de la memoria. El puntero `r` indica el primer byte de la estructura, se accede mediante el offset, que como hemos dicho lo determina el compilador en tiempo de compilación.

```

void set_i(struct rec *r, size_t val)
{
    r->i = val;
}

set_i:
    movq %rsi, (%rdi, 16)
    ret

```

■

Hemos introducido ya el alineamiento de los datos, detallamos los principios que rigen su funcionamiento:

- Si el tipo de dato primitivo requiere K bytes, la dirección debe ser múltiplo de K .
- El alineamiento es requisito en algunas maquinas, en x86-64 es recomendado por motivos de eficiencia (Es ineficiente cargar o almacenar datos que cruzan fronteras de quad word y la memoria virtual es delicada cuando un dato se extiende a 2 páginas).
- En la mayoría de lenguajes compilados (C, C++, Rust, etc) es el compilador el que inserta huecos en la estructura para alinear los campos.
- El alineamiento se hace:
 - Para el struct, se toma el máximo valor miembro para alinear, esto se llama alineamiento externo.
 - Para los datos primitivos, se toma el tamaño del dato, es decir se añade padding antes del dato miembro si fuese necesario para que se encuentre en una posición múltiplo de su tamaño.

El compilador no reordena los datos con el fin de ahorrar espacio, sin embargo el programador debería considerar siempre cómo ordenar los datos para ocupar el menor espacio posible, esto es especialmente importante ya que a diferencia de otras optimizaciones que el compilador hace por nosotros, esta no se puede hacer ya que rompería la coherencia y predictibilidad del código.

1.6.3 Uniones

Las uniones proporcionan una manera de circunvalar el sistema de tipos de C, permitiendo a un solo objeto ser referenciado por múltiples tipos. La sintaxis de la unión es idéntica a la de un struct, pero sus semánticas son diferentes. En una unión, todos los campos comparten la misma dirección de memoria, lo que significa que si se modifica un campo, se modifica el resto, para hacer esto, el compilador reserva tamaño de acuerdo al elemento más grande.

Ejemplo 1.12: Veamos un ejemplo de una unión en comparación con un struct:

```

union U1 {
    char c;
    int i[2];
    double v;
} *up;

struct S1 {
    char c;
    int i[2];
    double v;
} *sp;

```

Si compilamos ambos códigos en una máquina Linux en x86-64, el offset y tamaño total de las estructuras sería el siguiente:

Tipo	c	i	v	Total Size
U1	0	0	0	8
S1	0	4	16	24

Este gran poder conlleva una gran responsabilidad, ya que el programador debe ser consciente de que si usa datos como float e int en la misma unión, e intenta interpretar los bits del entero como si fuese un float, el resultado no será el esperado, ya que no tendremos el mismo efecto que el cast (float) sobre el entero. ■

Recordamos que las palabras se almacenan en memoria como un número N de bytes, en esta organización, el byte mas significativo puede ser el que está en la dirección más baja (Big Endian) o en la dirección más alta (Little Endian). Esta diferencia puede causar problemas al intercambiar datos binarios entre máquinas, también mencionamos Middle endian, que son aquellas arquitecturas que se pueden configurar de ambas maneras, aparece en ARM, sin embargo, la mayoría de las máquinas modernas son Little Endian.

Para finalizar este capítulo, presentamos un resumen de los tipos compuestos en C:

- **Arrays**
 - Reservan memoria contigua para almacenar elementos
 - Se usa aritmética de indexación para localizar elementos individuales
 - Puntero al primer elemento
 - No comprueban límites
- **Structs**
 - Reserva de una sola región de memoria, los campos en el mismo orden declarado
 - Se accede usando desplazamientos determinados por el compilador
 - Alineamiento interno y externo mediante relleno
- **Unions**
 - Declaraciones superpuestas
 - Proveen de una manera para saltarse el sistema de tipos de C

Unidad de Control

2.1 Introducción

Recordamos que un computador con arquitectura Von Neumann consta de tres bloques fundamentales, la CPU o procesador, la memoria principal y las unidades de entrada/salida. Todos estos están unidos mediante buses (de datos, direcciones y control). En este tema nos centraremos en la CPU (Central Processing Unit), podemos entenderla como una unidad constituida por

- Unidad de procesamiento o camino de datos (datapath)
- Unidad de control

La unidad de procesamiento comprende elementos hardware como:

- Unidades funcionales (ALU, desplazador, multiplicador, etc.)
- Registros:
 - Registros de propósito general (General Purpose Registers)
 - Registro de estado (flag registers)
 - Program Counter (PC)
 - Registro de Instrucción (IR)
 - Registro de dato de memoria (MDR / MBR)
 - Registro de dirección de memoria (MAR)
- Multiplexores
- Buses internos
- ...

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

- **Secuenciamiento de las instrucciones:**

El PC almacena la dirección de la siguiente instrucción en memoria, al comienzo del ciclo de instrucción, la UC coloca el valor de PC en el bus de direcciones para acceder a la memoria, tras esto, la UC

activa una señal de lectura de memoria para acceder a la dirección especificada por el PC. Esta instrucción se transfiere desde la memoria hasta el registro de instrucción IR y se incrementa el PC en función del tamaño de las instrucciones (viene dado por la arquitectura) $IR \leftarrow M[PC]$. Este paso puede repetirse varias veces si la instrucción consta de varias palabras.

- **Ejecución/Interpretación de las instrucciones en IR:** La UC utiliza un decodificador de instrucciones que convierte los bits de la instrucción en señales de control específicas. La instrucción contiene campos específicos que indican la operación (e.g ADD, LOAD, JUMP) y sus operandos. Estos son separados y analizados por el decodificador. Una vez decodificado, se generan y mandan señales para ejecutar la instrucción y para enviar los resultados a su destino.

Este proceso además puede superponerse en el llamado *pipeline* permitiendo que se ejecuten varias instrucciones simultáneamente. En cada ciclo de reloj, una instrucción distinta puede estar en cada fase, mejorando la eficiencia.

Ejemplo 2.1: Veamos como se ejecutan las instrucciones LOAD y SUM en un estilo RISC-V simple:

Load R5, X(R7)

Involucra las siguientes acciones:

- Leer la instrucción de memoria.
- Incrementar el PC.
- Decodificar la instrucción para determinar la operación a ejecutar.
- Leer el registro R7.
- Sumar la dirección X al contenido de R7.
- Usar la suma como la dirección efectiva para el source de la operación, y leer los contenidos de esa dirección en la memoria.
- Cargar los contenidos recibidos en el registro destino R5.

Dependiendo de cómo se organice el hardware, algunas de estas operaciones pueden realizarse al mismo tiempo. Dividiremos el proceso en 5 etapas ya que esta forma es la más común en procesadores de estilo RISC-V. En el caso anterior, podemos dividir el proceso en las siguientes etapas:

1. Leer la instrucción e incrementar el PC.
2. Decodificar la instrucción y leer los contenidos del registro R7 en el archivo de registros.
3. Calcular la dirección efectiva.
4. Leer el source de la operación de la memoria.
5. Escribir el resultado en el registro destino R5.

Las instrucciones que conllevan una operación aritmético-lógica pueden ejecutarse usando pasos similares. Respecto a LOAD difieren que no se requieren operaciones de acceso a memoria. Por ejemplo la instrucción ADD R3,R4,R5 requiere de los pasos:

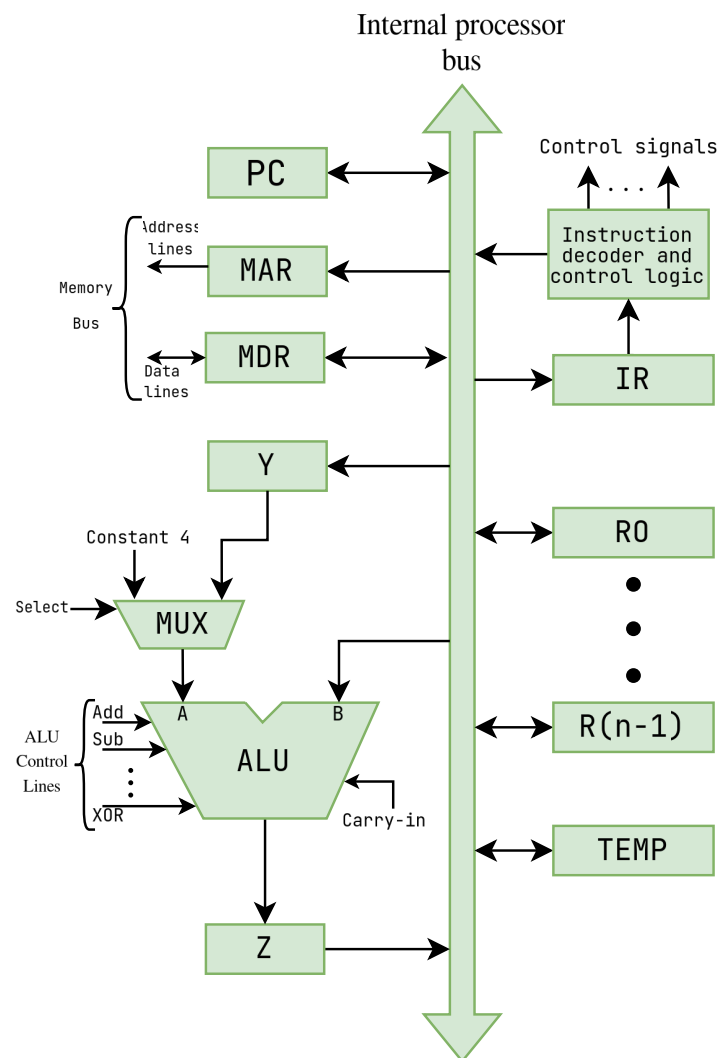
1. Leer la instrucción e incrementar el PC.
2. Decodificar la instrucción y leer los contenidos de los registros origen R4 y R5.

3. Realizar la operación aritmético-lógica.
4. nop (No action)
5. Escribir el resultado en el registro destino R3.

Como podemos ver, se puede completar en 4 pasos, sin embargo, se obtienen ventajas si usamos las mismas etapas de procesamiento en la mayoría de instrucciones. ■

2.1.1 Unidad de procesamiento con un bus

Veamos una representación de una unidad de procesamiento simple en la que los registros y la ALU se interconectan mediante un único bus común. No deberemos confundir este bus con el que conecta el procesador con la memoria y los dispositivos de E/S, ya que es un bus interno del procesador.



Como podemos ver, los elementos se interconectan mediante el bus común. El *Memory Data Register* (MDR) tiene dos entradas y dos salidas, esto es, los datos pueden ser cargados en MDR desde el bus de memoria o desde el bus interno del procesador, los datos guardados en MDR pueden ser colocados en ambos buses. El *Memory Address Register* (MAR) está conectado para la entrada al bus interno y su salida al bus externo. El uso y número de registros varían según el procesador. Los registros Y, Z y TEMP son registros transparentes

para el programador, en el sentido de que no son referenciados explícitamente mediante ninguna instrucción, son usados por el procesador para el almacenamiento temporal de datos. El multiplexor MUX selecciona la entrada A de la ALU, la constante 4 se usa para incrementar el PC. La UC genera señales internas y externas.

Salvo algunas pocas excepciones, una instrucción se puede ejecutar mediante una o más de las operaciones siguientes:

- Transferencia de una palabra desde un registro de procesador a otro o a la ALU.
- Realizar una operación aritmético-lógica y almacenar el resultado en un registro.
- Cargar los contenidos de una ubicación de memoria y cargarlos en un registro del procesador.
- Almacenar una palabra de datos desde un registro de un procesador a una ubicación de memoria.

Para transferir datos de un registro a otro, cada registro usa dos señales de control

- **Load:** Para cargar datos en paralelo
- **Enable:** Habilitación de salida (tri-state buffer)

Recordamos que un buffer tri-estado es un dispositivo que puede ser activado o desactivado, para permitir o no el paso, y además tiene otro estado (alta impedancia) que lo desconecta del buffer, permitiendo que otros dispositivos conectados al bus puedan utilizarlo sin interferencias.

Todas las operaciones en el procesador ocurren en periodos de tiempo definidos por el reloj del procesador. Para realizar operaciones aritmético-lógicas se usa la ALU, que es un circuito combinacional sin memoria interna, se encarga de realizar operaciones sobre los dos operandos proporcionados a sus entradas A y B. El resultado se almacena temporalmente en el registro Z.

Ejemplo 2.2: Por ejemplo, para sumar los contenidos de $R1$ a $R2$ y guardar el resultado en $R3$ se sigue la secuencia de operaciones:

1. Enable $R1$, Load Y
2. Enable $R2$, SELECT Y , Add, Load Z
3. Enable Z , Load $R3$

■

Las señales cuyos nombres se dan en cada paso son activadas durante la duración del ciclo de reloj correspondiente a ese paso, todas las demás señales están desconectadas. Por tanto, en el paso 1, el output de $R1$ y el input de Y se activan, provocando que los contenidos de $R1$ se carguen mediante el bus a Y . En el paso 2, la señal de SELECT del multiplexor se activa, provocando que los contenidos de Y se carguen en A de la ALU, al mismo tiempo, los contenidos de $R2$ son cargados en el bus y por tanto, en B. La operación realizada por la ALU depende de la señal que se le pasa a sus líneas de control. En este caso, la línea de Add se pone a 1, por lo que la salida de la ALU es la suma de A y B, que se carga en Z. En el paso 3, los contenidos de Z se transfieren al registro de destino $R3$. Notemos que esta última transferencia no se puede llevar a cabo durante el paso 2 ya que solo se puede conectar un registro de salida al bus en cada ciclo de reloj.

Para simplificar la explicación, hemos asumido que la ALU tiene una señal dedicada para cada operación aritmético-lógica, sin embargo, en la práctica, es probable que se encuentre algún tipo de codificación, por ejemplo, para realizar 8 operaciones distintas, bastan 3 señales de control.

Cargar palabras de memoria

Para cargar una palabra de memoria, el procesador tiene que especificar la dirección donde se encuentra la información y solicitar una operación de lectura (Read). Para ello, el procesador transfiere la dirección al MAR, cuya salida se conecta al bus de direcciones, al mismo tiempo, el procesador usa las líneas de control del bus de memoria para indicar que necesita realizar una operación de lectura, el dato leído se almacena entonces en el MDR (que contiene dos entradas y dos salidas).

Durante las operaciones de lectura y escritura, la temporización interna de las operaciones del procesador debe ser coordinada con la de memoria, el procesador completa una transferencia interna en un ciclo de reloj, sin embargo, la velocidad de operación para la lectura puede requerir varios ciclos de reloj, es por ello, que el procesador debe esperar la activación de la señal de finalización de ciclo de memoria.

Ejemplo 2.3: Veamos el proceso para $\text{Load}(R1) \rightarrow R2$:

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

■

Por simplicidad, asumimos que el output de MAR esta activado todo el tiempo, es por ello que los contenidos de MAR están siempre visibles en las líneas de dirección del bus de memoria. Este es el caso cuando el procesador es el maestro del bus, cuando una nueva dirección es cargada en el MAR, aparecerá en el bus de memoria en el comienzo del siguiente ciclo de reloj, una señal de lectura es activada al mismo tiempo que MAR es cargado, esta señal causará que se mande un comando de lectura, MR, en el bus.

Ejecución instrucción completa

Veamos ahora la secuencia de instrucciones elementales que hacen falta para ejecutar una instrucción.

Ejemplo 2.4: Consideramos la instrucción $\text{Add}(R3), R1$. Para ejecuta esta instrucción se siguen los siguientes pasos:

1. Enable PC, Load MAR, Select 4, Sumar, Enable Z
2. Comenzar lectura, Enable Z, Load PC
3. Esperar fin de ciclo de memoria, Load MDR desde memoria
4. Enable MDR hacia bus interno, Load IR

Hasta aquí se estan ejecutando aquellas instrucciones para la captación, para la ejecución:

1. Decodificar instrucción
2. Enable $R3$, Load MAR
3. Comenzar lectura, Enable $R1$, Load Y
4. Esperar fin de ciclo de memoria, Load MDR desde memoria
5. Enable MDR hacia bus interno, Select Y, Sumar, Load Z
6. Enable Z, Load $R1$, Saltar a captación

■

Ejecución de una instrucción de salto

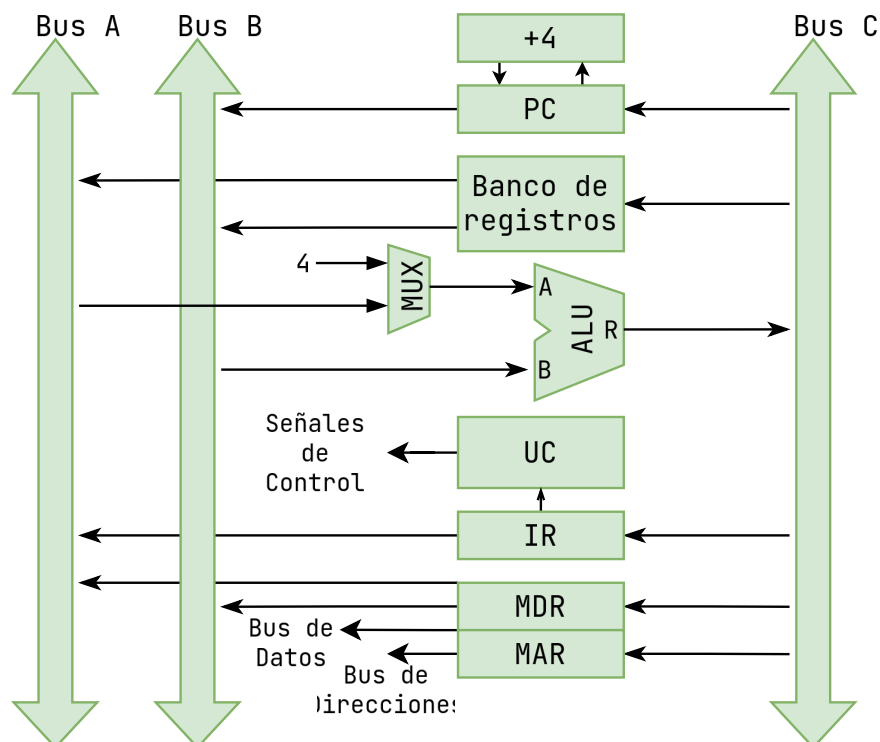
Si queremos ejecutar una instrucción de salto cualquiera (jmp [Address]) se realizan los pasos siguientes:

1. Enable PC, Load MAR, Select 4, Sumar, Enable Z
2. Comenzar lectura, Enable Z, Load PC, Load Y
3. Esperar fin de ciclo de memoria, Load MDR desde memoria
4. Enable MDR hacia bus interno, Load IR
5. Decodificar instrucción
6. Enable Campo desplazamiento en IR, Sumar, Load Z
7. Enable Z, Load PC, Saltar a captación

Notemos que, en caso de que haya que chequear el estatus de los códigos de condición antes de saltar, la instrucción 6 sería reemplazada añadiendo un if $N = 0$ the End indicando que el procesador volvería al paso 1.

2.1.2 Unidad de procesamiento múltiples buses

Hemos usado la estructura con un único bus hasta ahora para ilustrar las ideas básicas, como resultado, las secuencias de pasos que hemos descrito son largas ya que una única unidad de datos puede ser transferida en cada ciclo de reloj, para reducir el número de pasos, la mayoría de procesadores actuales implementan múltiples caminos internos que permiten transferencias en paralelo.



Donde podemos observar una estructura de tres buses usada para conectar los registros y la ALU con el procesador. Todos los registros de propósito general se combinan en un único bloque denominado banco de registros.

El banco de registros mostrado tiene tres puertos: dos de salida y uno de entrada. Esto permite acceder simultáneamente al contenido de dos registros diferentes, cuyos datos se colocan en los buses A y B. Al mismo tiempo, el tercer puerto permite cargar datos desde el bus C a un tercer registro durante el mismo ciclo de reloj.

Los buses A y B transportan los operandos fuente a las entradas de la ALU, donde se realiza una operación aritmética o lógica. El resultado se transfiere al destino a través del bus C. En caso necesario, la ALU puede pasar directamente uno de los operandos de entrada al bus C, mediante las operaciones de control $R=A$ o $R=B$. Esta disposición elimina la necesidad de los registros intermedios Y y Z presentes en diseños más simples.

Además, la figura introduce una unidad de *incrementador*, que se utiliza para incrementar el contador de programa (PC) en 4. Esto elimina la necesidad de usar la ALU principal para dicha operación, como se hacía en estructuras más simples. Sin embargo, la constante 4 en el multiplexor de entrada de la ALU sigue siendo útil, permitiendo incrementar otras direcciones, como en las instrucciones LoadMultiple y StoreMultiple.

Ejecución de una instrucción completa

A continuación, se describe el flujo de ejecución de una instrucción representativa, como por ejemplo: $R6 = R4 + R5$.

1. Enable PC, $R=B$, Load MAR.
2. Comenzar lectura e incrementar PC.
3. Esperar el fin del ciclo de memoria, Load MDR desde la memoria.
4. Enable MDR hacia B, $R=B$, Load IR.
5. Decodificar instrucción.
6. Enable R4 hacia A, Enable R5 hacia B, seleccionar A, sumar, Load R6, Saltar a captación.

De esta manera, la estructura de múltiples buses permite ejecutar instrucciones de manera más eficiente al reducir el número de ciclos necesarios y facilitar operaciones paralelas.

2.2 Unidades de control cableadas y microprogramadas

Para ejecutar instrucciones, el procesador debe tener algún mecanismo para poder generar las señales de control necesarias, para solventar este problema existen varias soluciones, en general podemos distinguir dos categorías:

- Control cableado: (hardwired control) Se emplean métodos de diseño de circuitos digitales secuenciales a partir de diagramas de estado. El circuito final se obtiene conectando componentes básicos como puertas y biestables, aunque más a menudo se usan PLA (P)rogrammable (L)ogic (A)rrays.
- 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. Con este enfoque se logra mayor facilidad de diseño para instrucciones complejas por lo que es el método estándar en la mayoría de los CISC.

2.2.1 Diseño UC cableada

Se diseña mediante puertas lógicas y biestables siguiendo uno de los métodos clásicos de diseño de sistemas digitales secuenciales que ya se vieron en TOC.

- El diseño es laborioso y difícil de modificar debido a la complejidad de los circuitos

- Suele ser más rápida que una UC microprogramada
- Se utilizan PLA para llevar a cabo la implementación

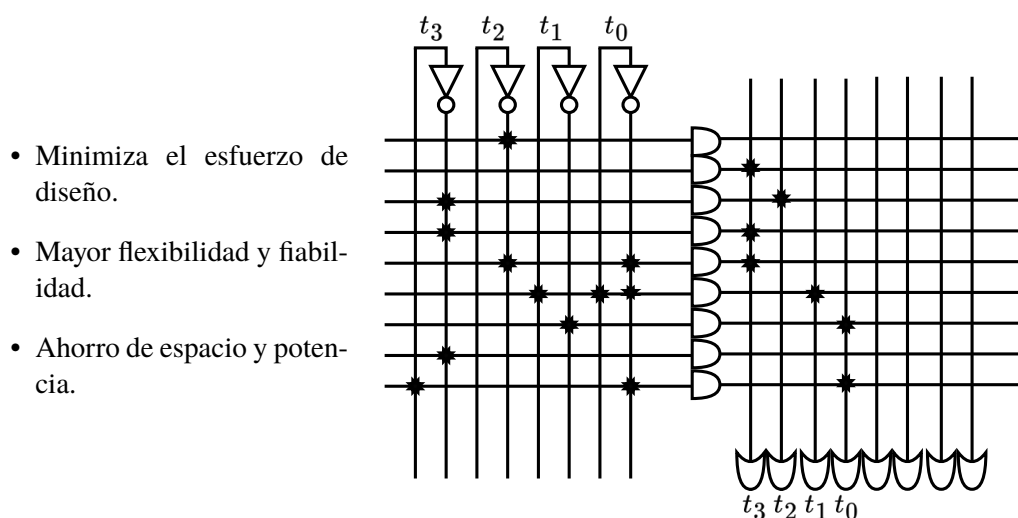
Con las nuevas técnicas de diseño y la tendencia a usar RISC ha tomado nuevo auge este planteamiento.

Técnicas de diseño por computador (CAD) para circuitos VLSI (compiladores de silicio)

Resuelven automáticamente la mayor parte de las dificultades de diseño de lógica cableada. Generan directamente las máscaras de fabricación de circuitos VLSI a partir de descriptores del comportamiento funcional del circuito en un lenguaje de alto nivel.

Organización de UC basada en PLA

Los biestables de la PLA contienen la información relativa al estado en que se encuentra el sistema. La PLA utiliza esta información de estado, junto con las entradas externas para generar el siguiente estado.



- Minimiza el esfuerzo de diseño.
- Mayor flexibilidad y fiabilidad.
- Ahorro de espacio y potencia.

Ejemplo 2.5: Veamos una implementación de una unidad de control cableada sencilla (ODE), para ello, seguimos los siguientes pasos:

1. **Definir una máquina de estados finitos:** Dado el diagrama de flujo de la UC de ODE, detallamos este como un conjunto de estados finitos 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 (como un *autómata (MC)*)
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 (salidas dependen de entradas y estado presente) o un modelo Moore (salidas dependen exclusivamente del estado actual).
4. **Minimizar la tabla de verdad:** Mediante un programa que utilice algoritmos heurísticos rápidos.
5. **Diseñar físicamente la PLA partiendo de la tabla de verdad:** Se puede hacer automáticamente (Mediante un programa especial para diseño de layouts de PLA) o semiautomáticamente (Diseñando un programa de CAD de circuitos VLSI cada una de las celdas que, repetidas convenientemente, forman un PLA o dando una especificación de cómo han de colocarse (tabla de verdad minimizada)).



2.2.2 Unidad de control microprogramada

La idea básica es emplear una memoria (de control) para almacenar las señales de control de los períodos de cada instrucción. Comenzamos introduciendo algunos términos comunes en el diseño de UC microprogramadas:

- **Palabra de control:** Es una palabra cuyos bits individuales reepresentan las señales de control que deben activarse en un ciclo de reloj.
- **Microinstrucción:** Es una palabra de control que se almacena en la memoria de control.
- **Microprograma:** Es una secuencia de microinstrucciones que se ejecutan para llevar a cabo una (macro) instrucción de lenguaje máquina.
- **Ejecución de un microprograma:** Es la 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:** Es el conjunto de los microprogramas de una máquina.

Ventajas de la microprogramación

- 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 la alternativa cableada. No hay que rediseñar el hardware, únicamente 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).

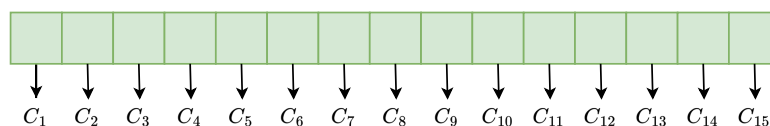
Como ya dijimos, este enfoque tiene como contrapartida una mayor lentitud frente a la cableada, debido a una menor capacidad de expresar paralelismo de las microinstrucciones.

2.3 Control microprogramado

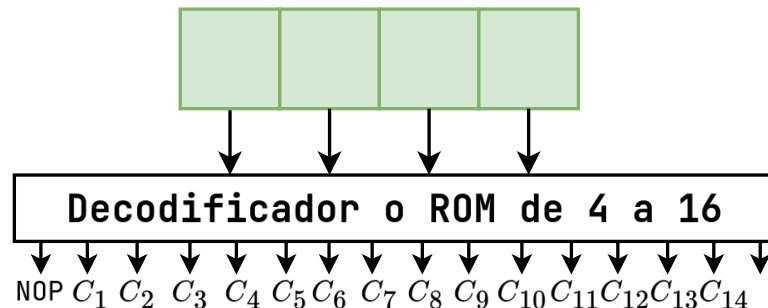
2.3.1 Formato de las microinstrucciones

Las señales de control que gobiernan un mismo elemento del *datapath* se suelen agrupar en campos, por ejemplo: las señales triestado que controlan un bus, las señales de operación de la ALU, memoria, etc. Existen dos enfoques:

- **Formato no codificado:** Hay un bit para cada señal de control de un campo.



- **Formato codificado:** Con el objetivo de acortar el tamaño de las microinstrucciones, se codifican todos o algunos de sus campos. Esto, sin embargo, implica la necesidad de incluir decodificadores para extraer la información real.



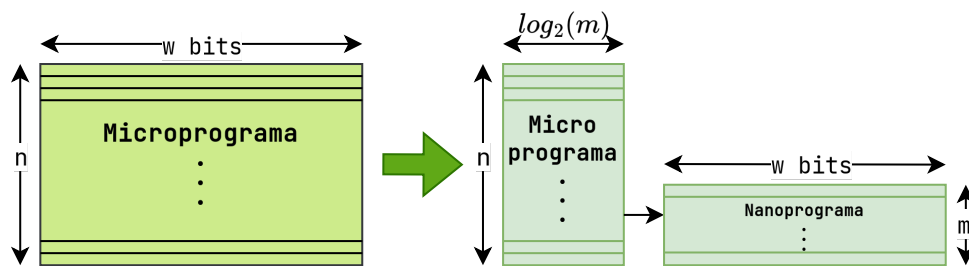
Cuando pocas señales de control están activas en cada ciclo, o existen con frecuencia señales mutuamente excluyentes, es posible acortar la longitud de las microinstrucciones solapando campos. Sin embargo, esto introduce un retardo debido al uso de un demultiplexor y hace incompatibles las operaciones con campos solapados. El nivel de abstracción de un microprograma depende de la cantidad de codificación de las señales de control y del paralelismo presente en el formato de las microinstrucciones. Distinguimos dos estilos principales:

- **Microprogramación vertical:**
 - Está altamente codificada y puede parecerse a una macroinstrucción simple, que incluye un único campo de control y uno o dos campos para especificar operandos.
 - Cada microinstrucción especifica una única operación sobre el *datapath* y, al decodificarse, activa múltiples señales de control.
 - Las bifurcaciones se manejan mediante microinstrucciones separadas que contienen códigos de salto.
- **Microprogramación horizontal:**
 - Puede estar completamente sin codificar, asignando un bit separado a cada señal de control.
 - Habitualmente utiliza varios campos de control, cada uno codificando opciones mutuamente excluyentes.
 - Las bifurcaciones son más complejas, ya que cada microinstrucción puede tener condiciones de salto y direcciones explícitas.

Aunque es menos familiar para los programadores tradicionales, permite explotar el paralelismo, lo que la hace eficiente y similar a la programación en procesadores VLIW.

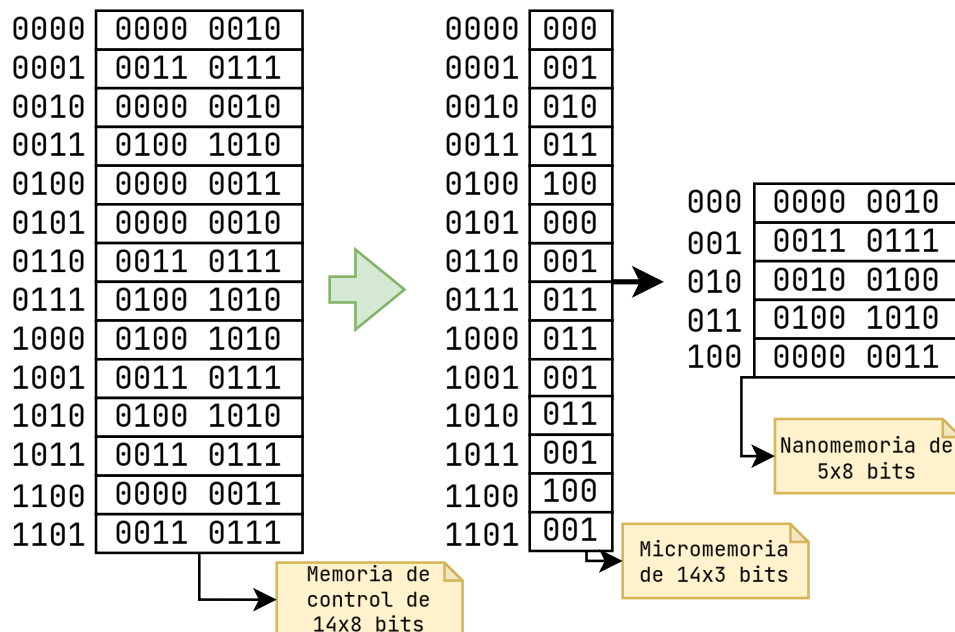
2.3.2 Nanoprogramación

La combinación de microinstrucciones verticales y horizontales en un esquema de dos niveles se denomina **nanoprogramación**. Su objetivo principal es reducir el tamaño de la memoria de control.



A la izquierda, el microprograma original contiene n instrucciones de w bits, ocupando un total de $n \cdot w$ bits. Si de estas n instrucciones, m son únicas, se reemplaza cada microinstrucción por su dirección en la nanomemoria (tamaño: $n \cdot \log_2 m$). Esta se conecta a un nanoprograma con las m instrucciones únicas ($m \cdot w$ bits), logrando un ahorro de memoria de $n \cdot w - n \cdot \log_2 m + m \cdot w$ bits.

Ejemplo 2.6: Consideremos una memoria de control de 14×8 bits con 5 microinstrucciones únicas:



El ahorro de memoria sería:

$$112 - (14 \cdot 3) - (5 \cdot 8) = 30 \text{ bits, es decir, un } 27\%.$$

Considerando un caso real, como la unidad de control del Motorola 68000, con 640 microinstrucciones (280 únicas) y longitud de instrucción de 70 bits:

$$\lceil \log_2(280) \rceil = \lceil 8.129 \rceil = 9.$$

El ahorro total sería:

$$44800 - 25360 = 19440 \text{ bits, es decir, un } 43\%.$$

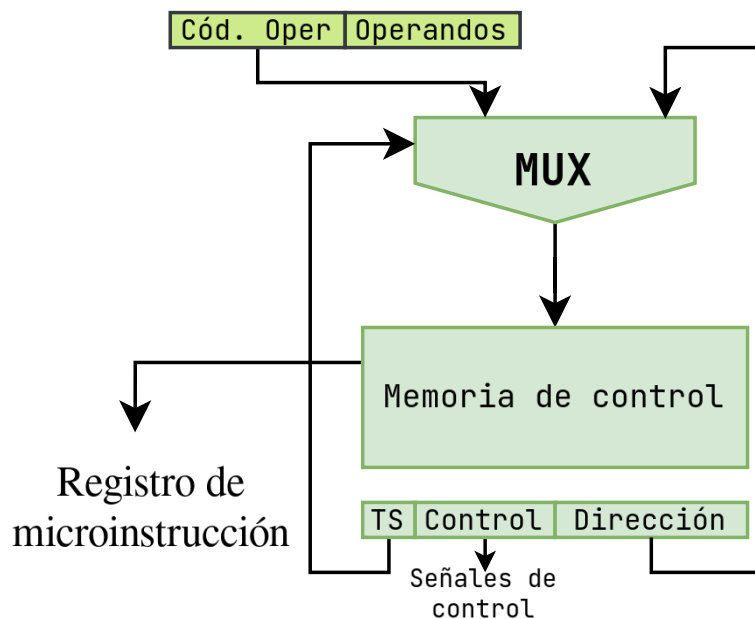
2.3.3 Secuenciamiento de microinstrucciones

Si cada instrucción de lenguaje máquina se traduce en un microprograma, la secuencia de microinstrucciones que lo componen se puede organizar de dos maneras:

Secuenciamiento de microinstrucciones explícito:

En este método, cada microinstrucción contiene información explícita sobre la dirección de la siguiente microinstrucción a ejecutar. Esto se logra típicamente incluyendo un campo de "dirección siguiente" dentro de cada microinstrucción.

- **Funcionamiento:** La unidad de control lee la microinstrucción actual, ejecuta las operaciones especificadas por sus campos de control y luego utiliza el campo de "dirección siguiente" para determinar la dirección de la siguiente microinstrucción que debe leer de la memoria de control.
- **Ventajas:**
 - **Flexibilidad:** Permite un control muy preciso sobre el flujo del microprograma, incluyendo saltos condicionales y bucles complejos. Las microinstrucciones no necesitan estar almacenadas en ubicaciones consecutivas de la memoria.
 - **Implementación de bifurcaciones complejas:** Facilita la implementación de bifurcaciones condicionales basadas en múltiples condiciones o en valores almacenados en registros.
- **Desventajas:**
 - **Mayor tamaño de microinstrucción:** Requiere más bits por microinstrucción para almacenar la dirección siguiente, lo que aumenta el tamaño de la memoria de control.
 - **Complejidad del hardware de control:** El hardware de control debe ser capaz de extraer y utilizar la información de la dirección siguiente de cada microinstrucción.

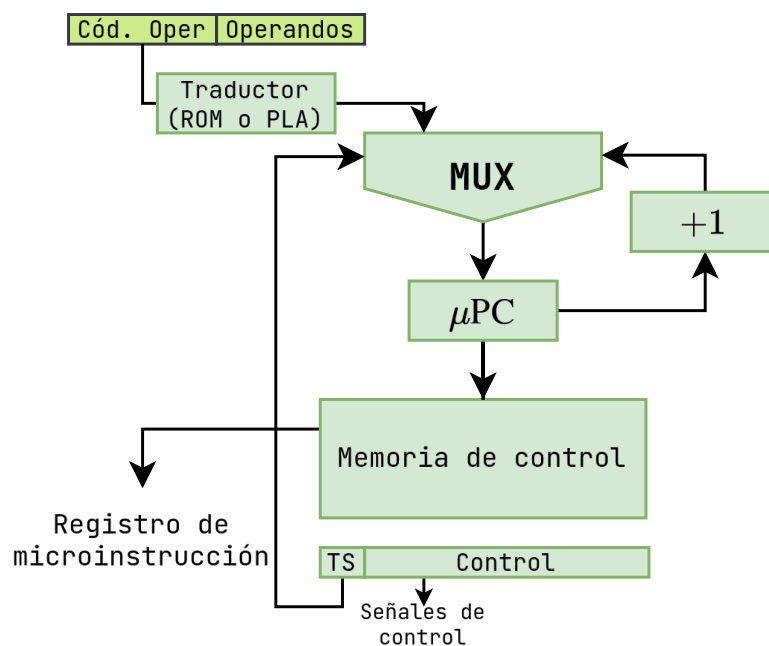


Secuenciamiento de microinstrucciones implícito:

En este enfoque, la dirección de la siguiente microinstrucción se determina de forma implícita, generalmente incrementando un contador de microprograma (μ PC). Las microinstrucciones se almacenan en ubicaciones consecutivas de la memoria de control.

- **Funcionamiento:** Después de ejecutar una microinstrucción, el μ PC se incrementa automáticamente para apuntar a la siguiente ubicación en la memoria de control. Este método asume que la secuencia de microinstrucciones es lineal y consecutiva.

- **Manejo de bifurcaciones:** Para implementar bifurcaciones (saltos), se utilizan microinstrucciones especiales de "salto" que modifican el valor del μ PC con una nueva dirección. A menudo se combina con un multiplexor para seleccionar la siguiente dirección entre el μ PC incrementado y la dirección de salto.
- **Ventajas:**
 - **Menor tamaño de microinstrucción:** No necesita un campo explícito para la dirección siguiente, lo que reduce el tamaño de la memoria de control.
 - **Hardware de control más simple:** El hardware necesario para el secuenciamiento es más sencillo, ya que principalmente implica un contador y un multiplexor.
- **Desventajas:**
 - **Menor flexibilidad:** Las secuencias de microinstrucciones deben ser principalmente lineales. Las bifurcaciones complejas requieren más microinstrucciones y lógica de control.
 - **Limitaciones en la organización de la memoria de control:** Las microinstrucciones pertenecientes a un mismo microprograma deben almacenarse en posiciones consecutivas, lo que puede dificultar la optimización del uso de la memoria.



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.

Microbucles

Se da cuando una instrucción de lenguaje máquina contiene operaciones repetidas, e.g (desplazamiento múltiple, multiplicación, división, cadenas), es decir, se tiene la necesidad de tener microbucles para ejecutar de forma

eficiente este tipo de operaciones. Para implementar un microbucle, se puede utilizar la bifurcación condicional más un contador con autodecremento, cuando el contador llega a 0, se bifurca, este contador debe poder inicializarse desde una microinstrucción.

Microsubrutinas

Las instrucciones máquina tienen con frecuencia partes de su ejecución en común, es decir, se puede evitar la redundancia de código mediante la implementación de microsubrutinas, la llamada a microsubrutinas sin embargo exige un almacenamiento para guardar la dirección de retorno, la solución habitual es usar una pila de direcciones de retorno.

2.3.4 Control residual

Hasta ahora, todas las señales de control necesarias para manipular la microarquitectura estaban codificadas en campos de la microinstrucción actual, a esto se le llama **Control inmediato**. Sin embargo, en ciertos casos puede ser útil que una microinstrucción pueda almacenar señales de control en un registro, conocido como **registro de control residual**, para usarlas en ciclos posteriores. Esto se llama **Control residual**.

El objetivo principal del control residual es optimizar el tamaño del microprograma al reducir la redundancia de información de control que, de otra forma, debería codificarse en múltiples microinstrucciones sucesivas. Sus usos más comunes son:

- En microsubrutinas o conjuntos compartidos de microinstrucciones.
- En situaciones donde parte de la información de control permanece invariable durante muchas microinstrucciones.

Ejemplo 2.7: Veamos un ejemplo de control residual en microsubrutinas o conjuntos compartidos de microinstrucciones. Supongamos un procesador con dos instrucciones máquina para realizar las operaciones AND y OR entre dos cadenas de bytes. En lugar de tener microinstrucciones específicas para cada una de estas operaciones, se puede usar una microsubrutina o un conjunto común de microinstrucciones que aproveche un registro de control residual para almacenar las señales de control de la operación deseada.

En este caso, la ALU está controlada por 3 bits, con la siguiente codificación:

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

Para las instrucciones AND y OR, utilizaremos el valor 111 en la microinstrucción común para especificar que las señales de control de la ALU no proceden del campo de microinstrucción, sino del registro de control residual. Esto permite reutilizar el microcódigo común para ambas operaciones. ■

Carga del registro de control residual. Para cargar información en el registro de control residual, se requiere un mecanismo específico. Este puede ser una microinstrucción especial (como la microinstrucción de llamada a microsubrutina) que contenga:

- Un campo que indique el valor a almacenar en el registro.
- Un bit que especifique que se desea cargar información en el registro de control residual.

Una vez cargado, el registro de control residual puede ser utilizado por microinstrucciones comunes para especificar señales de control necesarias en operaciones posteriores. Esto reduce significativamente el número de microinstrucciones necesarias y, en algunos casos, la anchura de estas.

Generalización del concepto. El control residual no se limita a operaciones en la ALU. Su concepto puede aplicarse a otros elementos de control en el procesador, como:

- Selección de registros destino.
- Control de operaciones de memoria.
- Determinación de direcciones o condiciones de salto.

Además, cuando parte de la información de control permanece invariable durante un período prolongado, el registro de control residual también puede denominarse **registro de configuración** (*setup register*). Este enfoque permite reducir tanto el número de microinstrucciones como su anchura, optimizando el diseño y la implementación del microprograma. cargar información en el registro de control residual, por ejemplo, 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.

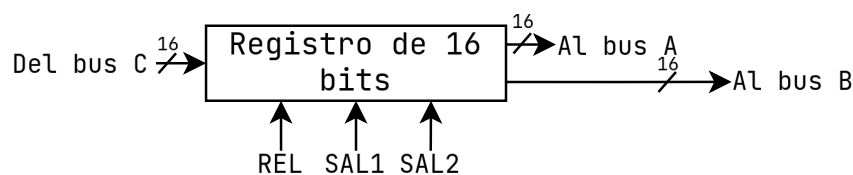
2.3.5 Camino de datos

Temporización

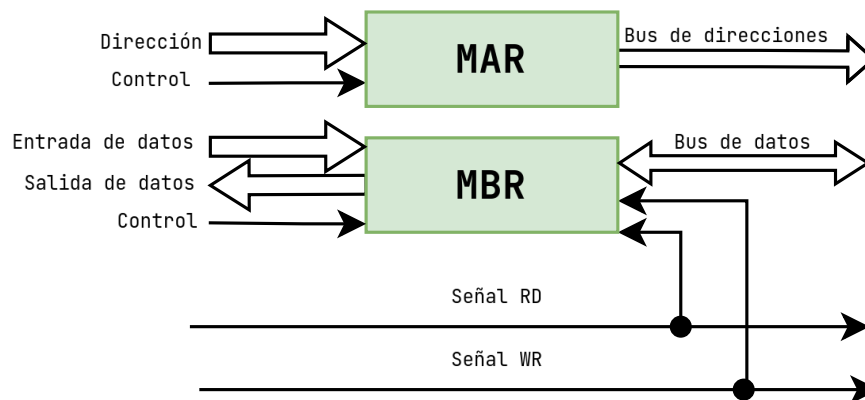
Un ciclo básico de la UC consiste en una secuencia de 4 subciclos controlada por un reloj de 4 fases:

- 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 de 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.
- La salida del desplazador está estabilizada. Se almacena el bus *C* en un registro y en MBR si es necesario.

Veamos uno de los registros de 16 bits con más detalle



Donde en REL se carga del bus C, y en SAL 1 y 2 se tiene la salida a los buses A y B. Veamos ahora las señales y buses de MAR y MBR con más detalle:



Donde cada entrada de control supone la escritura sobre MAR o MBR, la señal RD se encarga de la lectura de la memoria y escritura en MBR del dato leído, en cuanto a WR, se encarga de la lectura de MBR y escritura en memoria del dato leído, ambas van a la memoria.

2.3.6 Diseño Horizontal

En el caso del diseño horizontal de la unidad de control, se necesitan las siguientes señales de control:

- 16 señales para cargar el bus A desde registros.
- 16 señales para cargar el bus B desde registros.
- 16 señales para cargar registros desde el bus C.
- 2 señales para controlar los buffers A y B.
- 2 señales para controlar la función de la ALU.
- 2 señales para controlar el desplazador.
- 4 señales para controlar MAR y MBR.
- 2 señales para lectura/escritura en memoria.
- 1 señal para controlar el multiplexor MUXA.

Es decir, un total de 61 señales.

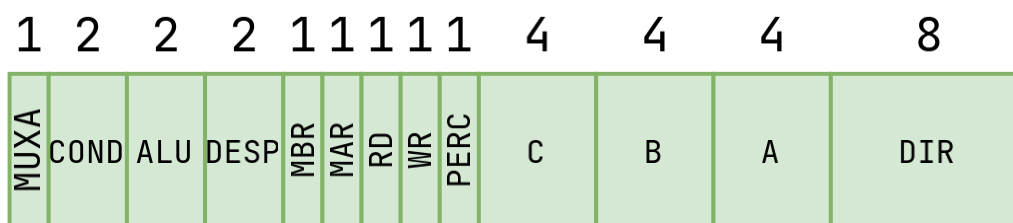
Optimización de Señales

Para reducir las señales necesarias, se pueden aplicar las siguientes técnicas:

- Codificación de señales de registros: Reducción de 48 a 12 bits.
- Eliminación de los 2 bits para carga de los buffers A y B: Como siempre se cargan en el mismo momento del ciclo máquina, se pueden activar en el segundo subciclo de reloj, reducción de 2 bits a 0.
- Simplificación de MAR y MBR: Reduciendo el número de señales que controlan MAR y MBR (LEC puede usarse para cargar MBR desde la memoria y ESC para darle salida). Reducción de 4 bits a 2.

Sin embargo es necesario añadir otras señales, por ejemplo nos puede interesar generar N y Z sin almacenar el resultado, o almacenarlo solo 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). Con estas modificaciones, nos quedan 22 bits de control. De esta forma, el formato de una microinstrucción queda de la forma siguiente (32 bits):

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



Cada uno de los campos anteriores tiene la siguiente codificación:

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

En cuanto a la unidad de control:

- **Memoria de control:** 256 palabras de 32 bits = 8192 bits.
- La unidad de incremento calcula $\text{microPC} + 1$
- 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 microIR.
 - 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 sub-ciclo 4 (cuando N y Z son válidos), a partir de N, Z y los dos bits COND (I = Izdo., D = dcho.):

$$\text{MUXM} = \bar{I}DN + I\bar{D}Z + ID = DN + IZ + ID$$

La arquitectura de la unidad de control es la siguiente:

- Memoria principal: 4096 palabras de 16 bits.
- 3 registros visibles por el programador de lenguaje máquina:
 - PC: contador de programa.
 - SP: puntero de pila.
 - AC: acumulador.

- 3 modos de direccionamiento:
 - Directo: los 12 bits menos significativos son una dirección de memoria.
 - Indirecto: AC contiene una dirección de memoria.
 - Local: los 12 bits menos significativos son un desplazamiento que se suma al puntero de pila.
- La pila crece hacia direcciones de memoria menores.
- La E/S es mapeada en memoria, por lo que no hay instrucciones de E/S específicas.
- El registro AMASK (0FFF) se utiliza para obtener el campo dirección en las instrucciones de direccionamiento directo y local.
- El registro SMASK (00FF) se utiliza para obtener el incremento/decremento del puntero de pila en las instrucciones INSP y DESP.

Veamos un repertorio de 23 instrucciones máquina:

Binario	Alias	Instrucción	Significado
0001xxxxxxxxxxxx	LODD	Carga directa	$ac := m[x]$
0010xxxxxxxxxxxx	STOD	Almacenamiento directo	$m[x] := ac$
000100xxxxxxxxxx	ADDD	Suma directa	$ac := ac + m[x]$
000101xxxxxxxxxx	SUBD	Resta directa	$ac := ac - m[x]$
0100xxxxxxxxxxxx	JPOS	Salto si positivo	$if\ ac \geq 0\ then\ pc := x$
0101xxxxxxxxxxxx	JZER	Salto si cero	$if\ ac = 0\ then\ pc := x$
0110xxxxxxxxxxxx	JUMP	Salto incondicional	$pc := x$
0111xxxxxxxxxxxx	LOCO	Carga de constante	$ac := x\ (0 \leq x \leq 4095)$
1000xxxxxxxxxxxx	LODL	Carga local	$ac := m[sp + x]$
1001xxxxxxxxxxxx	STOL	Almacenamiento local	$m[sp + x] := ac$
1010xxxxxxxxxxxx	ADDL	Suma local	$ac := ac + m[sp + x]$
1011xxxxxxxxxxxx	SUBL	Resta local	$ac := ac - m[sp + x]$
1100xxxxxxxxxxxx	JNEG	Salto si negativo	$if\ ac < 0\ then\ pc := x$
1101xxxxxxxxxxxx	JNZE	Salto si no cero	$if\ ac \neq 0\ then\ pc := x$
1110xxxxxxxxxxxx	CALL	Llamada a subrutina	$sp := sp - 1; m[sp] := pc; pc := x$
1111000000000000	PSHI	Apilamiento indirecto	$sp := sp - 1; m[sp] := m[ac]$
1111000100000000	POPI	Desapilamiento indirecto	$m[ac] := m[sp]; sp := sp + 1$
1111001000000000	PUSH	Apilamiento	$sp := sp - 1; m[sp] := ac$
1111001100000000	POP	Desapilamiento	$ac := m[sp]; sp := sp + 1$
1111010000000000	RETN	Retorno de subrutina	$pc := m[sp]; sp := sp + 1$
1111010100000000	SWAP	Intercambio de AC y SP	$tmp := ac; ac := sp; sp := tmp$
11111110yyyyyyyy	INSP	Incremento de SP	$sp := sp + y\ (0 \leq y \leq 255)$
11111110yyyyyyyy	DESP	Decremento de SP	$sp := sp - y\ (0 \leq y \leq 255)$

Lenguaje microprogramar en alto nivel

Los microprogramas se pueden escribir:

- En binario: 32 bits por microinstrucción.

- Nombrando cada campo distinto de 0 y su valor (1 microinstrucción por línea).
- Con instrucciones de alto nivel tipo PASCAL.

Ejemplo 2.8: Veamos varios ejemplos de microinstrucciones en alto nivel:

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

■

La decodificación de las instrucciones se realiza examinando IR bit a bit, esto conlleva una proporción considerable de tiempo, veamos otro enfoque:

2.3.7 Diseño Vertical

Cada microinstrucción contiene ahora 3 campos de 4 bits:

- OP: Código de operación de la ALU.
- R_1 y R_2 : Dos registros (Dirección en caso de salto)

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

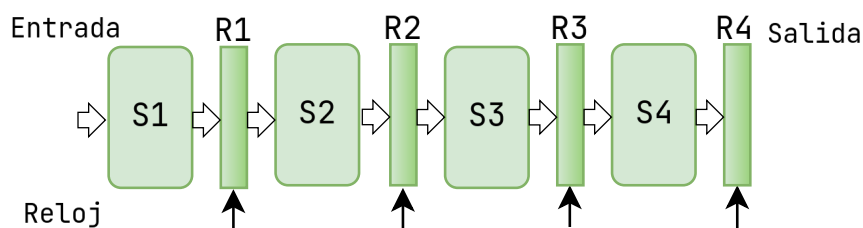
Por ejemplo un programa pudiera tener 160 microinstrucciones de 12 bits, es decir 1920 bits, al tener menos memoria de control, la UC es más lenta.

Segmentación de Cauce

3.1 Concepto de Segmentación

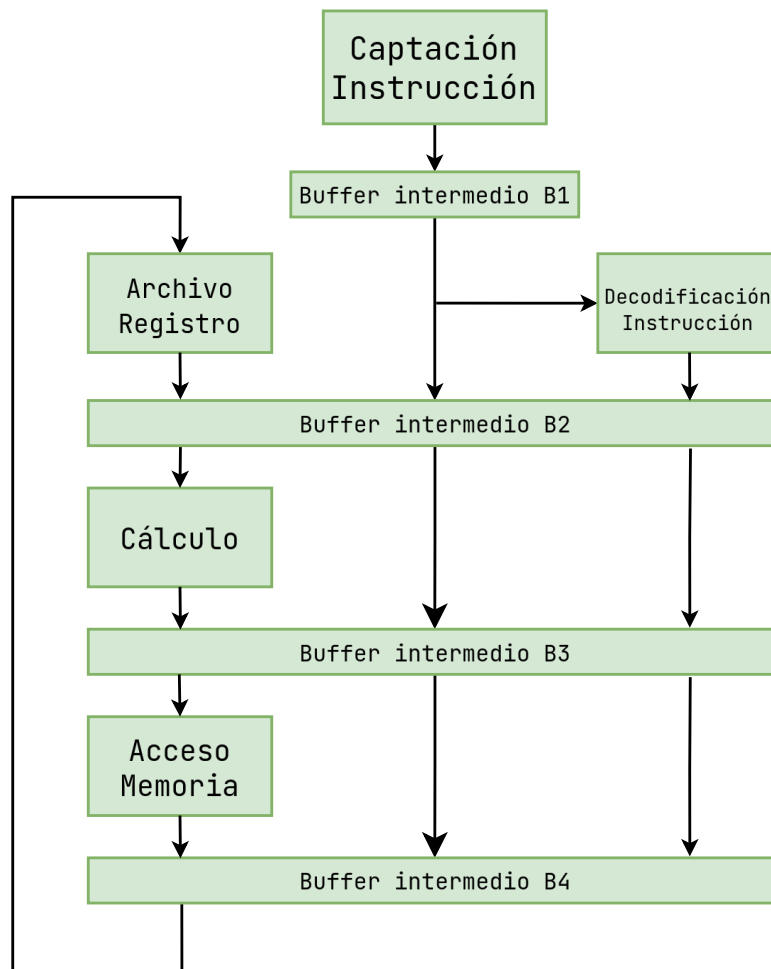
La velocidad de ejecución de un programa depende de varios factores, una manera de incrementar la eficiencia es usar tecnologías de circuitos más rápidos para la memoria principal y el procesador. Otra posibilidad es ajustar el hardware para que más de una operación se pueda ejecutar al mismo tiempo, de esta manera, el número de operaciones por segundo aumenta a pesar de que el tiempo necesario para ejecutar cada operación no se modifica.

Es a esto a lo que se le llama segmentación o *pipelining*, la idea básica es muy simple, por ejemplo es muy común encontrarla en cadenas de montaje de fábricas, en las que cada operario realiza una tarea concreta. En el caso de un procesador, la idea será subdividir el procesador en n etapas, permitiendo el solapamiento en la ejecución de instrucciones:



Donde cada registro entre las etapas se encarga de almacenar la información necesaria para la siguiente etapa. De esta manera, las instrucciones entran por un extremo del cauce son procesadas en distintas etapas y salen por el otro extremo. Cada instrucción individual sigue tardando un tiempo T , pero hay varias instrucciones ejecutándose simultáneamente en distintas etapas.

Ejemplo 3.1: Veamos un ejemplo de una segmentación de 5 etapas: (IF) Captación de instrucción, (ID) Decodificación de instrucción, (EX) Ejecución de instrucción, (MEM) Acceso a memoria y (WB) Escritura de resultados.



Cada etapa del cauce debe completarse en un ciclo de reloj. En las fases de captación y memoria trataremos de tener un acceso a caché, que permitiera acceder en un ciclo de reloj, si hubiera un fallo de caché, el acceso a memoria principal tardará varios ciclos de reloj (Aproximadamente 10). El periodo de reloj se ajusta de acuerdo a la etapa más larga del cauce. ■

3.2 Aceleración

Para medir la aceleración que se consigue mediante la segmentación, podemos usar la simple fórmula:

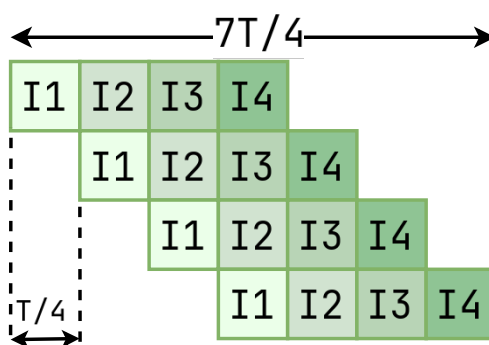
$$\text{Aceleración} = \frac{\text{Tiempo sin segmentación}}{\text{Tiempo con segmentación}}$$

Notemos que en el caso ideal, si tenemos k instrucciones y n etapas, la aceleración ideal viene dada por:

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

Es decir, la aceleración ideal coincide con el número de etapas de segmentación.

Ejemplo 3.2: Veamos un ejemplo con 4 etapas de segmentación y 4 instrucciones:



En este caso, la aceleración es $\frac{4T}{(4-1+4)T/4} = 16/7$ ■

La segmentación aunque es buena, no es perfecta, y hay varios problemas que pueden provocar una disminución en la aceleración:

- Coste de segmentación.
- Duración del ciclo de reloj impuesto por la etapa más larga.
- Riesgos (hazards) que pueden bloquear el avance de instrucciones.

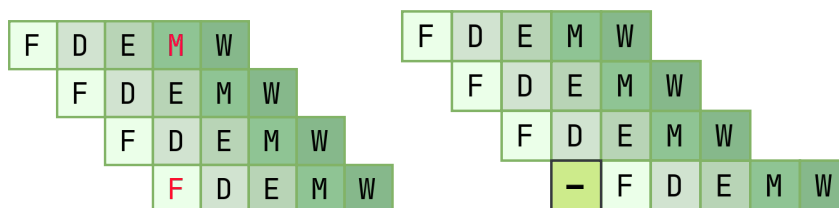
3.3 Riesgos

Cualquier condición que cause una parada en el avance de las instrucciones se llama riesgo, existen varios tipos de riesgos, que como sabemos, obligan a modificar la forma en la que avanzan las instrucciones en el cauce y por tanto, disminuyen la aceleración. Para esta sección, supongamos las etapas F (fetch), D (decode), E (execute), M (memory) y W (write-back) por simplicidad, aunque en los procesadores modernos encontramos unas pocas más (alrededor de 30).

3.3.1 Riesgos estructurales

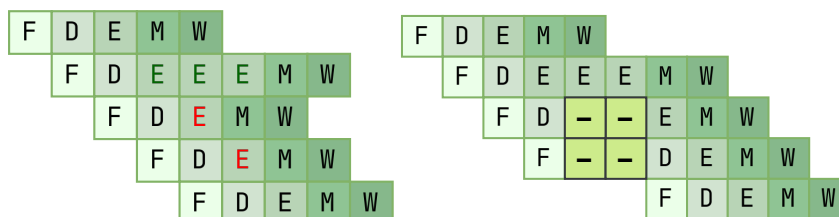
Son aquellas situaciones en las que el hardware no puede proporcionar suficientes recursos para ejecutar varias instrucciones al mismo tiempo. Por ejemplo, si una instrucción necesita acceder a la memoria (etapa M) y otra necesita acceder a memoria para captar la instrucción (etapa F). La solución en este caso es atrasar la instrucción que provoca el riesgo:

```
LW R4, 20(R1)
AND R7, R2, R5
OR R8, R6, R2
ADD R9, R2, R2
```



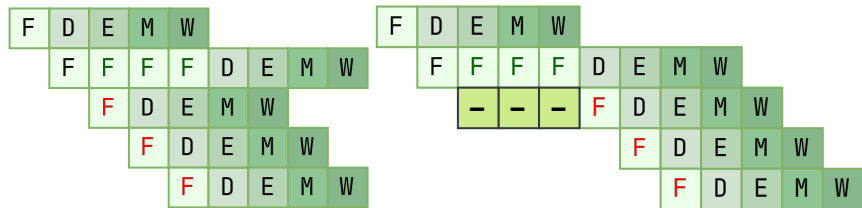
Donde la instrucción ADD se retrasa una etapa para evitar el riesgo estructural entre *M* de lw y *F* de ADD. Otro caso es la ejecución de una operación con más de un ciclo en E.

```
ADD RX, RX, RX
MUL RD, RS, RS
ADD RX, RX, RX
ADD RX, RX, RX
ADD RX, RX, RX
```



En este caso, las instrucciones se tienen que retrasar hasta que termine la ejecución de MUL ya que únicamente se ha supuesto una unidad de ejecución. Por último veamos un ejemplo en el que encontramos un fallo de caché.

```
ADD RX, RX, RX
SUB RX, RX, RX
ADD RX, RX, RX
ADD RX, RX, RX
ADD RX, RX, RX
```



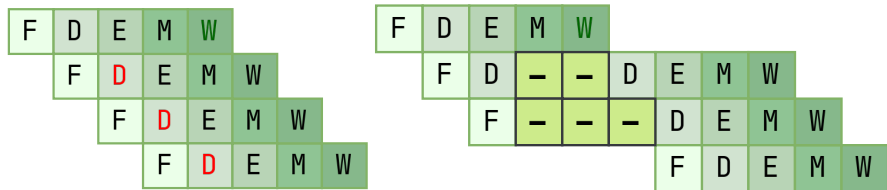
En este caso, lo que pasa es que hay un fallo de caché al buscar la instrucción SUB. Más adelante se profundizará en la caché sin embargo, podemos explicarlo brevemente como que la caché mantiene una copia de las instrucciones siguientes a ejecutar, esto conlleva una velocidad de ejecución en un ciclo sin retardo, sin embargo, si se produce un fallo de caché ya sea por un salto o por otra razón, la siguiente instrucción se tendrá que buscar en memoria principal, lo que aproximadamente conlleva un gran retardo, en este caso por simplicidad se ha representado como tres unidades de tiempo pero realmente podemos esperar alrededor de 10 ciclos de reloj.

Para evitar en la medida posible los fallos de caché, se implementa una cola de instrucciones (precaptación) que almacena las instrucciones siguientes a ejecutar, en el caso de saltos condicionales, se evalúa la probabilidad mayor (se verá en Arquitectura de Computadores), cuando el salto sea directo, no tendremos este problema ya que sabremos con certeza la dirección de salto.

3.3.2 Riesgos por dependencias de datos

Son aquellos riesgos que se producen cuando una instrucción depende de los resultados de otra instrucción anterior.

```
SUB R2, R1, R3
AND R7, R2, R5
OR R8, R6, R2
ADD R9, R2, R2
```



En el anterior caso, hasta que no se haya completado SUB no se puede decodificar ninguna instrucción (si se pudiese, se podría plantear reordenar las operaciones). Sin embargo, si añadimos un bus entre las etapas D y E, podemos adelantar la ejecución ya que tras la etapa E ya estará actualizado el registro R2 y no será necesario esperar hasta W.

Estas dependencias de datos las descubre el hardware al decodificar las instrucciones, alternativamente, el compilador puede introducir instrucciones nop, esto tiene como ventaja que el hardware es más simple y se pueden reordenar las instrucciones para aprovechar todo el tiempo de ejecución, sin embargo, aumenta el tamaño del código.

3.3.3 Riesgos de control

Como hemos adelantado antes, las instrucciones de salto pueden provocar fallos de caché, veamos un ejemplo:

```
BR L1
AND R2, R1, R4
SUB R5, R6, R7
OR R8, R1, R6
L1:
ADD R6, R1, R4
```


Hasta que no se completa la etapa de Memoria (M) de la instrucción de salto, no se conoce la dirección de destino del salto incondicional, lo que provoca una pérdida de tres ciclos, ya que las instrucciones cargadas en ese tiempo deben descartarse. Para mitigar este impacto, es fundamental calcular la dirección de salto lo antes posible, por ejemplo, en la etapa de Decodificación. Esto permite que el contador de programa se actualice antes y se reduzca la pérdida a solo un ciclo de reloj, mejorando el rendimiento del pipeline.

En cuanto a los saltos condicionales, deberemos tratar de predecir el salto, y calcular la condición de salto lo antes posible para disminuir la penalización. De manera similar a como hemos hecho antes, el comparador que evalúa la condición de salto se puede mover a la etapa de decodificación. Para calcular la degradación de las prestaciones se puede usar la siguiente fórmula (suponiendo que tenemos algún mecanismo hardware que permita descartar las instrucciones):

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

Donde:

- b Es el número de ciclos desperdiciados cuando se produce un salto.
- p_b Es la probabilidad de que se ejecute una instrucción de salto.
- p_t Es la probabilidad de que realmente se produzca un salto.
- $p_e = p_b p_t$ Es la probabilidad de que se produzca un salto.
- CPI Es el número medio de ciclos por instrucción (1 si no hay saltos).

Si llamamos ahora F_b a la fracción de máximas prestaciones:

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

Notemos que la degradación crece rápidamente si lo hace p_e

Salto retardado

Considerando un ejemplo simple de salto y suponiendo que podemos determinar la decisión de salto en la etapa de decodificación (al mismo tiempo que la siguiente instrucción es captada), la instrucción de salto puede causar el descarte de la instrucción captada. Esto significa que, o bien se pierde un ciclo de reloj, o bien no hay penalización. En cualquiera de los dos casos, la instrucción siguiente siempre es captada. Veamos una manera de reducir esta penalización:

En lugar de descartar condicionalmente la instrucción, podemos reordenar la segmentación de tal manera que siempre se ejecute una instrucción independientemente del salto. Es importante notar que esta instrucción no puede ser la siguiente instrucción lógica. En su lugar, el compilador busca una instrucción adecuada para rellenar el hueco, una que deba ejecutarse independientemente de si se toma el salto o no. Este enfoque se conoce como *salto retardado*. Por ejemplo:

Antes

```
mov r1, #3
jmp L1
nop
```

Después

```
jmp L1
mov r1, #3
```

De esta manera, se logra reducir o eliminar la penalización asociada al salto, dependiendo de si se encuentra una instrucción adecuada para rellenar el hueco del salto retardado. Esto sería para saltos condicionales, en el caso de saltos incondicionales, podemos colocar la instrucción destino del salto tras la instrucción de salto.

Annulling branch

En este caso se ejecuta la instrucción siguiente al salto si este se produce, en caso contrario, la ignora, con este tipo de salto, el destino de un salto condicional se puede colocar tras el salto:

Antes	Después
<code>bz else</code>	<code>bz,a else # + 4</code>
<code>nop</code>	<code>mov r3, #100</code>
<code>; codigo then</code>	<code>; codigo then</code>
<code>else:</code>	<code>else:</code>
<code>mov r3, #100</code>	<code>mov r3, #100</code>

Predicción de salto

En los anteriores casos, hemos visto que tomando decisiones sobre el salto en el ciclo de decodificación, podemos reducir la penalización asociada a los saltos. Aun así, la instrucción siguiente se capta y puede ser descartada, la decisión de captar la instrucción se toma en el primer ciclo, cuando el PC se incrementa, por tanto, para reducir aún más la penalización, el procesador debe anticipar que la siguiente instrucción es un salto y predecir el resultado del mismo. Para esto, hay varios métodos (nuevamente, se verá en Arquitectura de Computadores), adelantamos dos tipos de predicción:

- Predicción estática: Se asume que los saltos son siempre tomados o no tomados.
- Predicción dinámica: Se basa en la historia de los saltos para predecir el resultado.

Dentro de la predicción dinámica, encontramos varios métodos, uno de ellos es el de dos bits, en el que se mantiene un contador de dos bits, este se incrementa (con un máximo de 3) si el salto se toma, y se decrementa (con un mínimo de 0) si no se toma, la predicción se toma como tomado si el contador es mayor o igual a 2 y no tomado en caso contrario.

3.4 Influencia en el repertorio de instrucciones

Debido a los riesgos anteriores, es deseable que un operando no necesite más de un acceso a memoria, así como que sólo puedan acceder a memoria instrucciones de carga y almacenamiento, comparamos dos modos de direccionamiento:

<code>lw r4, (20(r1))</code>	F D E M M W	<code>lw r3, 20(r1)</code>	F D E M W
<code>and r7, r2, r5</code>	F D E - M W	<code>lw r4, (r3)</code>	F D E M W
		<code>and r7, r2, r5</code>	F D E M W

Que son de idéntica duración, sin embargo, el modo de direccionamiento simple permite un hardware más sencillo. También deberemos tener cuidado con la ejecución de instrucciones antes del salto, ya que será deseable elegir una que no modifique los códigos de condición.

3.5 Funcionamiento superescalar

El máximo rendimiento de un procesador segmentado es un instrucción por ciclo. Otro enfoque más agresivo es equipar al procesador con múltiples unidades de ejecución (superescalar), cada una de las cuales puede estar segmentada, esto incrementa la capacidad del procesador de manejar varias instrucciones en paralelo. De esta forma, varias instrucciones comienzan la ejecución en el mismo ciclo de reloj, pero en distintas unidades de ejecución.

Esto incluye la necesidad de incluir varias unidades funcionales, sin embargo, obtenemos el rendimiento superior a una instrucción por ciclo, para esto es fundamental tener una conexión ancha con caché, una cola de instrucciones, una unidad llamada *dispatch unit* (Se incluye en la etapa D) que se encarga de tomar dos o mas instrucciones del frente de la cola, decodificarlas, y enviarlas a las unidades de ejecución correspondientes. Esto también supone que el efecto negativo de los riesgos es más pronunciado, sin embargo, el compilador puede reordenar las instrucciones para evitar estos riesgos, además sucede que las instrucciones pueden emitirse en orden y finalizar de forma desordenada, esto conlleva múltiples problemas cuyas soluciones se verán en Arquitectura de Computadores. Una forma muy sencilla es por ejemplo, emitir la etapa de WB en orden aunque la ejecución sea desordenada.

Como ejemplo de todo lo visto en esta sección, el procesador Core i7 de Intel tiene 4 cores por chip, su etapa de segmentación es de 16 etapas, es superescalar y se pueden ejecutar 4 instrucciones en paralelo. En cuanto a la caché, tiene 32KB en L1 para I y D, 256KB en L2 y 8MB en L3 que es compartida por los 4 cores.

Jerarquía de memoria

4.1 Abstracción de memoria

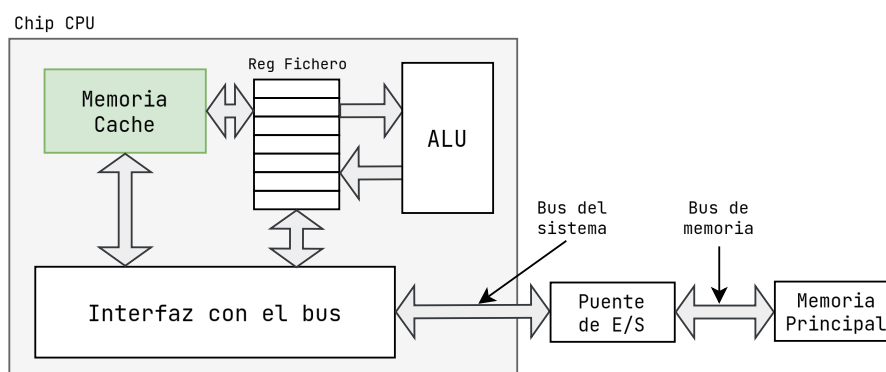
Hasta ahora, hemos visto un modelo simplificado del funcionamiento de un computador, considerando una CPU que ejecuta instrucciones y una memoria que guarda instrucciones y datos para la CPU. Sin embargo, este modelo no refleja como los computadores modernos funcionan realmente.

En la práctica, sistema de memoria consiste en una jerarquía de dispositivos de almacenamiento con distintas capacidades, costes y velocidades. Estas jerarquías funcionan ya que los programas tienden a acceder al almacenamiento de cada nivel más frecuentemente que al de los niveles inferiores. Como programadores, debemos entender esta jerarquía de memoria ya que tiene un gran impacto en el rendimiento de nuestros programas.

4.1.1 Lectura y escritura

Llamamos escritura a la operación de transferir datos de CPU a memoria *movq %rax, 8(%rsp)*. Esto es, la operación store. Por otro lado, decimos que una lectura es la operación que transfiere datos de memoria a CPU, *movq 8(%rsp), %rax*. Esto es, la operación load.

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. Los datos se transfieren entre procesador y DRAM sobre los buses, cada transferencia de datos involucra una serie de pasos llamados transferencias de bus. Como ya hemos dicho, una transferencia de lectura transfiere datos de memoria a CPU, mientras que una transferencia de escritura transfiere datos de CPU a memoria. Veamos una configuración de una computadora de ejemplo:



Veamos ahora un ejemplo de cada tipo de transferencia, comenzamos con una transferencia de lectura (`movq A, %rax`):

1. La CPU pone la dirección de A en el bus de sistema, el puente de E/S redirige la señal al bus de memoria.
2. La memoria principal recibe la dirección A del bus de memoria, recupera la palabra x, y la pone en el bus.
3. La CPU lee la palabra x del bus y la copia al registro `%rax`

En el caso de la escritura (`movq %rax, A`):

1. La CPU pone la dirección de A en el bus de sistema, la memoria principal la recibe y espera a que llegue la palabra de datos correspondiente.
2. La CPU pone la palabra de datos en el bus.
3. La memoria principal recibe la palabra del bus de memoria y la guarda en la dirección A.

4.2 RAM

Random Access Memory, viene en dos variedades, estática y dinámica. Veremos detenidamente cada una de ellas, pero primero presentamos las siguientes definiciones:

- **Tiempo de acceso:** Tiempo que se requiere para leer (o escribir) un dato (palabra) en la memoria, se mide en ciclos o en (nano|micro|pico) segundos.
- **Ancho de banda:** 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 (Kilo|Mega|Giga) bytes por segundo.
- **Métodos de acceso:**
 - Aleatorio (RAM): Tiempo de acceso independiente de la posición a la que se accede, e.g SRAM, ROM.
 - Secuencial (SAM): Tiempo de acceso dependiente de la posición de los datos a los que se accede, e.g cinta magnética.
 - Directo (semialeatorio DASD - direct access storage device): Tiempo de acceso con componente aleatoria y secuencial, e.g disco duro.

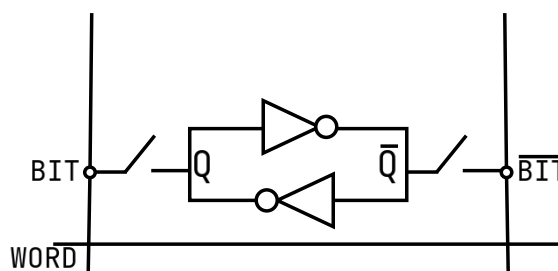
Actualmente, muchos dispositivos tienen 2 o más métodos de acceso. La RAM tradicionalmente se empaqueta como un chip o va empotrada como parte de un chip procesador, su unidad básica de almacenamiento es normalmente una celda (1 bit/celda). Cuando juntamos múltiples chips de RAM formamos una memoria. Como hemos dicho antes, la RAM viene en dos variedades, estática y dinámica.

4.2.1 SRAM

La memoria SRAM guarda cada bit en una celda de memoria biestable. Cada celda se implementa con un circuito de 6 transistores con la capacidad de estar indefinidamente (si hay alimentación) en alguno de dos configuraciones de voltajes distintas o estados. Cualquier otro estado será inestable.

Operación de lectura:

- Se selecciona la celda poniendo WORD a 1 y se conectan Q y \bar{Q} a BIT y \overline{BIT} respectivamente.
- \overline{BIT} se pone a 0 si Q = 1 y a 1 si Q = 0. Es decir, se lee un 1
- BIT se pone a 0 si Q = 0 y a 1 si Q = 1. Es decir, se lee un 0

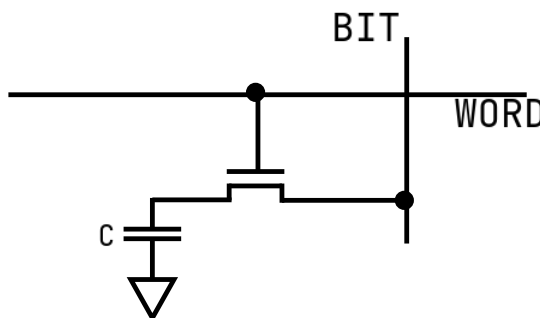


4.2.2 DRAM

La memoria DRAM guarda cada bit como una carga de un condensador, dispuesto de forma vertical. A diferencia de SRAM, la célula de memoria DRAM es muy sensible a cualquier perturbación, por lo que si el voltaje del condensador no es refrescado, el bit se pierde. Distintas corrientes de fuga pueden causar que el bit se pierda en un periodo de 10 a 100 milisegundos, sin embargo, como el ciclo de reloj se mide en nanosegundos, no es un problema. Algunos sistemas usan códigos de error adicionales (pudiendo hacer que palabras de 64 bits se conviertan en palabras de 72 bits) para que si un bit se pierde, se pueda recuperar. En general, la DRAM es más lenta, más sensible a errores, y requiere actualizaciones, sin embargo, es más barata y menos densa que la SRAM ya que esta última necesita más transistores por bit, lo que incrementa el coste y hace que consuma mas potencia.

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, es decir, se produce un pulso de corriente que es detectado por un amplificador de salida, por tanto aparece un 1 en la línea de datos de salida.
- Si C está descargado (=0) no se produce pulso de corriente, por lo que aparece un 0 en la línea de datos de salida.



Debido a la capacidad elevada de los chips de memoria DRAM, las direcciones han de proporcionarse multiplexadas en el tiempo. A modo de resumen:

	Transistores por bit	Tiempo de acceso	Persistencia	Sensibilidad	Coste	Aplicaciones
SRAM	6	1×	Sí	No	1,000×	Memoria caché
DRAM	1	10×	No	Sí	1×	Memoria principal, buffers

Table 4.1: Comparación de SRAM y DRAM.

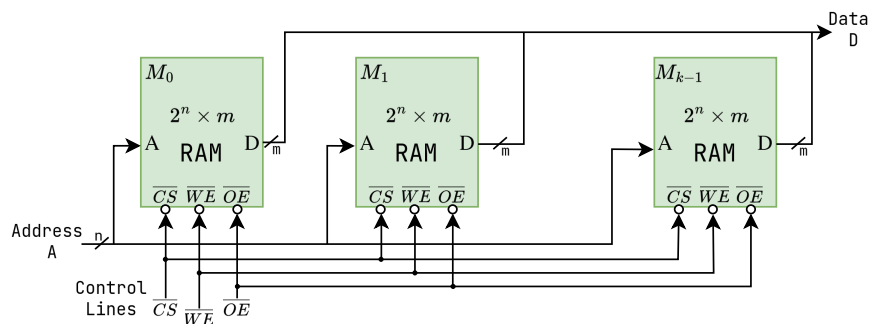
4.3 Configuración y diseño de memorias usando varios chips

4.3.1 Ampliación de la memoria

Se nos presenta el problema de construir una memoria de 2^N palabras de M bits a partir de chips de 2^n palabras de m bits. Para ello, veremos por separado como se amplía cada parte.

Incrementar el ancho de la palabra de m a $k \cdot m = M$ bits

Para este fin, se utiliza la llamada técnica de *bit-slicing*, esta técnica consiste en utilizar módulos de chips de menor tamaño de palabra, conectandolos de tal forma que cada uno se encargue de procesar un "slice" de la palabra completa.

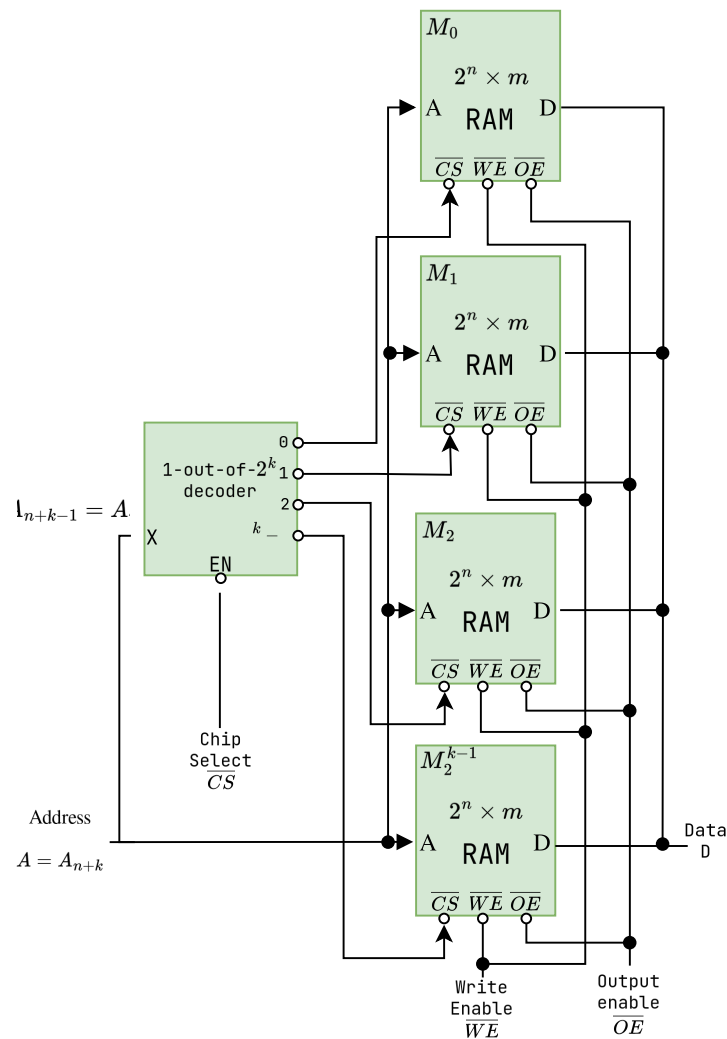


De esta forma, junto con líneas de control:

- **CS:** (Chip select) Selección de chip
- **WE:** (Write enable) Habilitación de escritura
- **OE:** (Output enable) Habilitación de salida

Incrementar el número de palabras de 2^n a $2^{n+k} = 2^N$

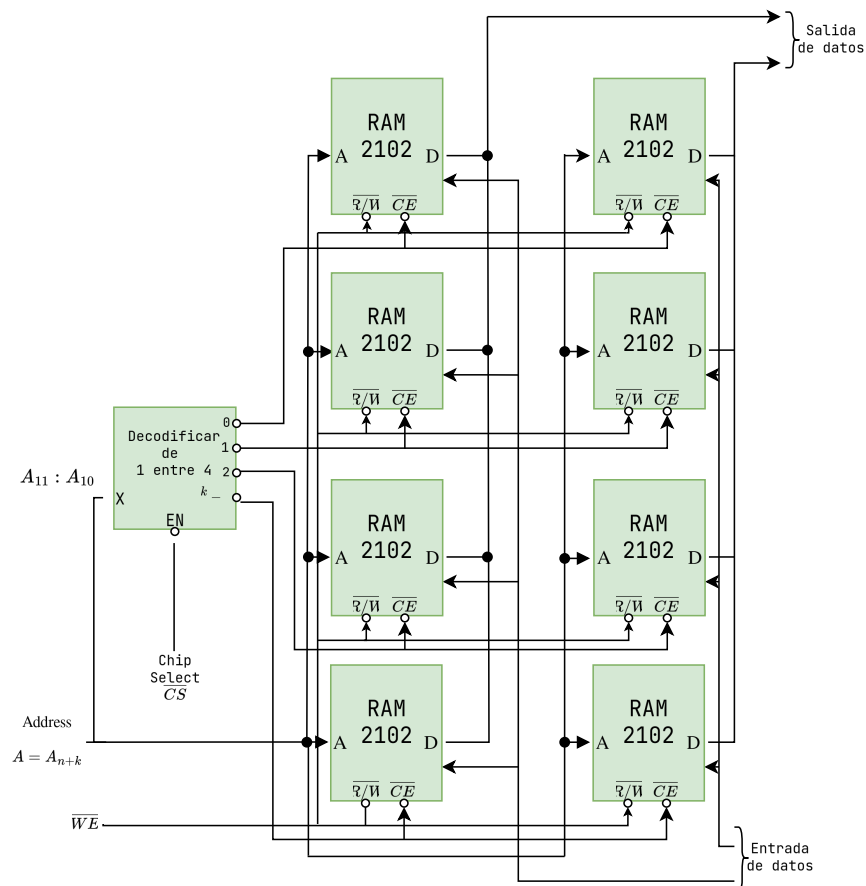
En este caso necesitamos 2^k circuitos de 2^n palabras y un decodificador de 1 entre 2^k .



Las k líneas de dirección de orden superior ($A_{n+k-1} : A_n$) se conectan a las entradas de un decodificador de k a $2^k \Rightarrow$ cada configuración de los $n + k$ bits hace que se seleccione solamente el circuito RAM indicado por los bits de dirección ($A_{n+k-1} : A_n$). Las 2^k líneas de salida del decodificador se conectan a las entradas de selección \overline{CS} de los 2^k circuitos. Las n líneas de dirección de orden inferior se conectan en común a las líneas de dirección de los 2^k chips.

4.3.2 Incrementar simultáneamente el número de palabras y el ancho de palabra

En este caso basta con combinar las dos técnicas anteriores



4.3.3 Módulos de memoria en línea

En los años 80, la memoria solía soldarse directamente en la placa base del ordenador. Pero a medida que aumentaron los requisitos de memoria, esta técnica resultó poco factible. Es por ello, que los chips de memoria DRAM se enpaquetan en módulos de memoria que se insertan en zócalos de la placa base. Los módulos de memoria actuales (SIMM, DIMM, SODIMM) los veremos en breves. De esta forma, obtenemos un método flexible para actualizar la memoria, ocupando menos espacio de la placa base. Normalmente sólo se ampliaba el bus de datos, no el de direcciones (no había necesidad de un decodificador).

SIMM

SIMM (Single Inline Memory Module) es un módulo de memoria en el que cada contacto de una cara está conectado con el alineado den la otra cara para formar un único contacto. En los 80 y 90, FPM DRAM y EDO DRAM eran los tipos de memoria más comunes. Venían en módulos de 30 pines (8 bits) y 72 pines (32 bits).

- SIMM 30-pin: Tiene 12 líneas de dirección, lo que da un máximo de 24 bits de dirección. Con un ancho de palabra de 8 bits, la capacidad máxima es de 16 MB.
- SIMM 72-pin: Tiene 12 líneas de dirección, lo que da un máximo de 24 bits de dirección. Con un ancho de palabra de 32 bits, la capacidad máxima es de 128 MB.

DIMM

DIMM (Dual Inline Memory Module) es un módulo de memoria en el que los contactos opuestos están aislados electrónicamente para formar dos contactos separados. Son los usados desde los 90 hasta hoy, (SDRAM, DDR,

DDR2, DDR3, DDR4, DDR5). Con 64 bits de datos (72 bits con ECC), los pines han ido aumentando, de 168 a 288. Con 256GiB de capacidad máxima en 2024.

SODIMM

SO-DIMM (Small Outline DIMM) es una versión más pequeña de DIMM, usada en portátiles y otros dispositivos. Tiene menos tamaño y menos pines que un DIMM. Variando entre 100 hasta 260 pines

4.3.4 Localidad

A lo largo de los años, la brecha entre las velocidades de disco, DRAM y CPU se ha ensanchado, la solución a este problema ha sido la **localidad**. Los programas (bien escritos) tienden a exhibir buena localidad, esto es, tienden a referenciar segmentos de datos que están cerca unos de otros o incluso a referenciarlos a ellos mismos en un futuro cercano. Esta propiedad, llamada **principio de localidad**, es un principio fundamental en el diseño y rendimiento de las aplicaciones y sistemas. Normalmente se subdivide en localidad temporal y espacial:

- **Localidad temporal:** Si un dato es referenciado, es probable que sea referenciado de nuevo en un futuro cercano.
- **Localidad espacial:** Si un dato es referenciado, es probable que datos cercanos a él sean referenciados en un futuro cercano.

Este, junto con la caché, es probablemente uno de los conceptos de esta asignatura que más deberíamos tener en cuenta en el diseño de programas, ya que, en general, los programas con buena localidad corren mucho más rápido que aquellos que no la tienen.

Los sistemas modernos, desde el hardware hasta los programas de aplicación, están diseñados para aprovechar la localidad. A nivel de hardware, el principio de localidad permite a los diseñadores de computadoras acelerar el acceso a la memoria principal mediante la introducción de memorias pequeñas y rápidas conocidas como cachés que mantienen bloques de las instrucciones y datos más recientemente referenciados. A nivel de sistema operativo, el principio de localidad permite que el sistema utilice la memoria principal como una caché de los fragmentos de la dirección virtual más recientemente referenciados. De manera similar, el sistema operativo utiliza la memoria principal para almacenar en caché los bloques de disco más recientemente utilizados en el sistema de archivos.

Ejemplo 4.1: Veamos algunos ejemplos de esto para consolidar el concepto:

```
int sum(int *a, int n) {  
    size_t i{0};  
    size_t j{0};  
    int sum{0};  
  
    for (i = 0; i < n; ++i) {  
        for (j = 0; j < n; ++j) {  
            sum += a[i][j];  
        }  
    }  
    return sum;  
}  
  
int sum(int *a, int n) {  
    size_t i{0};  
    size_t j{0};  
    int sum{0};  
  
    for (i = 0; i < n; ++i) {  
        for (j = 0; j < n; ++j) {  
            sum += a[j][i];  
        }  
    }  
    return sum;  
}
```

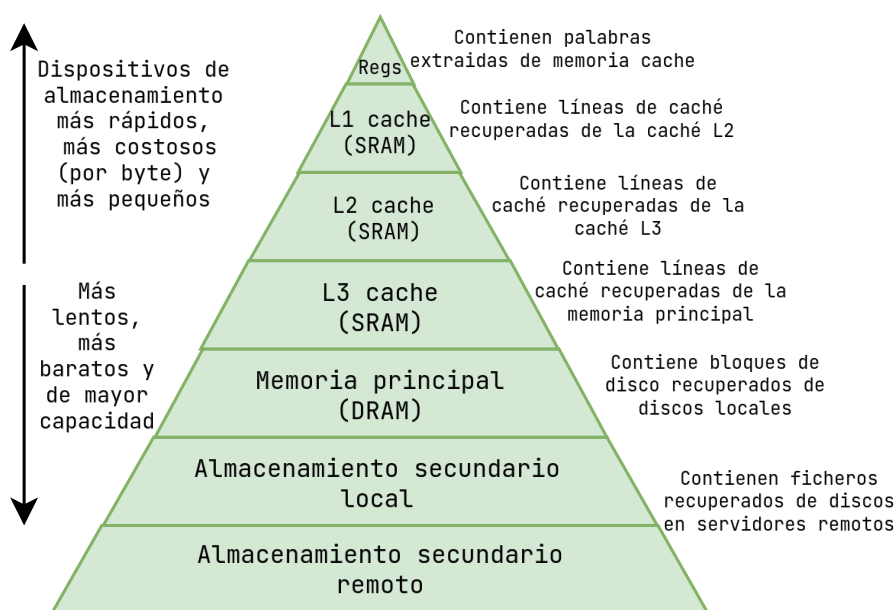
En ambos casos, tenemos buena localidad temporal para la variable `sum`, que es referenciada cada iteración del bucle, sin embargo para el acceso a la matriz `a`, en el primer caso por la manera en la que se dispone en memoria (fila a fila), tenemos buena localidad espacial ya que accedemos, a cada elemento de forma secuencial, a esto se le llama *stride-1*, en el segundo caso, vamos dando saltos en memoria en cada iteración del bucle.

4.4 Jeraquía de memoria

Como ya hemos descrito anteriormente, el hardware y software tienen ciertas propiedades fundamentales y perdurables:

- **Tecnología de almacenamiento:** Las tecnologías de almacenamiento rápidas cuestan más por byte, tienen menor capacidad y requieren más potencia \equiv calor
- **Localidad:** Los programas bien escritos tienden a exhibir buena localidad

Estas propiedades sugieren un enfoque para organizar sistemas de memoria y almacenamiento conocido como **jerarquía de memoria** que es usado en todas las computadoras modernas.

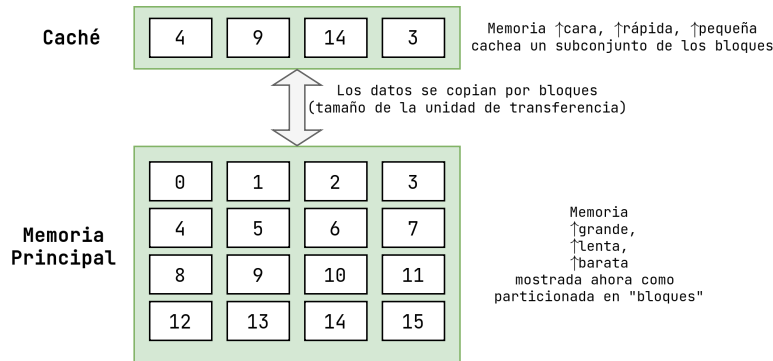


En general, los dispositivos de almacenamiento se vuelven más baratos, más lentos y tienen mayor capacidad a medida que nos movemos hacia abajo en la jerarquía. En el nivel más alto (L0 o Registros) tenemos un pequeño número de registros de CPU a los que se puede acceder en un único ciclo de reloj, después están las cachés basadas en SRAM a las que se puede acceder en unos pocos ciclos de reloj, seguidas de la memoria principal basada en DRAM a la que se puede acceder en cientos de ciclos de reloj, y finalmente los dispositivos de almacenamiento basados en tecnologías de almacenamiento más lentas y de mayor capacidad.

4.4.1 Caché

En general, una caché es un pequeño y rápido dispositivo de almacenamiento que actúa como una zona de trabajo temporal para los objetos de datos guardados en un disp. de almacenamiento más lento y de mayor capacidad.

La idea general de la jerarquía de memoria es que para cada k el dispositivo de nivel k actúa como una caché para el dispositivo de nivel $k + 1$. Esto funciona bien ya que debido a la localidad, los programas suelen acceder a los datos a nivel k más a menudo que a los datos a nivel $k + 1$. En consecuencia, tenemos un conjunto de almacenamiento que cuesta como el más barato y proporciona un rendimiento como el más rápido.



Los datos son siempre copiados de un nivel a otro en tamaños de bloque denominados unidades de transferencia de caché. Notemos que aunque el tamaño entre dos niveles es fijo, no tiene por qué ser el mismo entre otros dos niveles.

Hit y Miss

- **Hit:** Cuando un programa necesita un objeto d del nivel $k + 1$, primero busca d en uno de los bloques en el nivel k , si d resulta estar en k (cacheado) entonces se produce lo que llamamos un *hit*, el programa lee d directamente de k lo que por la naturaleza de la jerarquía es mucho más rápido que leerlo de $k + 1$.
- **Miss:** Por otro lado, si d no está cacheado en k , entonces se produce un *miss*, cuando esto ocurre, la caché de nivel k capta un bloque de nivel $k + 1$ que contiene d y lo guarda en k para futuros accesos posiblemente sobrescribiendo algún otro bloque. Este proceso se llama *reemplazo*, al bloque que se sobrescribe se le llama *víctima*. La decisión de qué bloque reemplazar es tomada por una *política de reemplazo*. (LRU, FIFO, Round Robin, etc).

4.4.2 Tipos de Fallos

- **Fallo en frío:** Los fallos en frío ocurren cuando la caché está vacía, lo cual es típico al inicio de un programa o después de un reinicio. Como no hay datos previamente almacenados, cualquier referencia a un bloque provocará un fallo.
- **Fallo de capacidad:** Los fallos por capacidad ocurren cuando el tamaño de la caché es insuficiente para almacenar el conjunto activo de bloques referenciados, también conocido como el conjunto de trabajo.
- **Fallo por conflicto:** Los fallos por conflicto ocurren cuando la caché es suficientemente grande para almacenar los datos referenciados, pero debido a restricciones de ubicación, varios bloques son mapeados al mismo lugar. Como resultado, los datos son reemplazados frecuentemente, provocando fallos repetitivos. Por ejemplo, en una correspondencia directa, un bloque i del nivel $k + 1$ debe mapearse al bloque $(i \bmod 4)$ en el nivel k . Esto significa que los bloques 0, 4, 8, y 12 del nivel $k + 1$ se asignarían al bloque 0 del nivel k , y una referencia a 0,8,0,8,0,8 provocaría fallos continuamente.

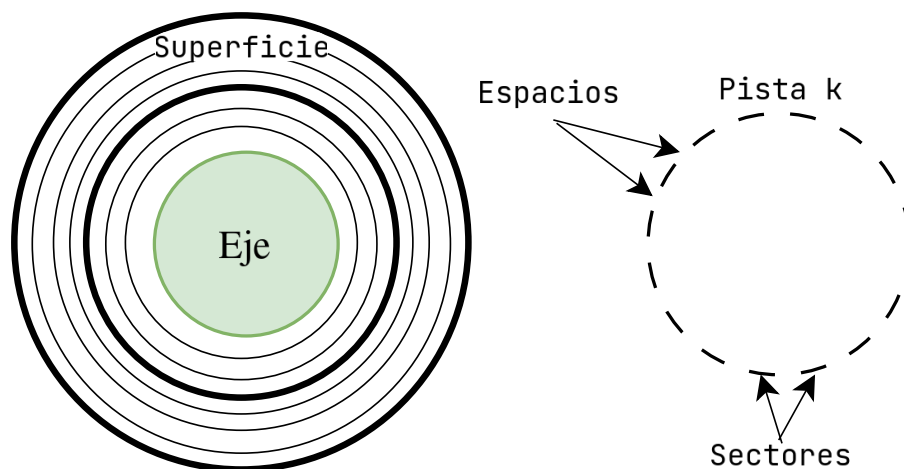
4.5 Tecnologías de almacenamiento

Los discos son dispositivos de almacenamiento que guardan enormes cantidades de datos, en el orden de los miles de GB a diferencia de los miles de MB de la memoria principal. Sin embargo, los discos son mucho más

lentos que la memoria principal, alrededor de cien mil veces más lentos. Distinguimos entre discos magnéticos y memorias flash. En los discos magnéticos, se accede a los datos electromecánicamente mientras que en una memoria flash se implementa el almacenamiento con una estructura 3D de celdas de memoria.

4.5.1 Geometría de un disco

Los discos consisten en platos con dos superficies (caras). Cada cara consisten en anillos concéntricos llamados pistas, cada pista consisten en sectores separados por espacios.



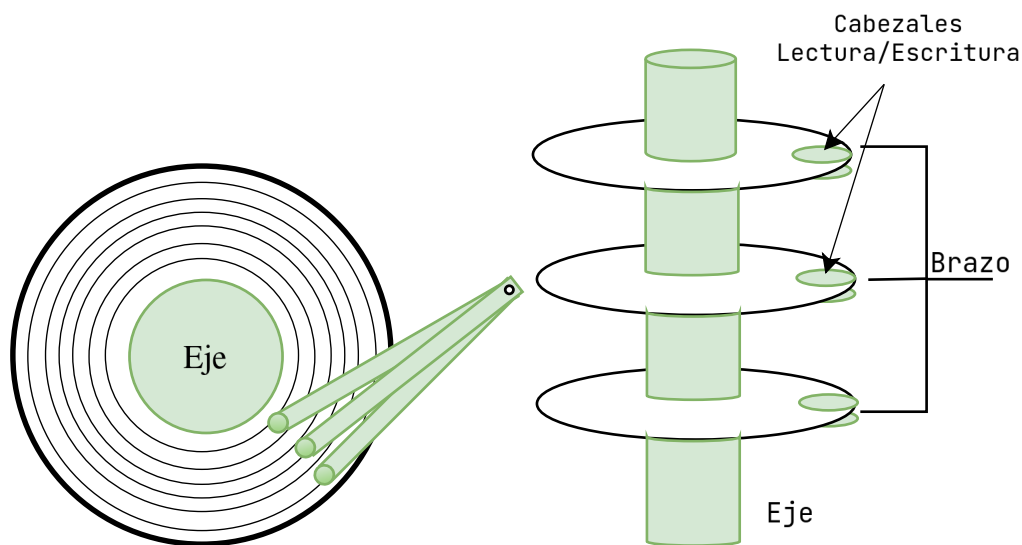
4.5.2 Capacidad de un disco

El máximo número de bits que pueden ser guardados por un disco se conoce como su capacidad máxima o simplemente capacidad, viene expresada en unidades de GB o TB. La capacidad de un disco depende de varios factores:

- **Densidad de grabación:** El número de bits que se pueden almacenar en una pista de una pulgada.
- **Densidad de pistas (radial):** El número de pistas que se pueden comprimir en un tramo radial de una pulgada.
- **Densidad superficial:** Producto de ambas densidades (grabación y radial).

4.5.3 Funcionamiento de un disco

Si miramos desde un único plato, el disco lee y escribe bits guardados en la superficie magnética usando una cabeza de lectura/escritura conectada a un brazo actuador que la desplaza a lo largo de su eje radial permitiendo colocar el brazo sobre cualquier pista.



Si miramos desde varios platos (derecha), los cabezales de lectura/escritura se mueven al unísono de cilindro a cilindro.

4.5.4 Tiempo de acceso

El tiempo de acceso a disco puede calcularse como la suma de tres componentes principales:

$$T_{\text{acceso}} = T_{\text{prom. búsqueda}} + T_{\text{prom. rotación}} + T_{\text{prom. transferencia}}$$

Tiempo de Búsqueda ($T_{\text{prom. búsqueda}}$)

- Es el tiempo necesario para mover los cabezales del disco hacia el cilindro que contiene el sector objetivo.
- El tiempo promedio de búsqueda ($T_{\text{prom. búsqueda}}$) en discos modernos es típicamente de 3 a 9 ms.

Latencia Rotacional ($T_{\text{prom. rotación}}$)

- Una vez que el cabezal está posicionado, se debe esperar a que el primer bit del sector pase bajo el cabezal.
- El tiempo máximo de latencia rotacional es:

$$T_{\text{máx. rotación}} = \frac{1}{\text{RPM}} \times \frac{60 \text{ s}}{1 \text{ min}}$$

- El tiempo promedio de latencia rotacional es la mitad del máximo:

$$T_{\text{prom. rotación}} = \frac{1}{2} \times T_{\text{máx. rotación}}$$

- Ejemplo típico: para 7,200 RPM:

$$T_{\text{prom. rotación}} = \frac{1}{2} \times \frac{1}{7,200} \times 60 \approx 4.17 \text{ ms}$$

Tiempo de Transferencia ($T_{\text{prom. transferencia}}$)

- Es el tiempo requerido para leer o escribir los bits del sector.
- Se puede calcular aproximadamente como:

$$T_{\text{prom. transferencia}} = \frac{1}{\text{RPM}} \times \frac{1}{\text{Prom. sectores/pista}} \times \frac{60 \text{ s}}{1 \text{ min}}$$

El tiempo total para acceder a un sector depende de la posición previa del cabezal y de la rotación actual del disco. Para optimizar el rendimiento, los sistemas operativos intentan minimizar el tiempo de búsqueda agrupando solicitudes cercanas. El primer bit de un sector es el más caro (lento), el resto sale gratis.

4.5.5 Bus de E/S

Los dispositivos de entrada salida como las tarjetas gráficas, monitores, ratones, teclados, discos, etc, se conectan a la CPU a través de un bus de E/S, a diferencia del bus de sistema y memoria, que son específicos de la CPU, los buses de E/S están diseñados para ser independientes de la CPU:

4.5.6 Memorias no volátiles

Las memorias **SRAM** y **DRAM** son volátiles, es decir, pierden su contenido cuando se apagan. Por el contrario, las memorias no volátiles retienen su información incluso cuando no tienen alimentación.

Tipos de Memorias No Volátiles

- **Memoria de sólo lectura (ROM):** Es programada durante su fabricación y no puede modificarse posteriormente.
- **PROM (Programmable ROM):** Programable por el usuario de manera irreversible.
- **EPROM (Erasable PROM):** Puede borrarse utilizando luz ultravioleta para reprogramarla.
- **EEPROM (Electrically Erasable PROM):** Permite el borrado y la reprogramación eléctrica.
- **Memorias Flash:**
 - Son un tipo de EEPROM con capacidad de borrado parcial por bloques.
 - Limitación: suelen desgastarse tras aproximadamente 100,000 ciclos de borrado.
- **3D XPoint (Intel Optane) y Memorias No Volátiles Emergentes:** Utilizan nuevos materiales para mejorar la velocidad y la durabilidad.

Aplicaciones de las Memorias No Volátiles

- Almacenamiento de programas de firmware en ROM, tales como: BIOS, Controladoras de disco, Tarjetas de red, Aceleradores gráficos, Subsistemas de seguridad,
- **Discos de estado sólido (SSD):** Reemplazan los discos duros tradicionales con mayor velocidad y fiabilidad.
- **Cachés de disco:** Mejoran el rendimiento del almacenamiento secundario.

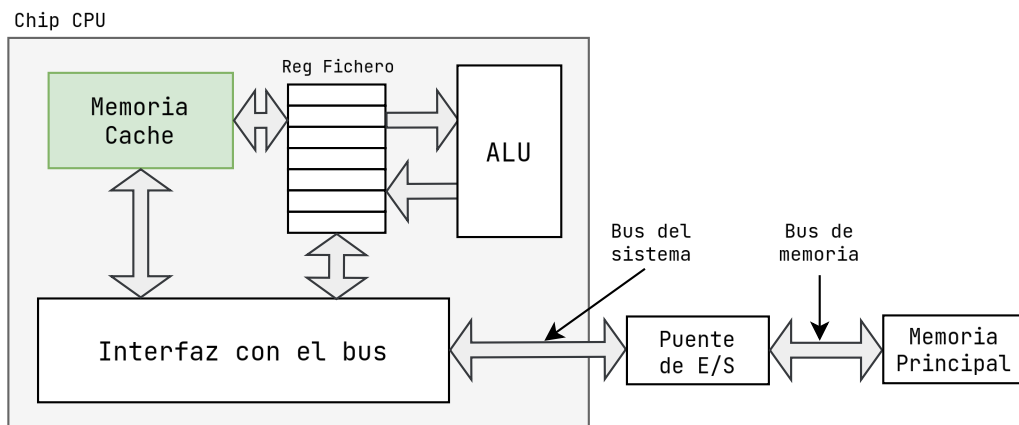
4.5.7 Discos de estado sólido

Un SSD tiene acceso secuencial significativamente más rápido que aleatorio , siendo las escrituras aleatorias especialmente lentas debido al tiempo de borrado de bloques (1 ms) y la necesidad de copiar otras páginas útiles a un nuevo bloque antes de modificar una página. La capa de traducción flash acumula pequeñas escrituras para optimizar el rendimiento. Ventajas: sin partes móviles, son más rápidos, resistentes y eficientes energéticamente. Desventajas: eventual desgaste mitigado por la lógica de nivelado de desgaste; por ejemplo, un Samsung 970 EVO Plus permite escribir 600 veces su capacidad antes de desgastarse. Aunque en 2019 eran 4 veces más caros por byte que discos giratorios, el costo sigue disminuyendo. Aplicaciones: MP3, smartphones, portátiles, y cada vez más comunes en servidores y PC de sobremesa.

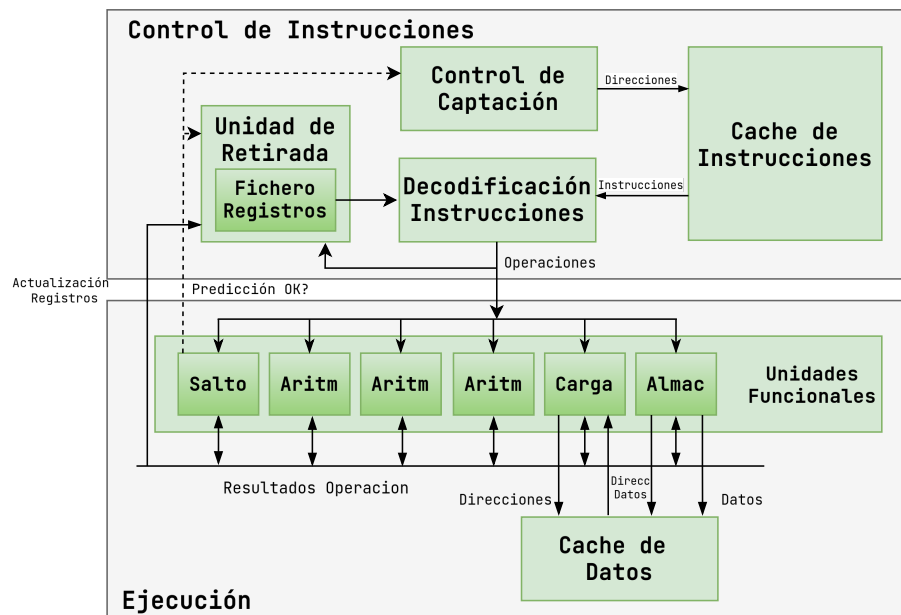
Tema 5

Caché

Las memorias caché son memorias pequeñas y rápidas basadas en SRAM gestionadas automáticamente por el hardware. Su objetivo es retener bloques de memoria principal accedidos frecuentemente. La CPU busca los datos primero en caché, la estructura es la siguiente:



El diseño moderno de una CPU es de la siguiente forma:



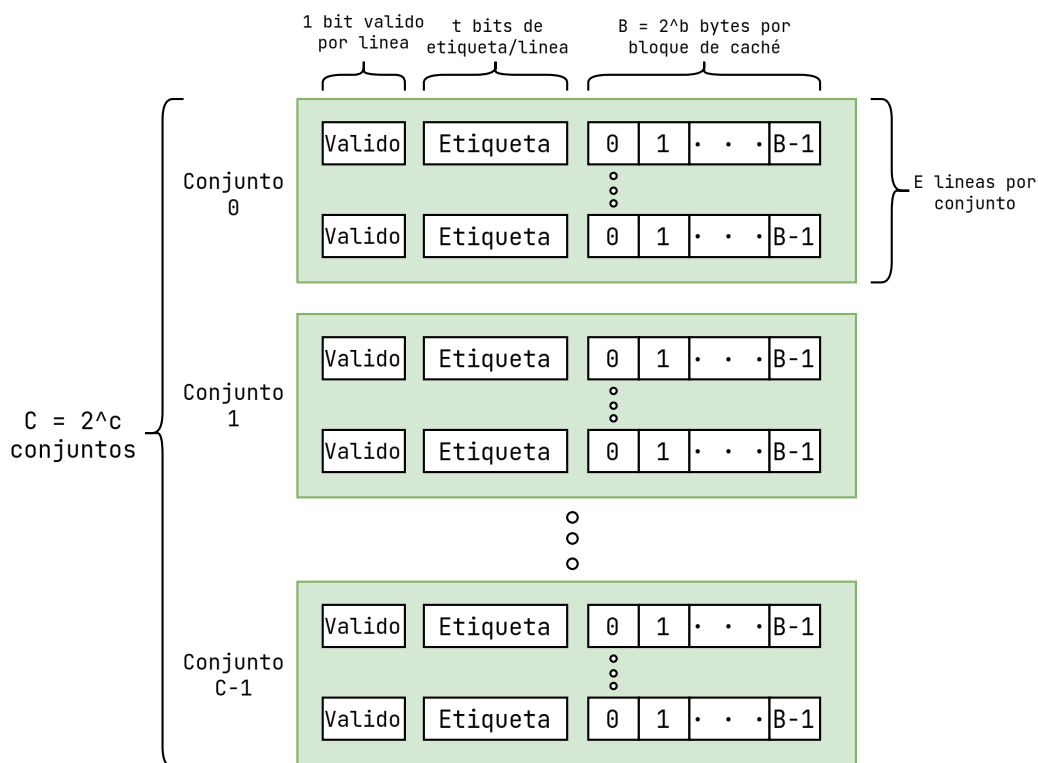
Se caracteriza por ser *superescalar* (capaz de realizar múltiples operaciones por ciclo de reloj) y de ejecución *fuera de orden* (sin respetar necesariamente el orden del programa). Se divide principalmente en dos unidades:

- **Unidad de Control de Instrucciones (ICU):** Encargada de leer las instrucciones desde la memoria, realizar predicción de saltos, y descomponerlas en operaciones primitivas (micro-operaciones).
- **Unidad de Ejecución (EU):** Responsable de ejecutar las operaciones utilizando unidades funcionales especializadas (aritmética, carga, almacenamiento, etc.), con soporte para ejecución especulativa y paralelismo.

Ambas unidades trabajan con memoria caché de alta velocidad para minimizar tiempos de acceso y optimizar el rendimiento del procesador. Como podemos imaginarnos, el aspecto real de una CPU es más complejo.

5.0.1 Organización general de Cache

Consideramos un sistema en el que cada dirección de memoria tiene m bits que forman 2^m direcciones. Una caché para un sistema de estas características se organiza como un array de 2^c conjuntos de caché. Cada conjunto consiste en E líneas de caché, cada línea consiste en un bloque de datos B de 2^b bytes. Un bit de estado válido que indica si la línea contiene o no información válida y $t = m - (b + c)$ bits de etiqueta que identifican el bloque guardado en la línea de caché:



Cuando el procesador ejecuta una instrucción de carga para leer una palabra de la dirección A en la memoria principal, envía esa dirección a la caché. El proceso para determinar si la caché contiene una copia de la palabra solicitada es el siguiente:

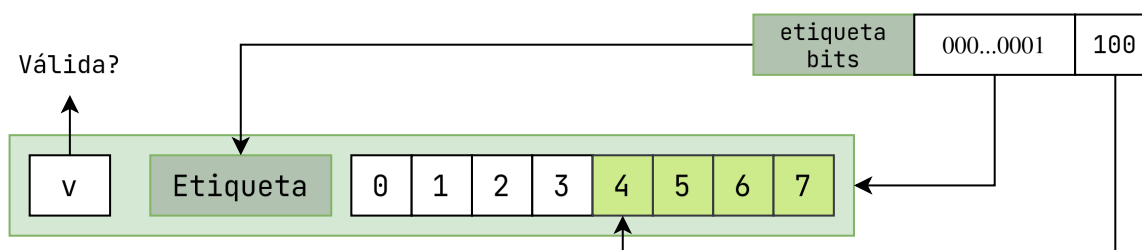
1. La dirección A se divide en tres campos: *etiqueta*, *índice conjunto* y *offset*.
2. Los bits de *índice de conjunto* se utilizan para seleccionar el conjunto en la caché donde podría estar almacenada la palabra.

3. Dentro del conjunto, los bits de *etiqueta* identifican la línea que puede contener la palabra. Esto se confirma si el bit de validez está activado y los bits de *etiqueta* coinciden.
4. Finalmente, los bits de *offset* especifican la posición exacta de la palabra dentro del bloque de datos.

A continuación, se muestra la partición de la dirección en sus respectivos campos:

Etiqueta (t bits)	Indice conjunto (c bits)	offset (b bits)
$m - t - c$	c	b

Las cachés se dividen en diferentes grupos en función del número de líneas por conjunto (E). Una caché con una única línea por conjunto ($E = 1$) se conoce como una caché con correspondencia directa. Veamos un ejemplo de su funcionamiento, para ello, sea el tamaño del bloque $B = 8$ bytes, buscaremos en la primera línea el segundo entero guardado (4 bytes):



Si la etiqueta no coincide, se producirá un fallo.

Ejemplo 5.1: Veamos un ejemplo del funcionamiento de la caché en correspondencia directa, para ello, consideraremos un direccionamiento de 4 bits, con 4 conjuntos, 2 bytes por bloque y cada palabra es un byte. De esta forma, tendremos un total de 16 direcciones posibles, las describimos en la siguiente tabla:

Dirección	Tag bits ($t = 1$)	Bits índice ($s = 2$)	Offset bits ($b = 1$)	Número bloque
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Ahora simulamos la caché en acción, en primer lugar está vacía (Se añade la columna Conjunto para facilitar la comprensión sin embargo en la caché sabemos que no hay tal cosa). Realizaremos la siguiente secuencia de lecturas:

Conjunto	Valido	Etiqueta	Bloque
0	0		
1	0		
2	0		
3	0		

1. Leer la palabra en la dirección 0: Como el bit de válido para el conjunto 0 es 0, esto es un *cache-miss* por lo que la caché capta el bloque 0 de memoria principal (o de un nivel inferior de caché) y guarda el bloque en el conjunto 0. Tras esto la caché retorna m[0].

Conjunto	Valido	Etiqueta	Bloque
0	1	0	M[0-1]
1	0		
2	0		
3	0		

2. Leer la palabra en la dirección 1: Esto es un *hit*, por tanto la caché retorna el bloque 1 de la línea de caché, el estado de la caché no cambia.
3. Leer la palabra en la dirección 13: Como el bit de válido para el conjunto 2 es 0, esto es otro *cache-miss* por lo que la caché cata el bloque 6 de memoria principal y guarda el bloque en el conjunto 2. Tras esto la caché retorna m[13].

Conjunto	Valido	Etiqueta	Bloque
0	1	0	M[0-1]
1	0		
2	1	1	M[12-13]
3	0		

4. Leer la palabra en la dirección 8: Esto es un *miss* ya que aunque el bit para la línea en el conjunto 0 es válido, la etiqueta (tag) no coincide, por lo que la caché carga el bloque 4 en el conjunto 0 sustituyendo el bloque anterior. Tras esto la caché retorna m[8].

Conjunto	Valido	Etiqueta	Bloque
0	1	1	M[8-9]
1	0		
2	1	1	M[12-13]
3	0		

5. Leer la palabra en la dirección 0: Esto es otro *miss* ya que desafortunadamente, acabamos de reemplazar el bloque 0, por lo que la caché carga el bloque 0 en el conjunto 0 sustituyendo el bloque 4. Tras esto la caché retorna m[0].

Conjunto	Valido	Etiqueta	Bloque
0	1	1	M[0-1]
1	0		
2	1	1	M[12-13]
3	0		

5.0.2 Escrituras

Como ya hemos visto, las operaciones de lectura son directas, en el caso de la escritura, supongamos que una palabra w ya está en caché (*write hit*), tras actualizarla, qué pasa con la copia de w en el siguiente nivel de la jerarquía? Existen dos alternativas:

- **Write through:** Se actualiza el bloque inferior inmediatamente, este enfoque es simple pero tiene la desventaja de causar un tráfico en el bus tras cada escritura.
- **Write back:** Aplaza la actualización lo máximo posible actualizando el bloque en el nivel inferior solo cuando sale de la caché por reemplazo. Por la localidad, se puede reducir significativamente el tráfico del bus pero tiene la desventaja de que introduce una complejidad adicional, ya que se debe mantener un bit adicional (*dirty bit*) para cada línea de caché que indica si el bloque ha sido o no modificado.

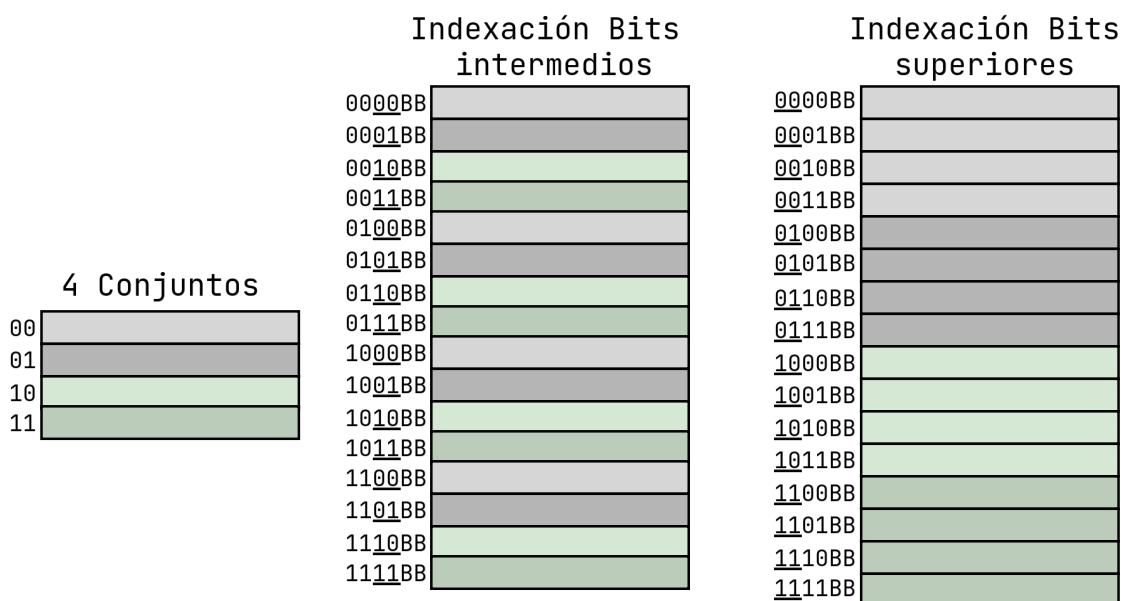
En el caso de que la palabra no esté en caché (*write miss*), se puede optar por dos estrategias:

- **Write allocate:** Se carga el bloque completo desde el siguiente nivel de la jerarquía a la caché y se actualiza la línea, con este enfoque se trata de aprovechar la localidad espacial de las escrituras pero tiene la desventaja de que cada *miss* implica una transferencia desde el nivel inferior.
- **No write allocate:** Se actualiza el bloque en el siguiente nivel de la jerarquía sin cargarlo en caché, este enfoque es más rápido pero tiene la desventaja de que no se aprovecha la localidad espacial.

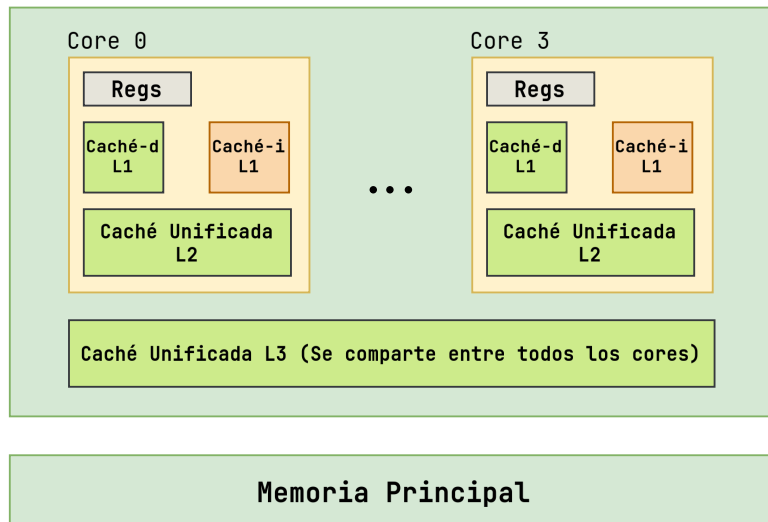
Estas estrategias se suelen combinar en los pares *Write through + No write allocate* y *Write back + Write allocate*. Cuando intentemos programar un programa *cache-friendly* asumiremos que se sigue el modelo *write back + write allocate* ya que los detalles varían de un sistema a otro y no son públicos, sin embargo es la estrategia más común.

5.0.3 Por qué indexar los bits intermedios

Es posible que surja la pregunta de por qué se indexan los conjuntos con los bits intermedios en lugar de los bits más significativos, la respuesta es que si se indexan con los bits más significativos, los bloques que esten contiguos en memoria serán asignados al mismo conjunto, esto puede ser un problema si se accede a bloques consecutivos ya que se producirán *misses* en la caché. Para verlo mejor, consideremos una caché con 4 conjuntos (00, 01, 02, 03) y una memoria de 64 Bytes, es decir, 6 bits para direcciones:



En el caso de indexación por bits intermedios, las direcciones son de la forma: TTSSBB y en el caso de indexación por bits más significativos SSTTBB. Veamos un ejemplo de un acceso en una cache de datos L1 de un intel core i7:



- **Cache L1:** 32KB, 8 vías, Acceso 4 ciclos.
- **Caché unificada L2:** 256KB, 8 vías, Acceso 10 ciclos.
- **Caché unificada L3:** 8MB, 16 vías, Acceso 40-75 ciclos.
- **Tamaño de bloque:** 64 Bytes.

Como hemos dicho que el acceso será a L1, tenemos 32KB en 8 vías, con 64 Bytes por bloque y 47 bits de dirección.

- $B = 64$ Bytes, por tanto necesitamos $\log_2(64) = 6$ bits para el offset.
- $V = 8$ vías, por tanto necesitamos $\log_2(8) = 3$ bits para el índice.
- Tamaño total = 32KB, como $\text{Tamaño} = C \times V \times B$, despejando se tiene $C = \frac{2^{15}}{2^9} = 2^6$, por tanto necesitamos 6 bits para el índice de conjunto c.

En este caso, dada ahora la dirección de pila 0x0000 7f72 62a1 e010 la descomponemos de la siguiente manera:

- Usamos 6 bits para el desplazamiento (b) y 6 para el índice de conjunto (c), es decir, 12 bits en total, de nuestra dirección en hexadecimal esto corresponde a los 3 bits menos significativos de la dirección ya que como bien sabemos, cuatro bits en binario se corresponden a un dígito en hexadecimal.
- Usamos ahora que el tamaño de etiqueta es $47 - 12 = 35$ bits, por tanto, los 35 siguientes bits serán la etiqueta.
- Teniendo en cuenta un bit de válido, la etiqueta será por tanto: 0x7f 7262 a1e0

Veamos un ejemplo más para comprender perfectamente los conceptos:

Ejemplo 5.2: Sea un procesador cuya jerarquía de memoria se distribuye en 1GB de memoria principal, 64B de tamaño de bloque y la caché:

- L1: 32KB de 8 vías
- L2: 128 KB de 8 vías
- L3: 16MB de 16 vías

Veamos cómo podemos calcular el tamaño de cada campo y el total ocupado por la etiqueta (por ejemplo):

En primer lugar, nos fijamos en que la memoria principal ocupa $1GB = 2^{30}B$, es decir, se tienen 30 bits de dirección, como el tamaño de bloque es $64B = 2^6B$, se necesitan 6 bits para el offset, en el caso en particular de L3, tenemos $16MB = 2^{24}B$ y $16 = 2^4$ vías, es decir, el campo de vías necesitará 4 bits. A partir de esta información ya podemos calcular lo que queremos:

- El campo de conjunto tendrá $24 - 6 - 4 = 14$ bits.
- El campo de etiqueta tendrá $30 - 14 - 6 = 10$ bits.
- El tamaño total que ocupan las etiquetas en L3 será el número de bits para el campo de etiqueta, multiplicado por el número de líneas por conjunto y el número de conjuntos, es decir: $10 \times 16 \times 2^{14} \approx 2.5MB$.

■

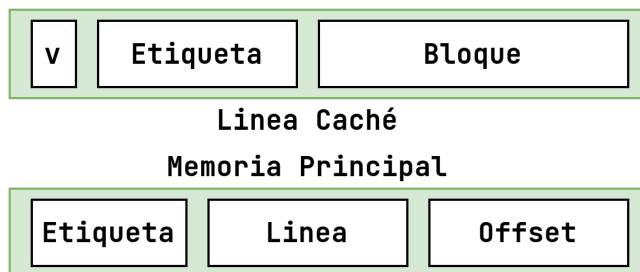
5.1 Políticas de Colocación

5.1.1 Correspondencia Directa

El bloque i de la memoria principal (MP) se asigna a la línea $i \bmod 2^l$ de la caché. Es decir, la caché está compuesta únicamente por líneas, esto supone que cada línea de caché contenga un bit de validez, los bits de etiqueta y el contenido del bloque. Respecto a la memoria principal, contiene la etiqueta, la línea correspondiente y el offset.

$$\text{Bloque } i \text{ de MP} \implies \text{línea } i \bmod 2^l \text{ de caché.}$$

Ejemplo: Veamos una representación de una línea de caché junto con memoria principal:



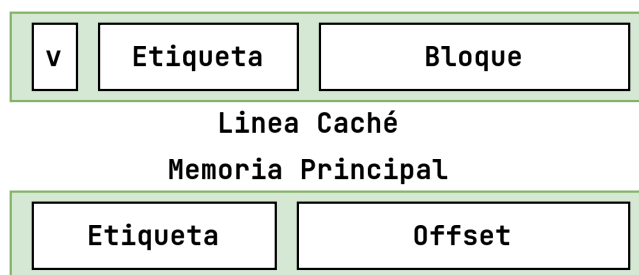
Supongamos por tanto por ejemplo (ignorando bit de válido) que tenemos una caché de $128KB \rightarrow 16$ bits, con un tamaño de bloque de $64B \rightarrow 6$ bits, entonces tendremos 2^{10} líneas, es decir, necesitamos 10 bits para la línea y el resto del tamaño es para el campo de etiqueta. Esto trae como ventaja la simplicidad y el bajo coste, sin embargo, si dos o mas bloques, utilizados alternativamente, corresponden al mismo marco de bloque, se producirán fallos de caché.

5.1.2 Correspondencia Totalmente Asociativa

El bloque i de la memoria principal puede residir en cualquier línea de la caché. En este caso, no hay un índice de conjunto, y todas las líneas son comparadas simultáneamente para verificar si el bloque está presente.

$$\text{Bloque } i \text{ de MP} \implies \text{cualquier línea de la caché.}$$

Ejemplo: Veamos una representación como en el caso anterior:



Si la dirección de la memoria principal tiene $m = 18$ bits, la etiqueta es la concatenación de los $m - b$ bits más significativos de la dirección de la memoria. Con este enfoque se tiene una mayor flexibilidad ya que permite cualquier combinación de bloques de memoria principal en la caché y elimina en gran medida los conflictos entre bloques, sin embargo, es costosa y compleja de implementar.

5.1.3 Correspondencia Asociativa por Conjuntos

La caché se divide en 2^c conjuntos disjuntos, con 2^b marcos de bloque por conjunto. El bloque i de la memoria principal se asigna al conjunto $i \bmod 2^c$, pero dentro del conjunto, el bloque puede residir en cualquier marco de bloque.

Bloque i de MP \implies conjunto $i \bmod 2^c$ de caché.

Ejemplo: Si $c = 6$, $b = 4$, la dirección se divide en una etiqueta de $m - (c + b) = 8$ bits, que identifica el bloque en la caché. El acceso se realiza en dos fases: primero se selecciona el conjunto, y luego se realiza una búsqueda asociativa dentro del conjunto. En este caso se pretende reducir el coste de la totalmente asociativa manteniendo un rendimiento similar, es por ello que es la técnica más utilizada. Los 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.

5.1.4 Política de reemplazo

Si se produce un fallo en la lectura hay que traer un nuevo bloque, en caso de que la caché esté llena, deberemos decidir cual de los bloques quitamos, este problema no existe para correspondencia directa, sin embargo, para el resto de organizaciones se tienen los siguientes algoritmos de reemplazo:

- **FIFO:** (First in First out) De todos los bloques existentes en la caché se reemplaza el primero que se introdujo.
- **LRU:** (Least Recently Used) De todos los bloques existentes en la caché se reemplaza el que se ha usado menos recientemente.
- **RAND:** Se escoje al azar.

El más usado es el LRU, sin embargo, notemos que para tener información sobre cual es el menos usado hace falta memoria adicional.

5.1.5 Métricas para prestaciones de caché

Para medir el rendimiento de una caché, se usan las siguientes métricas:

- **Miss rate (Tasa de fallo):** La fracción de referencias a memoria que resultan en un fallo de caché durante la ejecución de un programa, se calcula como $\frac{\text{fallos}}{\text{referencias}}$. Notemos que la tasa de acierto (hit rate) es $1 - \text{miss rate}$.

- **Hit time (Tiempo en acierto):** Es el tiempo necesario para entregar una línea de caché al procesador, incluye el tiempo para determinar si la línea está en caché. 4 ciclos para L1 10 para L2.
- **Miss penalty (Penalización por fallo):** Tiempo adicional necesario por causa de un fallo. Entre 50 y 200 ciclos de reloj.

La optimización del coste y rendimiento de las memorias caché es un ejercicio delicado que requiere de notables simulaciones y benchmarks. Notemos que hay una enorme diferencia entre un acierto y un fallo, alrededor de incluso 100 veces más lento si consideramos L1 y memoria principal. Destacamos que usamos la tasa de fallo en lugar de la de acierto ya que puede resultar confuso que una tasa de acierto del 99% sea el doble de buena que del 97%.

5.1.6 Jerarquía de memoria

Usaremos i para denotar el nivel dentro de la jerarquía, en estas condiciones, podemos caracterizar los dispositivos de almacenamiento por su:

- Tiempo de acceso (t_i): Tiempo desde que se inicia una lectura hasta que llega la palabra deseada.
- Tamaño de la memoria (s_i): Número de palabras, bytes, sectores, etc que se pueden almacenar en el dispositivo de memoria.
- Coste por bit o por byte (c_i).
- Ancho de banda (b_i): Velocidad a la que se transfiere la información desde un dispositivo.
- Unidad de transferencia (x_i): Tamaño de la unidad de información que se transfiere entre el nivel i y el $i + 1$.

Con las anteriores definiciones, podemos establecer los siguientes órdenes:

$$t_i < t_{i+1} \quad s_i < s_{i+1} \quad c_i > c_{i+1} \quad b_i > b_{i+1} \quad x_i < x_{i+1}$$

Además se verifica lo que llamaremos **Propiedad de inclusión:**

- $M_1 \subset M_2 \subset M_3 \subset \dots \subset M_n$.
- Si una palabra se encuentra en M_i entonces en los niveles siguientes podemos encontrar copias de esa palabra.
- Si una palabra está almacenada en M_i puede no estarlo en M_{i-1} , en ese caso, tampoco puede estar en M_{i-2} , etc.

5.1.7 Modelo de evaluación de la jerarquía

C.K. Chow propuso en 1974 un modelo que asume que las caches son inclusivas $\sum_{i=1}^j a_i = A_j$ y que el compilador no usa registros ($A_0 = 0$). En este modelo se propone el concepto de *Hit ratio* (tasa de aciertos (A_i)), este hace referencia al 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 la capacidad del nivel, la granularidad de la transferencia de información y de la estrategia de administración de memoria tomada. Análogamente, se define el *Miss ratio* (tasa de fallos (F_i)), como el porcentaje de información buscada que no está en el nivel i , se tiene que $F_i = 1 - A_i$.

Para cada nivel i , la tasa de aciertos específica de ese nivel (a_i) se define como la frecuencia de accesos de

primer éxito al nivel i . La probabilidad de acceder con éxito a una información en el nivel i y que esa información no se encuentre en los niveles superiores, se verifica $\sum_{i=1}^n a_i = 1$. Como ya mencionamos, el objetivo al diseñar una memoria con n niveles es obtener un rendimiento cercano al nivel más rápido, pero con un coste similar al nivel más lento. Para medir el rendimiento de la jerarquía, se utilizan varias métricas:

- El **tiempo medio** de acceso ponderado, que se calcula como:

$$\bar{T} = \sum_{i=1}^n a_i T_i \quad \text{ó} \quad \bar{T} = \sum_{i=1}^n F_{i-1} t_i$$

Donde se usa el tiempo T_i que es el tiempo de acceso efectivo al nivel i -ésimo:

$$T_i = \sum_{j=1}^i t_j$$

Para cada nivel, donde t_j es el tiempo de acceso específico del nivel j .

- El **Coste por bit** que se calcula como:

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

Donde el numerador representa el coste total, el denominador el tamaño total y c_0 el coste de interconexión.

Ejemplo 5.3: Veamos un ejemplo sobre un modelo con dos niveles (caché y memoria principal):

- $t_1 = t_c$ Tiempo de acceso a la memoria caché.
- $a_1 = A_1 = A$ Tasa de aciertos en la caché.
- $t_2 = t_m$ Tiempo de acceso a la memoria principal.
- Entonces el tiempo medio de acceso \bar{T} :

$$\bar{T} = A t_c + (1 - A)(t_c + t_m)$$

Ya que si hay un acierto en la caché, el tiempo de acceso es t_c , si no, el tiempo de acceso es $t_c + t_m$ porque se accede a la caché y luego a la memoria principal.

- Definiendo γ como la razón entre los tiempos de acceso a la MP y a la caché:

$$\gamma = \frac{t_m}{t_c}$$

En general la eficiencia de un sistema que utiliza memoria caché es:

$$E = \frac{t_c}{\bar{T}} = \frac{t_c}{A t_c + (1 - A)(t_c + t_m)} = \frac{1}{A + (1 - A)(\gamma + 1)} = \frac{1}{1 + \gamma(1 - A)}$$

- La eficiencia resulta máxima cuando $A = 1$.
- La eficiencia es mayor cuanto menor sea γ y mayor sea A .

Ejemplo 5.4: Consideramos un sistema cuyo tiempo de acceso a la caché es de 15ns, a la memoria principal de 100ns y la tasa de aciertos en la caché es del 90%.

$$\bar{T} = 0.9 \cdot 15 + 0.1 \cdot (15 + 100) = 25ns \quad E = \frac{15}{25} = 0.6$$

5.1.8 Caché separada o unificada

Como se pudo ver en la jerarquía de un Intel Core i7, la caché L1 es separada, es decir, hay una caché para instrucciones y otra para datos.

- **Caché de datos:** La localidad de los datos no es tan buena como la de las instrucciones por lo que es menor eficiente, además es más compleja ya que se puede escribir, modificando los datos. En muchos sistemas, no hay caché de datos.
- **Caché instrucciones:** Es fácil que bucles y pequeñas subrutinas entren totalmente en caché, permitiendo su ejecución sin necesidad de acceder a memoria principal, además es de solo lectura por lo que es más sencilla y rápida.

De esta manera, podemos emitir direcciones de instrucción y datos a la vez, doblando el ancho de banda entre caché y procesador, además, se puede optimizar cada caché por separado. Cuando comparamos la frecuencia de fallos de una caché de instrucciones, de datos y unificada, se tiene que la de instrucciones es la que menos fallos tiene y por consiguiente, la unificada es la siguiente.

5.1.9 Impacto del tamaño

Por un lado, una caché mas grande tenderá a incrementar la tasa de acierto, sin embargo, es más complejo hacer que una memoria más grande vaya rapido, por lo que el tiempo de acceso será mayor. En cuanto al tamaño de bloque, un bloque más grande incrementará la tasa de acierto ya que se aprovecha mejor la localidad espacial, sin embargo, si el bloque es muy grande, se perderá la localidad temporal.

5.1.10 La montaña de memoria

El ratio con el que un programa lee datos de la memoria del sistema se llama rendimiento de lectura (bandwidth), se mide en bytes por segundo, normalmente se expresa en MB/s. Si escribiesemos un programa que realizase una secuencia de lecturas en un bucle, podríamos obtener información acerca del rendimiento del sistema en esa secuencia particular. Presentamos el siguiente código:

```
/* Declaración del array global que vamos a recorrer */
long data[MAXELEMS];

/*
 * test - Itera sobre los primeros "elems" elementos del array "data"
 * con un salto (stride) de "stride", utilizando desenrollado de bucle 4 x 4.
 */
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;
    long 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];
    }
}
```

```

}

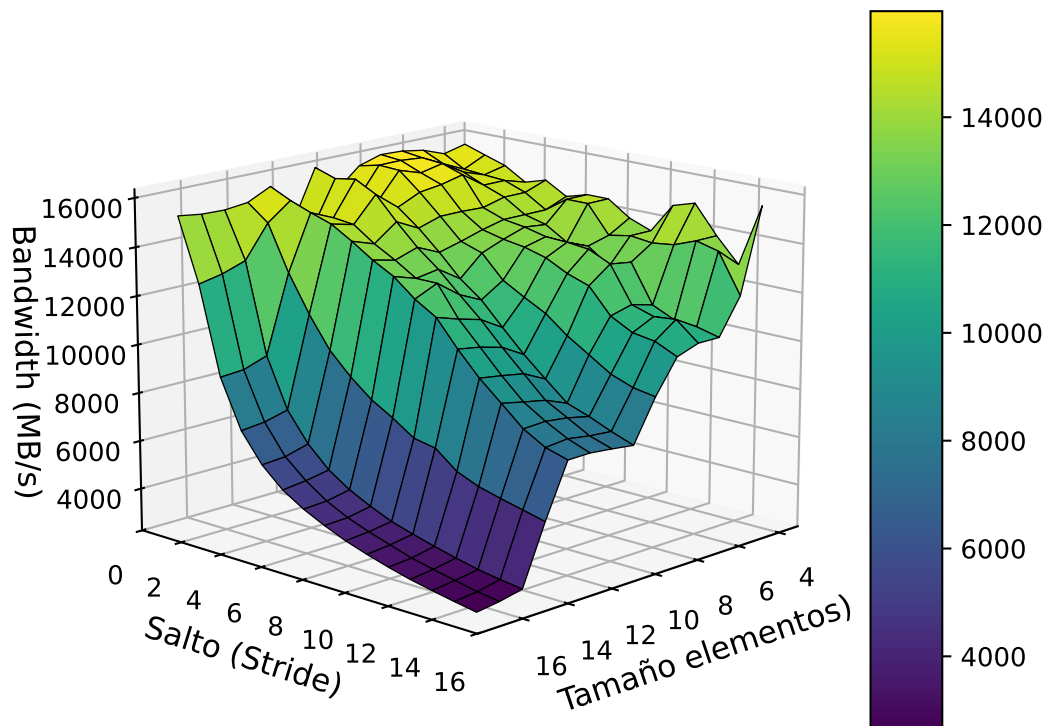
/* Procesar cualquier elemento restante */
for (; i < length; i++) {
    acc0 = acc0 + data[i];
}

return ((acc0 + acc1) + (acc2 + acc3));
}

```

La función test genera la secuencia de lectura deseada, el bucle está desenrollado en 4 iteraciones, para incrementar el paralelismo. Podríamos programar una función que corra el código anterior (con un calentamiento para obtener resultados precisos) y obtendríamos el ancho de banda. Usando que tenemos como parámetros el tamaño de los elementos y el salto, podemos obtener el ancho de banda en función de estos dos parámetros, obteniendo una gráfica en 3D cuyo aspecto es similar a una montaña, de ahí el nombre.

Montaña de memoria



En este caso, se ha generado la gráfica para un sistema con una CPU Ryzen 5 5600H, como se puede ver, existen varias crestas de localidad temporal mientras que encontramos curvas para la localidad espacial.

5.2 Código amigable con caché

Ya hemos introducido el término de localidad anteriormente, ahora que hemos visto la caché con mayor detenimiento, seremos más precisos, como dijimos, los programas con mejor localidad tenderán a provocar menos misses, lo que resulta en una mayor eficiencia, es por ello, que como programadores, deberemos tratar de escribir código *caché friendly*, con este fin, introducimos los principios que seguiremos:

- **Hacer que el caso común común vaya más rápido:** Los programas normalmente usan la mayoría del tiempo en algunas funciones esenciales, estas funciones a su vez, usan el tiempo en bucles, nos centraremos en optimizar estos bucles.

- **Minimar los fallos en los bucles internos:** Buscaremos referencias repetidas a variables para buscar localidad temporal y patrones de referencia de salto 1 para buscar localidad espacial.

Para ver cómo funciona esto, veamos un ejemplo de código:

Ejemplo 5.5: Calcularemos el centro de masas de 1024 algas en un array de 32x32 posiciones, donde cada una de ellas es un struct posición con dos enteros para indicar la coordenada x e y. El resto de variables auxiliares para calcular esto se guardarán en registros. Para este caso, consideramos una caché cuyas características son:

- **Tamaño:** 2KB
- **Tamaño bloque:** 32B
- **Política:** correspondencia directa

De estas características se obtiene que:

- Cada struct ocupa 8B
- Cada bloque tiene 4 structs
- Cada fila son 32 structs de 8B, es decir 256B
- Cada caché contiene $2^{11}/2^5 = 64$ bloques, es decir, 8 filas.

Presentamos cuatro versiones para resolver el código anterior:

```
for (j=0; j<32; j++)
  for (i=0; i<32; i++)
    total_x += grid[i][j].x;
for (j=0; j<32; j++) {
  for (i=0; i<32; i++) {
    total_y += grid[i][j].y;
```

Figure 5.1: Versión 1

```
for (i=0; i<32; i++)
  for (j=0; j<32; j++)
    total_x += grid[i][j].x;
for (i=0; i<32; i++) {
  for (j=0; j<32; j++) {
    total_y += grid[i][j].y;
```

Figure 5.2: Versión 2

```
for (j=0; j<32; j++)
  for (i=0; i<32; i++)
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
```

Figure 5.3: Versión 3

```
for (i=0; i<32; i++)
  for (j=0; j<32; j++)
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
```

Figure 5.4: Versión 4

Para cada una de las versiones:

- **Versión 1:** Se desaprovecha la localidad espacial entre x e y. Además se accede a cada elemento en saltos de 32 elementos luego también se desaprovecha la localidad espacial para elementos del array (stride 1). Como resultado, se produce un promedio de 1 fallo por acceso.
- **Versión 2:** Se desaprovecha la localidad espacial entre x e y. En este caso si se aprovecha el stride 1 para cada elemento. Como resultado, se produce un promedio de 0.25 fallos por iteración.

- **Versión 3:** Se aprovecha la localidad espacial entre x e y . Se accede a cada elemento en saltos de 32 elementos, por lo que se desaprovecha la localidad espacial para elementos del array. Como resultado, se produce un promedio de 0.5 fallo por acceso.
- **Versión 4:** Se aprovecha la localidad espacial entre x e y , además se accede secuencialmente a cada elemento. Como resultado, se produce un promedio de 0.125 fallos por acceso.

En pruebas realizadas en la misma CPU de la montaña se ha obtenido una diferencia en rendimiento de 3x entre la versión 1 y la 4, esta diferencia sería incluso más grande si el salto fuese mayor a 32. ■

Relaciones de problemas

6.1 Tema 5-6

Ejercicio 6.1.1: Suponga una jerarquía de memoria de dos niveles, M_1 y M_2 , con capacidades s_1 y s_2 bits, tiempo de acceso a los circuitos de cada nivel t_1 y t_2 , y coste por bit c_1 y c_2 , respectivamente. Obtenga:

- a) El costo por bit de toda la memoria.

Viene dado por:

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

En este caso, y suponiendo que el coste de interconexión es despreciable, se tiene que:

$$c(2) = \frac{c_1 \cdot s_1 + c_2 \cdot s_2}{s_1 + s_2}$$

- b) El tiempo medio de acceso.

Dado que no se proporciona la frecuencia de acceso con éxito a cada nivel, calculamos el tiempo medio de acceso como:

$$\bar{T} = t_1 \cdot A_1 + (t_1 + t_2) \cdot (1 - A_1)$$

- c) La eficacia de la jerarquía de memoria en términos de la razón de velocidad del nivel 2 respecto al 1.

En este caso:

$$E = \frac{t_1}{\bar{T}}$$

Ejercicio 6.1.2: Considere una jerarquía de memoria de dos niveles, M_1 (más cercano al procesador) y M_2 , con tiempos de acceso t_1 y t_2 , costes por byte c_1 y c_2 , y tamaños s_1 y s_2 , respectivamente. La razón de aciertos de M_1 es $A_1 = 0.9$.

- a) Escriba una fórmula mostrando el coste total C_{tot} del sistema de memoria, sin tener en cuenta el coste de conexión de los dos niveles de la jerarquía.

Se puede calcular como:

$$C_{tot} = \frac{c_1 \cdot s_1 + c_2 \cdot s_2}{s_1 + s_2}$$

- b) Escriba una fórmula que modele el tiempo de acceso efectivo T_{ef} del sistema de memoria, teniendo en cuenta que el acceso a M_1 y M_2 no se puede llevar a cabo simultáneamente.

Viene modelado por la fórmula:

$$T_{ef} = 0.9 \cdot t_1 + 0.1 \cdot (t_1 + t_2)$$

- c) Suponga que $t_1 = 20 \text{ ns}$, t_2 es desconocido, $s_1 = 512 \text{ KB}$, s_2 es desconocido, $c_1 = 0.00015 \text{ €/byte}$, $c_2 = 0.0000006 \text{ €/byte}$.

- i) ¿Cuántos GB de M_2 ($s_2 = ?$) se pueden adquirir sin exceder un presupuesto total aproximado de unos 400 €, si se desprecia el coste de la conexión de los dos niveles de la jerarquía?

Con un presupuesto de 400 €, se tiene que:

$$C_{tot} = c_1 \cdot s_1 + c_2 \cdot s_2 \Rightarrow 400 = 0.00015 \cdot 512 \cdot 10^3 B + 6 \cdot 10^{-7} \cdot s_2 \cdot 10^9 \Rightarrow s_2 = 0.539$$

- ii) ¿De qué velocidad hay que comprar la memoria M_2 ($t_2 = ?$) para conseguir un tiempo de acceso medio $T_{ef} = 30 \text{ ns}$ en el sistema de memoria completo bajo la suposición de tasa de aciertos anterior?

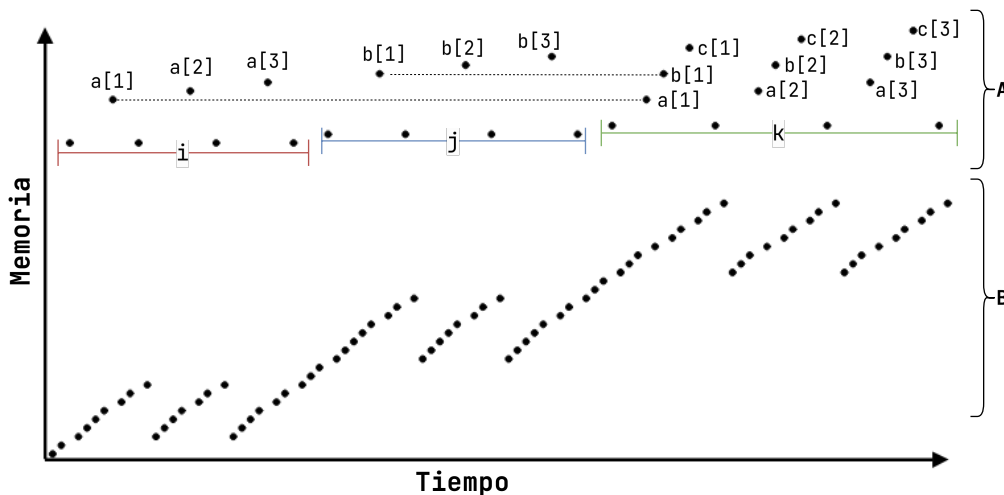
Usando la fórmula de T_{ef} , se tiene que:

$$30 \text{ ns} = 0.9 \cdot 20 \text{ ns} + 0.1 \cdot (20 \text{ ns} + t_2) \Rightarrow t_2 = 100 \text{ ns}$$

Ejercicio 6.1.3: Teniendo en cuenta que la gráfica de la Figura 1 pretende estar relacionada con el pseudocódigo siguiente, etiquete los ejes de coordenadas, rodee con un círculo los puntos que le parezcan relacionados, etiquételos según la nomenclatura del pseudocódigo e indicando qué tipo de localidad se para cada caso, y deduzca el valor de N .

- Para $i = 1 \dots N$, $a[i] = i \times i$
- Para $j = 1 \dots N$, $b[j] = j \times j \times j$
- Para $k = 1 \dots N$, $c[k] = a[k] + b[k]$

Veamos directamente la solución, (el problema original viene únicamente con los puntos)



En este caso, podemos ver que el eje de ordenadas corresponde con las direcciones de memoria de cada elemento, en el eje de abscisas se representa el tiempo. La parte señalada como A corresponde con los accesos a los datos, para saberlo, notemos que los accesos a lo largo del tiempo a los iteradores tienen que estar en la misma posición, de ahí se deducen i,j,k. Para distinguir A de B además notamos que a la derecha se producen accesos a posiciones anteriores muy seguidos, lo que se realiza en el pseudocódigo para c, a partir de esto, tanto a como b son fáciles de ver ya que en memoria están muy juntos. Viendo el número de iteraciones para cada uno de ellos se deduce que $N = 3$. En este caso se aprovecha la localidad temporal siempre y la espacial solo para a y b, que se acceden secuencialmente.

Respecto a la zona etiquetada como B, se puede deducir que se trata de las instrucciones, además podemos decir que tiene localidad espacial y temporal en la mayoría de su duración.

Ejercicio 6.1.4: Un computador con un bus de datos de 32 bits usa circuitos de memoria RAM dinámica de $1M \times 4$.

- a) ¿Cuál es la memoria más pequeña (en bytes) que puede tener ese computador?

Como se tienen 4 bytes por dirección y como mínimo hay 32 direcciones ya que se usan 32 bits para el bus de datos, la más pequeña sería de 128B.

- b) Dibuje un esquema de la misma, detallando las líneas de direcciones y de datos.

Ejercicio 6.1.5: Una placa madre de un IBM PC basada en el microprocesador Intel 8088 dispone de 256 KB de DRAM con 1 bit de paridad, constituida por circuitos integrados 4164 (Figura 2). Las señales que conectan esta memoria con los buses y otros circuitos (circuitaría de refresco de memoria y de generación/comprobación de paridad) se muestran en la Figura 3 (PAR_IN y PAR_OUT son las señales de paridad).

- a) Dibuje el sistema de memoria y su conexionado con las señales de la Figura 3. No incluya ninguna circuitaría de refresco.
- b) Especifique en el dibujo del apartado (a) o en una tabla las direcciones de memoria asociadas a cada chip o grupo de chips. Indique asimismo en qué chips y en qué posición dentro de esos chips se encuentra almacenado el byte de memoria cuya dirección es 20001h.

Ejercicio 6.1.6: Una placa madre de un PC basada en el microprocesador Intel 8088 disponía de 512 KB de DRAM más bit de paridad, constituida por circuitos integrados 41256 (Figura 4). Las señales que conectaban esta memoria con los buses y otros circuitos (circuitaría de refresco de memoria y de generación/comprobación de paridad) se muestran en la Figura 5 (PAR_IN y PAR_OUT son las señales de paridad).

- a) Dibuje el sistema de memoria completo (todos los chips) y su conexionado con las señales de la Figura 5. No hay que incluir la circuitaría de refresco.
- b) Especifique en el dibujo el rango de direcciones de memoria asociado a cada chip.

Ejercicio 6.1.7: Suponga que dispone de circuitos integrados de memoria SRAM de $32K \times 8$ (Figura 6) y desea construir con ellos una memoria de 256 K palabras de 32 bits, organizada con entrelazado de orden inferior con bancos de 64 K palabras.

- a) Dibuje el esquema del sistema de memoria, detallando las líneas de dirección.
- b) Indique en el dibujo en qué chips y en qué posición dentro de esos chips se encuentra almacenada la palabra cuya dirección es 20001h (el direccionamiento de memoria se realiza a nivel de palabras de 32 bits).

Ejercicio 6.1.8: La placa madre de un procesador de 32 bits (486) dispone de 4 MB de memoria DRAM distribuida en módulos SIMM de 30 contactos sin paridad de $1M \times 8$. Cada SIMM contiene 8 chips de DRAM. Aparte de 9 contactos para alimentación, tierra y no conectados, cada SIMM tiene los siguientes contactos:

- A0-A9: Entradas de dirección.
- RAS: Row Address Strobe.
- CAS: Column Address Strobe.
- WE: Write Enable.
- DQ1-DQ8: Entradas/salidas de datos.

- a) ¿Qué tamaño tiene cada chip de DRAM?
- b) Dibuje un esquema de uno de los SIMM con las conexiones entre sus contactos y las patillas de los chips de DRAM.
- c) Dibuje un esquema con todos los SIMM (sin dibujar el interior de cada uno) y su conexión con los buses de direcciones y datos del sistema.

Ejercicio 6.1.9: Un computador dispone de 16 MB de memoria principal entrelazada de orden inferior y acceso simultáneo (Tipo S), constituida por módulos de 1 M Byte. Suponga que el procesador desea acceder a los bytes cuyas direcciones son:

- AB0010h, AB0021h, 0010AAh, 227AAAh, 0101AAh, 01016Ah, 010163h

- a) ¿En qué módulo se encuentra cada uno de esos bytes?

En primer lugar, comenzamos notando que al tratarse de entrelazado inferior, los bits inferiores seleccionan el módulo de memoria, además el acceso simultáneo indica que se hace *latch* a la salida de datos, es decir, se muestra la misma dirección a todos los módulos en cada acceso. Dicho esto, cada módulo es de $1MB = 2^{20}B$ y hay 16 módulos, cada dirección tiene 24 bits, por lo que, restando los 4 bits para el módulo tenemos 20 bits para la dirección y 4 para el módulo:

- AB0010h: Módulo 0
- AB0021h: Módulo 1
- 0010AAh: Módulo 10
- 227AAAh: Módulo 10
- 0101AAh: Módulo 10
- 01016Ah: Módulo 10
- 010163h: Módulo 3

- b) ¿Cuántos accesos simultáneos a memoria se necesitan como mínimo?

6 accesos simultáneos ya que únicamente los dos últimos accesos coinciden en la dirección.

Ejercicio 6.1.10: Se tiene un computador con una memoria principal entrelazada con esquema de entrelazado de orden inferior con latch en las entradas (acceso C) de 256 K palabras dividida en 4 módulos.

- a) ¿Cuántos bytes hay en cada módulo?

- b) ¿Cuántos bits necesita cada módulo para acceder a 128 palabras?

Ejercicio 6.1.11: En un computador que dispone de caché, las instrucciones necesitan 8.5 ciclos de reloj para ejecutarse en caso de que no haya falta. Si la tasa de faltas de la caché es del 11%, cada falta da lugar a un incremento de tiempo de 6 ciclos de reloj, y por término medio cada instrucción necesita 3 accesos a memoria. ¿Cuál debe ser el número medio de ciclos de una instrucción (*CPI*) en el computador sin caché para que sea beneficioso utilizarla?

Ejercicio 6.1.12: Suponga que el tiempo de ejecución de un programa es directamente proporcional al tiempo de acceso a instrucciones, y que el acceso a una instrucción en la caché es ocho veces más rápido que el acceso a una instrucción en la memoria principal. Suponga que una instrucción pedida por el procesador se encuentra en la caché con una probabilidad de 0.9, y suponga también que si una instrucción no se encuentra en la caché primero debe ser captada desde la memoria principal a la caché y, a continuación, captada de la caché para ser ejecutada.

- a) Calcule la eficiencia como la relación entre el tiempo de ejecución de un programa sin la caché y el tiempo de ejecución con la caché.
- b) Siempre que se dobla el tamaño de la caché, la probabilidad de no encontrar en ella una instrucción buscada se reduce a la mitad. Dibuje una gráfica que represente la eficiencia, tal como se define en el apartado a), en función del tamaño de la caché, indicando cuál es la eficiencia mínima (tamaño 0) y máxima (tamaño $\rightarrow \infty$).

Ejercicio 6.1.13: El tiempo de acceso a la memoria caché de un sistema es de 50 ns, y el tiempo de acceso a la memoria principal es de 500 ns. Se estima que el 80% de las peticiones de acceso a la memoria son para lectura y el resto para escritura. La razón de aciertos es de 0.9.

- a) Determinar el tiempo de acceso promedio considerando sólo los ciclos de lectura.
- b) Determinar el tiempo de acceso promedio considerando también los ciclos de escritura.

Ejercicio 6.1.14: Un computador con una memoria principal de 1 M palabra dispone de una memoria caché de correspondencia directa de 4 K palabras con marcos de bloque de 16 palabras.

- a) Indique el número de faltas de caché que se producen si el procesador genera la secuencia de accesos a memoria:

ABC13h, CDC14h, ABC1Fh, AB305h, CAC13h, CDC1Ah, CA00Fh, ABC10h

- b) Para esa misma secuencia, indique los marcos de bloque de caché en los que se carga cada posición de memoria principal a la que se pretende acceder.

Ejercicio 6.1.15: Un computador direccionable por bytes tiene una pequeña caché de datos capaz de almacenar 8 palabras de 32 bits. Cada bloque de caché consiste en una palabra de 32 bits. Al ejecutar un determinado programa, el procesador lee datos de la siguiente secuencia de direcciones (en hexadecimal):

200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4

Este patrón se repite cuatro veces en total.

- a) Muestre los contenidos de la caché al final de cada pasada a ese bucle si se usa correspondencia directa. Calcule la tasa de aciertos para este ejemplo. Asuma que la caché está inicialmente vacía.
- b) Repita la parte a) para una correspondencia completamente asociativa que usa el algoritmo de reemplazo LRU.

- c) Repita la parte a) para una caché asociativa por conjuntos de cuatro vías.

Ejercicio 6.1.16: Un computador tiene una caché de 128 bytes. Usa correspondencia asociativa por conjuntos, de cuatro vías, con 8 bytes en cada bloque. El tamaño de la dirección física es 32 bits, y la unidad mínima direccionable es 1 byte.

- a) Dibuje un diagrama que muestre la organización de la caché e indique cómo se relaciona una dirección física con la caché (campos).
- b) ¿En qué marcos de bloque de la caché puede residir el byte de memoria principal cuya dirección es 000010AFh?
- c) Si las direcciones 000010AFh y FFFF7AXYh pueden asignarse simultáneamente al mismo conjunto de caché, ¿cuántos posibles valores distintos puede tener XY en la segunda dirección?

Ejercicio 6.1.17: Suponga una memoria caché de 4 K palabras asociativa por conjuntos, con marcos de bloque de 64 palabras y 8 conjuntos, y una memoria principal de 8 M palabras.

- a) ¿En qué posición de caché se situaría la palabra de memoria principal cuya dirección es 3A0A39h?
- b) ¿Qué direcciones de memoria principal no pueden encontrarse en caché al mismo tiempo que esa dirección?

Ejercicio 6.1.18: Un sistema de cómputo tiene una memoria principal de 32 K palabras de 16 bits. También tiene una memoria caché de 4 K palabras organizadas de forma asociativa por conjuntos con 4 bloques por conjunto y 64 palabras por bloque.

- a) Calcule el número de bits de cada uno de los campos: Etiqueta, Conjunto y Palabra.
- b) Si el tiempo de acceso a la memoria principal es 10 veces mayor que el de la memoria caché, y la razón de aciertos de la caché es de 0.9, determinar la eficiencia de acceso (razón entre el tiempo de acceso a la memoria caché y el tiempo medio de acceso).

Ejercicio 6.1.19: Un ordenador dispone de 32 KB de memoria principal direccionable por bytes y una memoria caché completamente asociativa de 4 KB. El tamaño del bloque de la memoria caché es de 8 palabras de 32 bits. El tiempo de acceso a la memoria principal es 10 veces mayor que el de la memoria caché.

- a) ¿Cuántos comparadores hardware se necesitan?
- b) ¿Cuál es el tamaño del campo identificador?
- c) Si se utiliza el esquema de sustitución directa, ¿cuál sería el tamaño del campo identificador?
- d) Suponer que la eficiencia de acceso se define como la razón entre el tiempo de acceso con memoria caché y el tiempo de acceso sin memoria caché. Determinar la eficiencia de acceso suponiendo que la razón de aciertos de la memoria caché es 0.9.
- e) Si el tiempo de acceso a la memoria caché es de 200 ns, ¿cuál será la razón de aciertos necesaria para lograr un tiempo de acceso de 500 ns?

Ejercicio 6.1.20: Los parámetros que definen la memoria de un computador son los siguientes:

- Tamaño de la memoria principal: 32 K palabras.
- Tamaño de la memoria caché: 4 K palabras.

- Tamaño de bloque: 64 palabras.

Determine el tamaño de cada campo de una dirección de memoria y explique brevemente cómo se obtiene, para las siguientes políticas de colocación:

- a) Totalmente asociativa.

En este caso se tienen m bits para la dirección de la memoria principal y el resto para los bytes, como en este caso hay 2^6 palabras se necesitan 6 bits para el campo b y como para la memoria principal se necesitan 2^{15} , es decir 15 bits, el campo de etiqueta tiene 9 bits.

- b) Por correspondencia directa.

Como antes, tenemos 6 bits para el campo b y 15 bits en total, la caché ocupa 12 bits, por lo que tenemos 2^6 líneas que direccionaremos con 6 bits luego el campo de etiqueta ocupará 3 bits.

- c) Asociativa por conjuntos con 16 bloques por conjunto.

16 Bloques necesitan 4 bits, es decir, mantenemos 6 bits para el campo b , por tanto tenemos $15 - 6 - 4$ bits para el campo de etiqueta

Ejercicio 6.1.21: Los parámetros que definen la memoria de un computador son los siguientes:

- Tamaño de la memoria principal: 4 G bytes.
- Tamaño de la memoria caché: 1 MB.
- Tamaño de bloque: 256 bytes.

Determine el tamaño de cada campo de una dirección de memoria desde el punto de vista de la caché, para las siguientes políticas de colocación:

- a) Totalmente asociativa.

Como tenemos 32 bits para la dirección principal y cada bloque son 8 bits, tenemos $32 - 8 = 24$ bits para la etiqueta.

- b) Por correspondencia directa.

Sabemos que para el offset tenemos 8 bits, la caché ocupa 2^{20} bytes, luego hay 2^{12} líneas de caché y se necesitan 12 bits para direccionarlas, como resultado nos quedan $30 - 8 - 12 = 10$ bits para la etiqueta

- c) Asociativa por conjuntos con 4 bloques por conjunto.

Se utilizan 2 bits para el conjunto, además 8 bits para el offset, de donde se tienen $20 - 8 - 2 = 10$ bits para la etiqueta.

Ejercicio 6.1.22: Los parámetros que definen la memoria de un computador son los siguientes:

- Tamaño de la memoria principal: 4 GB.
- Tamaño de la memoria caché: 16 MB.
- Tamaño de bloque: 256 bytes.

Determine el tamaño de cada campo de una dirección de memoria desde el punto de vista de la caché, para las siguientes políticas de colocación:

- a) Totalmente asociativa.

Como tenemos 32 bits para la dirección principal y cada bloque son 8 bits, tenemos $32 - 8 = 24$ bits para la etiqueta.

- b) Directa.

Sabemos que para el offset tenemos 8 bits, como la caché ocupa $2^{24}B$ tenemos 16 bits para las líneas, como la memoria principal tiene direcciones de 32 bits, se tienen $32 - 8 - 16 = 8$ bits para el campo de etiqueta.

- c) Asociativa por conjuntos, de 8 vías.

Necesitamos 3 bits para las vías, además se tienen 8 bits para el bloque de datos, es decir, 13 bits para conjunto, si tomamos ahora los 32 bits de la memoria principal y le restamos el de conjunto y el de los bloques de datos, nos quedan 11 bits para la etiqueta.

- d) Por sectores con 16 bloques por sector.

Ejercicio 6.1.23: Una memoria caché asociativa por conjuntos consiste en un total de 64 bloques divididos en conjuntos de 4 bloques. La memoria principal contiene 4096 bloques. Cada bloque contiene 128 palabras.

- a) Indicar el número de bits que hay en una dirección de memoria principal.
b) Indicar el número de bits que hay en cada uno de los campos Marca, Conjunto y Palabra.

Ejercicio 6.1.24: Un ordenador dispone de 32 K palabras de memoria principal y una memoria caché con colocación asociativa por conjuntos. El tamaño de un bloque es de 16 palabras y el campo identificador de 5 bits. Si la misma memoria caché se sustituye directamente, el campo identificador tendría una longitud de 3 bits.

- a) Determinar el número de palabras que alberga la memoria caché.
b) Determinar el número de bloques que alberga un conjunto de la memoria caché.

Ejercicio 6.1.25: Los parámetros que definen la memoria de un computador son los siguientes:

- Tamaño de la memoria principal: 8 K líneas.
- Tamaño de la memoria caché: 512 líneas.
- Tamaño de la línea: 8 palabras.

Determinar el tamaño de los distintos campos de una dirección en las siguientes condiciones:

- a) Colocación completamente asociativa.
b) Colocación directa.
c) Colocación asociativa por conjuntos con 16 líneas por conjunto.
d) Colocación por sectores con 16 líneas por sector.

Ejercicio 6.1.26: Sea un sistema de memoria con caché, con las siguientes características:

- Tamaño de la memoria principal: 4 GB.
- Tamaño de la memoria caché: 256 KB.
- Tamaño de palabra (anchura bus de datos): 32 bits.
- Correspondencia: Asociativa por conjuntos.
- Número de bloques / líneas por conjunto (número de vías): 4.
- Número de palabras por bloque / línea: 16.

Dibuje un esquema detallado de la memoria caché y su conexión con la CPU para una de las dos alternativas:

- a) Utilizar 1024 memorias asociativas de 4×16 , un decodificador de 10 a 1024, 1024 codificadores de 4 a 2, 1024 puertas OR de 4 entradas y 4096 circuitos SRAM de 64×8 .
- b) Utilizar 8 circuitos SRAM de $1\text{ K} \times 8$, 4 comparadores de dos entradas de 16 bits y 16 circuitos SRAM de $16\text{ K} \times 8$.