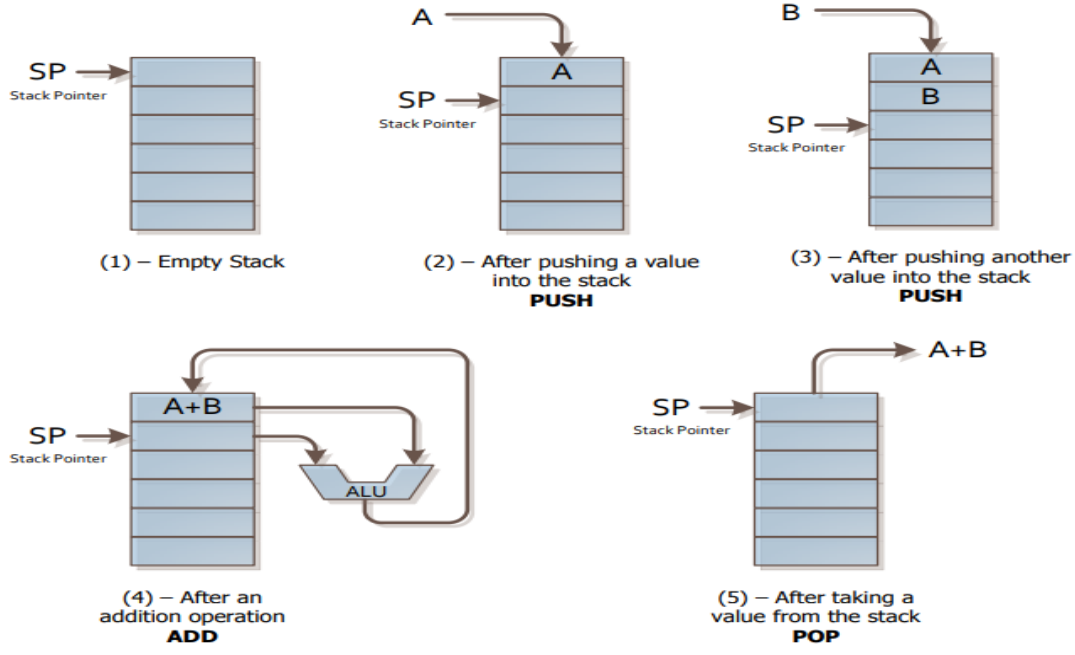
Clases de ISA

El almacenamiento interno en un procesador es la forma más básica de diferenciación. Las cuatro clasificaciones más importantes son:

- Stack.
- Acumulador.
- Registro-Memoria.
- Registro-Registro.

Stack

En la clase Stack, los operandos se encuentran en una *única pila*. Para realizar las operaciones se deben insertar los operandos (*push*) y luego extraer el resultado (*pop*). Los va extrayendo del *Top of Stack Register* (TOS).



Stack: Ejemplo

Suponga la operación: A = B + C.

```
push B; // insertar operando 1  hacia latch ALU
push C; // insertar operando 2  hacia latch ALU
add;    //suma
pop A; //extraer resultado del TOS hacia A
```

Los operandos están **implícitos** en el TOS (no se definen/nombran explícitamente).

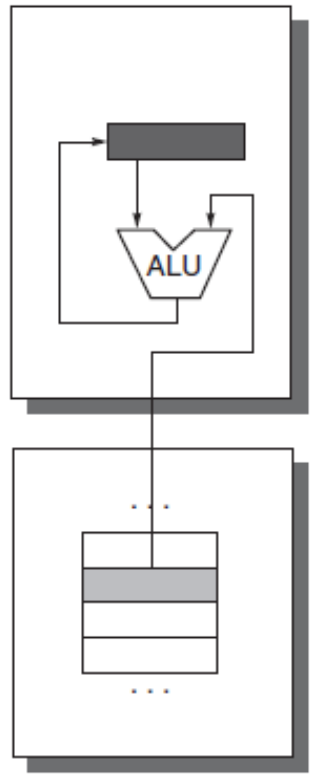
Acumulador

En las arquitecturas con acumulador, un operando se encuentra **implícito** en el acumulador y el otro se encuentra **explícito** en memoria.

Suponga la operación: A = B + C.

```
Load B;   // Carga B en el acumulador.
Add C;   // Suma C al operando implícito en memoria.
Store A; //Almacena en memoria el resultado.
```

Uno de los operandos es explícito (**C**) y el otro implícito cargado previamente desde **B**.



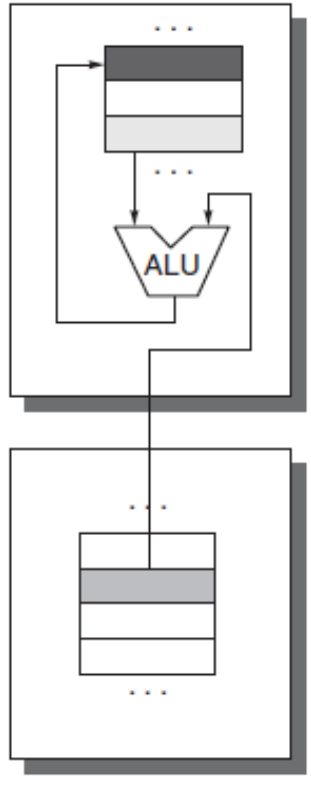
Registro (registro-memoria)

En esta arquitectura se cuenta con registros de propósito general (GPRs). Uno de los operados corresponde a un GPR y el otro proviene directamente de memoria.

Suponga la operación: A = B + C.

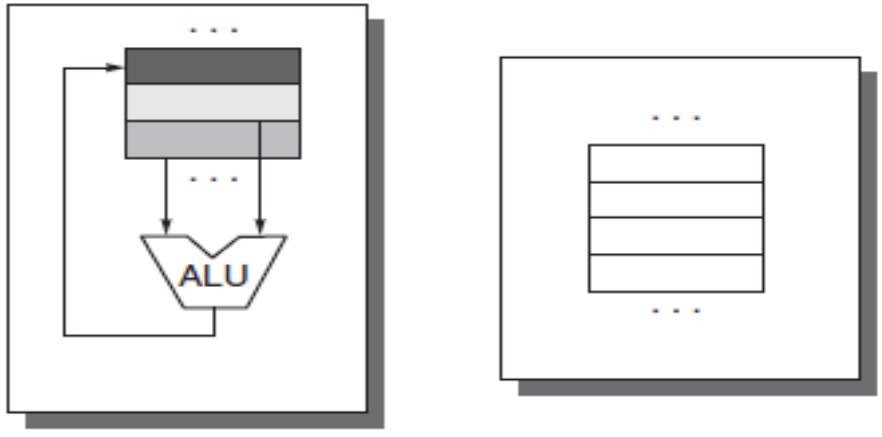
```
Load R1, B;//Cargar B en GPR R1
Add R3, R1, C;//Sumar R1 con C (dir. memoria)
Store R3, A;//Almacenar resultado en A (dir. memoria).
```

Ambos operandos son explícitos. La mayoría de arquitecturas CISC utilizan esta clase.



Registro-Registro

Una arquitectura de la clase registro-registro no realiza operaciones directamente desde memoria, sino que las realiza enteramente con operandos en registros de propósito general.



Suponga la operación: A = B + C.

```
Load R1, B;// Cargar B en GPR R1
Load R2, C;// Cargar C en GPR R2
Add R3, R1, R2;//Sumar R1 con R2 y almacena en R3.
Store R3, A;//Almacenar resultado en A (dir. memoria).
```

Ambos operandos son explícitos. La mayoría de arquitecturas CISC utilizan esta clase.

Direccionamiento de memoria

Independientemente de la arquitectura (reg-reg, reg-mem. etc) se debe tener una forma de interpretar y especificar las direcciones de memoria. Existen diferentes formas de acceder a una dirección:

Nivel de byte (8 bits)

Nivel de media palabra (16 bits)

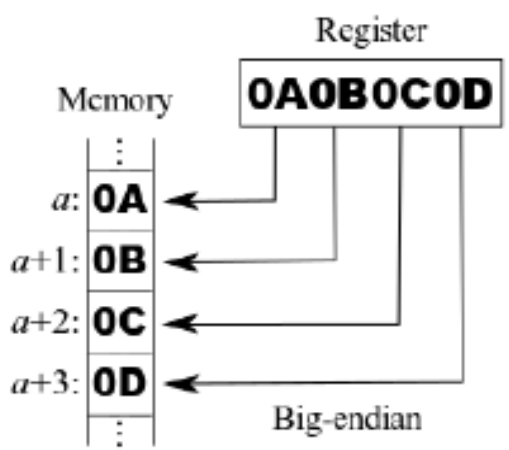
Nivel de palabra / word (32 bits)

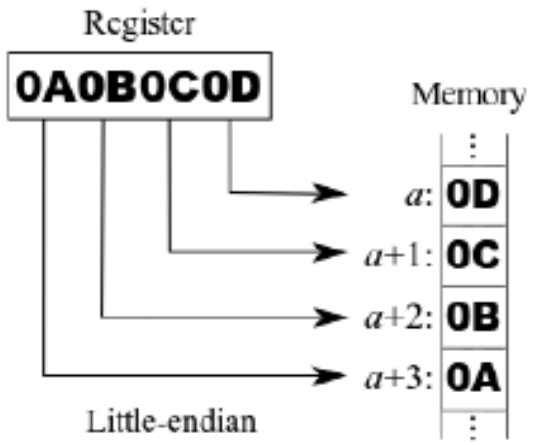
Nivel de doble palabra (64 bits)

Endianness

Existen dos formas de ordenar los bytes en una direccion memoria (*Endianness*) (también se conoce *byte ordering*):

- *Little Endian*: Ordena el byte menos significativo en la dirección más pequeña de la palabra (doble palabra).
- *Big Endian*: Ordena el byte más significativo en la direccion más pequeña de la palabra (doble palabra).



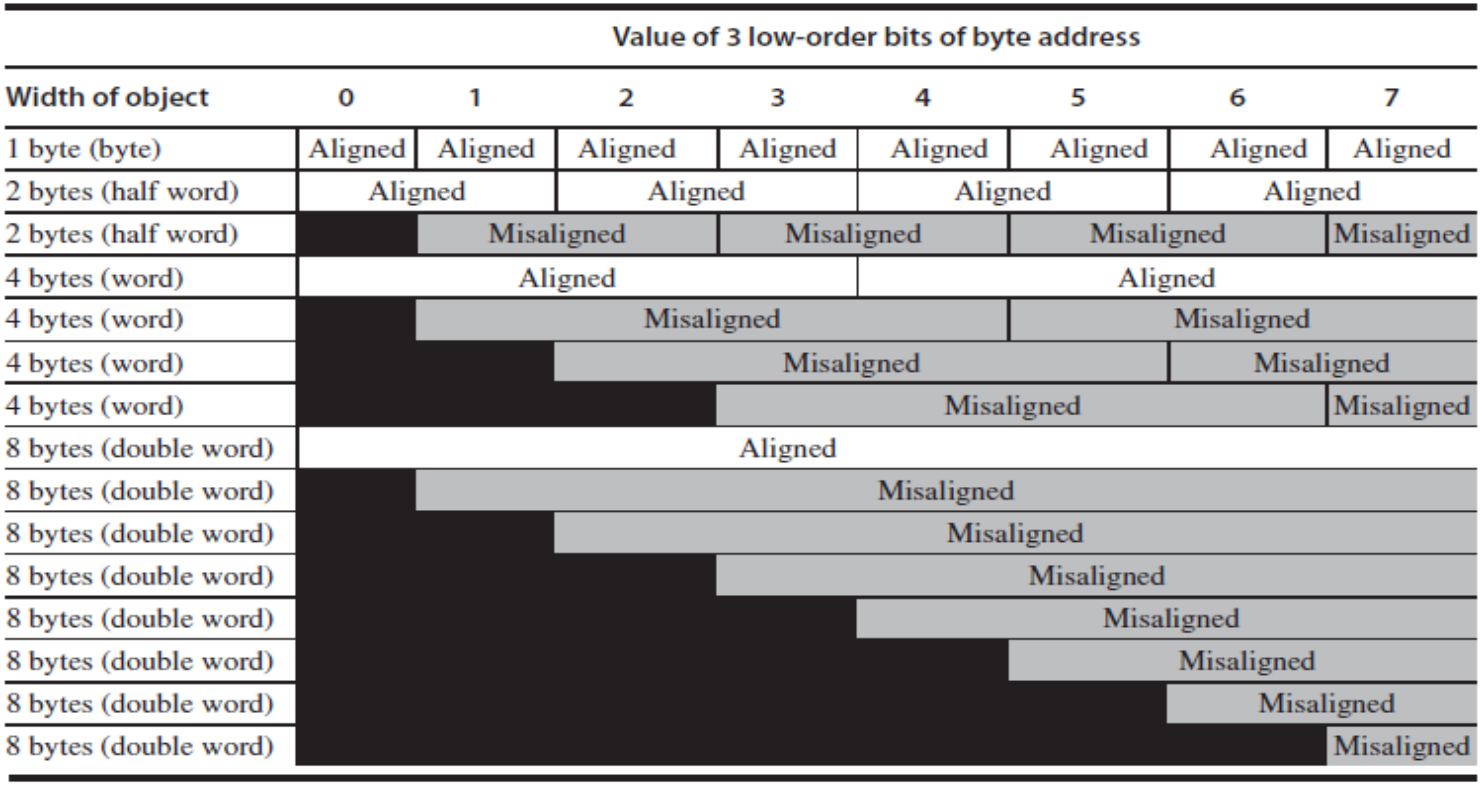


Alineamiento de memoria

Surge como una limitación de los procesadores modernos.

CPUs son mas eficientes cuando las direcciones de memoria son múltiplos del tamaño del dato (byte, KB, etc).

Algunos lenguajes de programación modernos esconden esta limitación al programador.



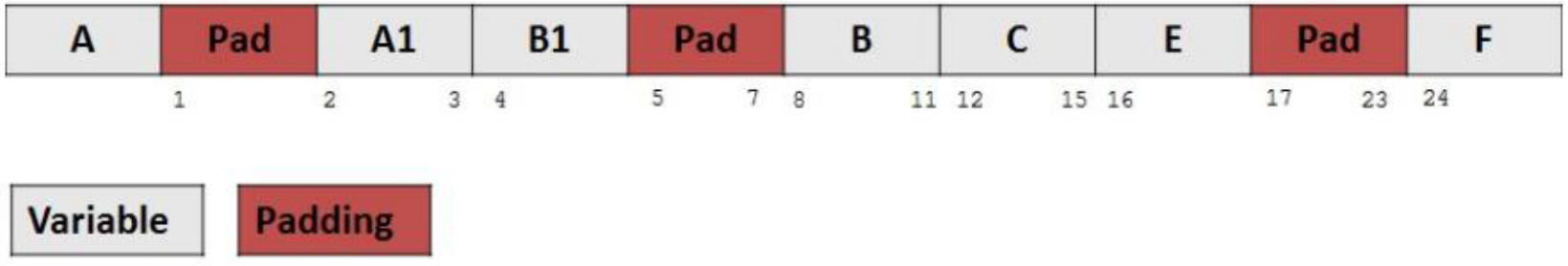
Ejemplo

Represente el alineamiento en memoria para x86 de la siguiente estructura y mejore su orden para un mejor alineamiento.

```
struct:
char a;
short a1;
char b1;
float b;
int c;
char e;
double f;
```

Data Type	32-bit (bytes)	64-bit (bytes)
char	1	1
short	2	2
int	4	4
long	8	8
float	4	4
double	8	8
long long	8	8
long double	4	16
Any pointer	4	8

Table1: typical alignment requirements for data types on 32-bit and 64-bit Linux* systems as used by the Intel® C++ Compiler



Modos de direccionamiento

Los modos de direccionamiento se refiere a la forma en que las arquitecturas **especifican** la dirección de un objecto que van a acceder.

- **Dirección efectiva**: El valor final de dirección especificado por el modo de direccionamiento.

A mayor cantidad de modos de direccionamiento → mayor complejidad (CISC).

A menor cantidad de modos de direccionamiento → menor complejidad (RISC).

Múltiples modos de direccionar datos dentro de una instrucción:

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100 + Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer p, then mode yields $*p$.
Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d.
Autodecrement	`Add R1, –(R2)`	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.