Arquitectura de Computadores I: Principios del ISA

Luis Alberto Chavarría-Zamora

ITCR

lachavarria@tec.ac.cr

10 de agosto de 2023

Contenido

Lección Anterior

¿Qué vimos?

1. ¿Qué es una arquitectura de Von Neumann?
2. ¿Qué es una arquitectura de Harvard?
3. ¿Qué diferencia hay entre una arquitectura y otra?
4. ¿Existe alguna alternativa que tenga los dos beneficios?
5. Tipos de operandos.
6. Complejidad de instrucciones (RISC vs CISC)

Arquitectura

ISA

Componentes de un set de instrucciones:

- Clase de ISA: Acceso a memoria / uso de operandos:
  - Registro-Memoria: x86.
  - Load / Store: ARM / MIPS.
- Direccionamiento de memoria: endianness, alineamiento.
- Modos de direccionamiento.
- Operaciones.
- Control de flujo.
- Decodificación.

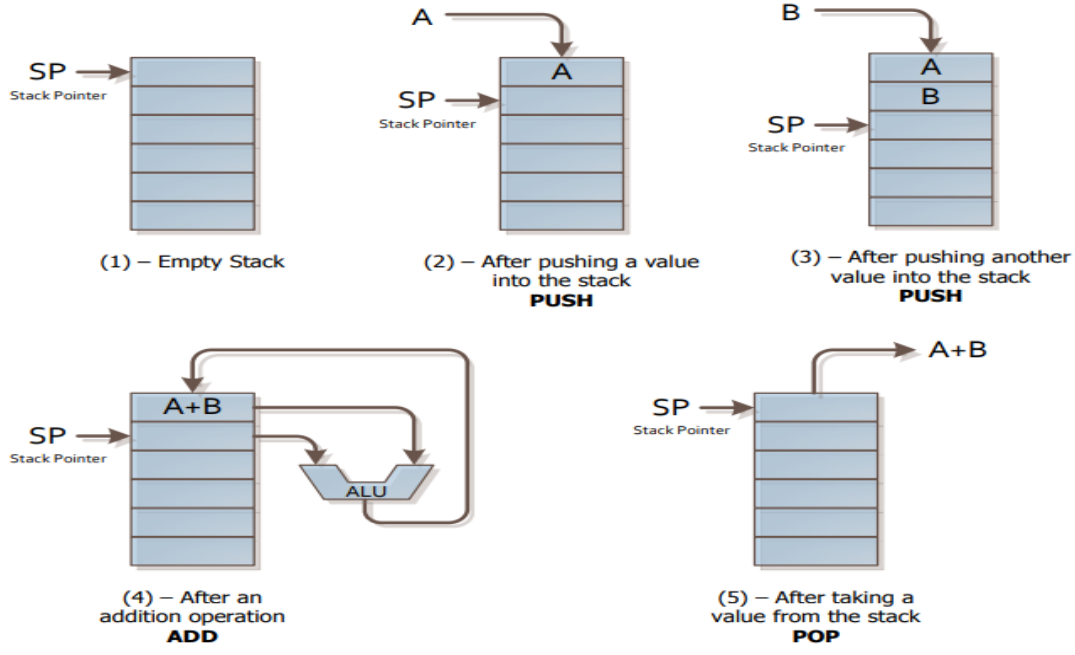
Clases de ISA

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).

Clases de ISA

Stack

Clases de ISA

Stack: Ejemplo

Suponga la operación: $A = B + C$.

Clases de ISA

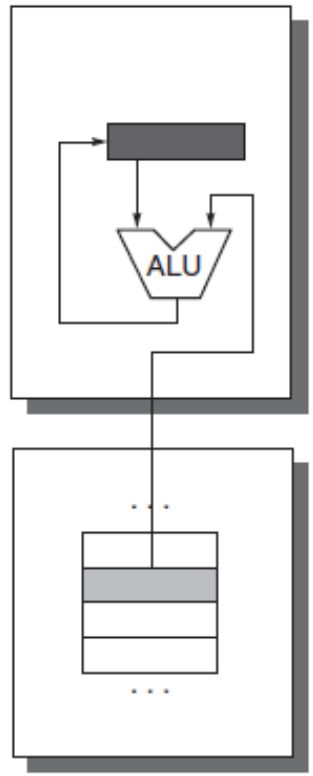
Acumulador

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**.



Clases de ISA

Acumulador: Ejemplo

Suponga la operación: $A = B + C$.

Clases de ISA

Acumulador: Ejemplo

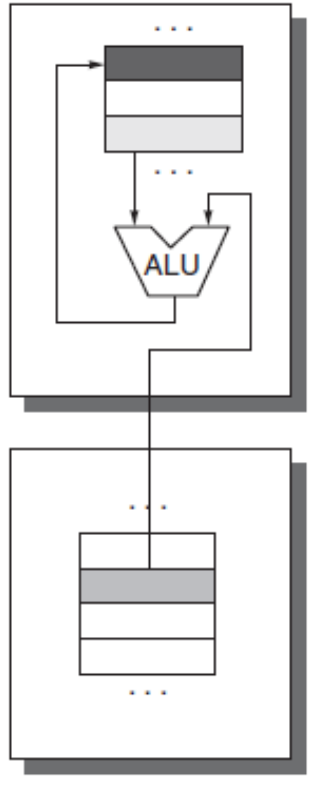
Registro (registro-memoria)

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.



Clases de ISA

Registro (registro-memoria)

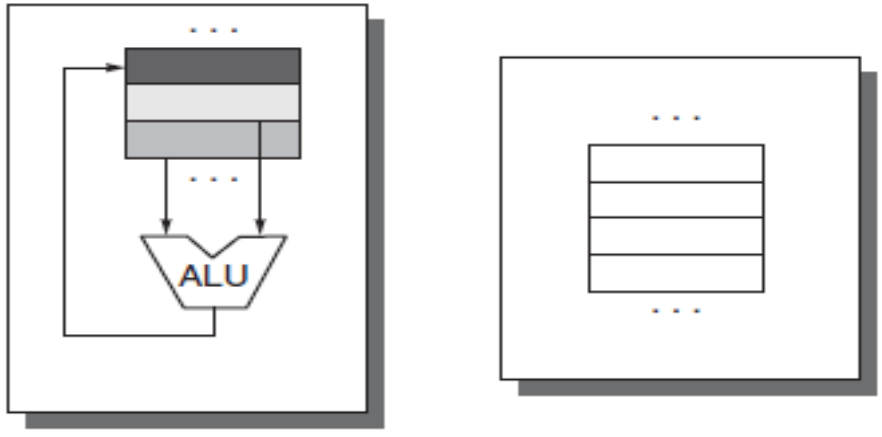
Suponga la operación: $A = B + C$.

Clases de ISA

Registro (registro-memoria)

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.

Clases de ISA

Registro-Registro

Suponga la operación: $A = B + C$.

Clases de ISA

Registro-Registro

Clases de ISA

Comparación

Tipo de Implementación	Ventajas	Desventajas
Stack	Simplicidad de evaluación de expresiones. Facilidad de implementación	No se ordena fácilmente Stack es *bottleneck*.
Acumulador	Instrucciones pequeñas	Acumulador es almacenamiento temporal, asociado con tráfico alto.
GPR	Generación de código sencilla Los datos se mantienen por largos períodos de tiempo	Todos los operandos deben ser nombrados previamente

Ejercicio Clases de ISA

Ejercicio

Represente el siguiente código como instrucciones de bajo nivel en cada una de las diferentes clases descritas anteriormente (para resta se usa la operación `sub`):

```
a=a+a;
c=a-b:
c=c-a;
```

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)

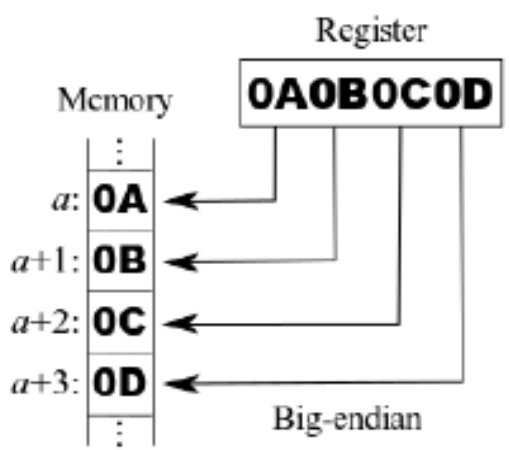
Nivel de palabra / word (32 bits)

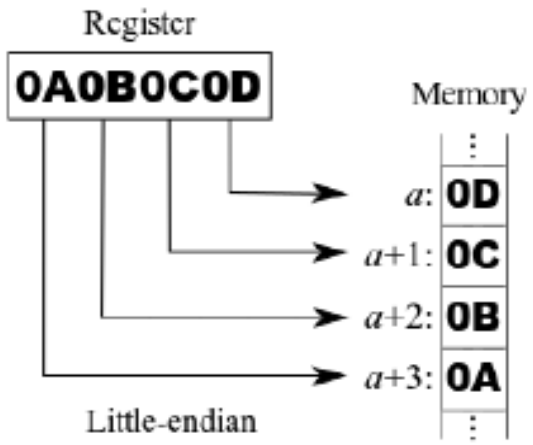
Nivel de doble palabra (64 bits)

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).





ón de

ón

Ejercicio Endianness

Ejercicio

Represente los siguiente objetos en memoria en *Little* y *Big Endian* a nivel de byte.

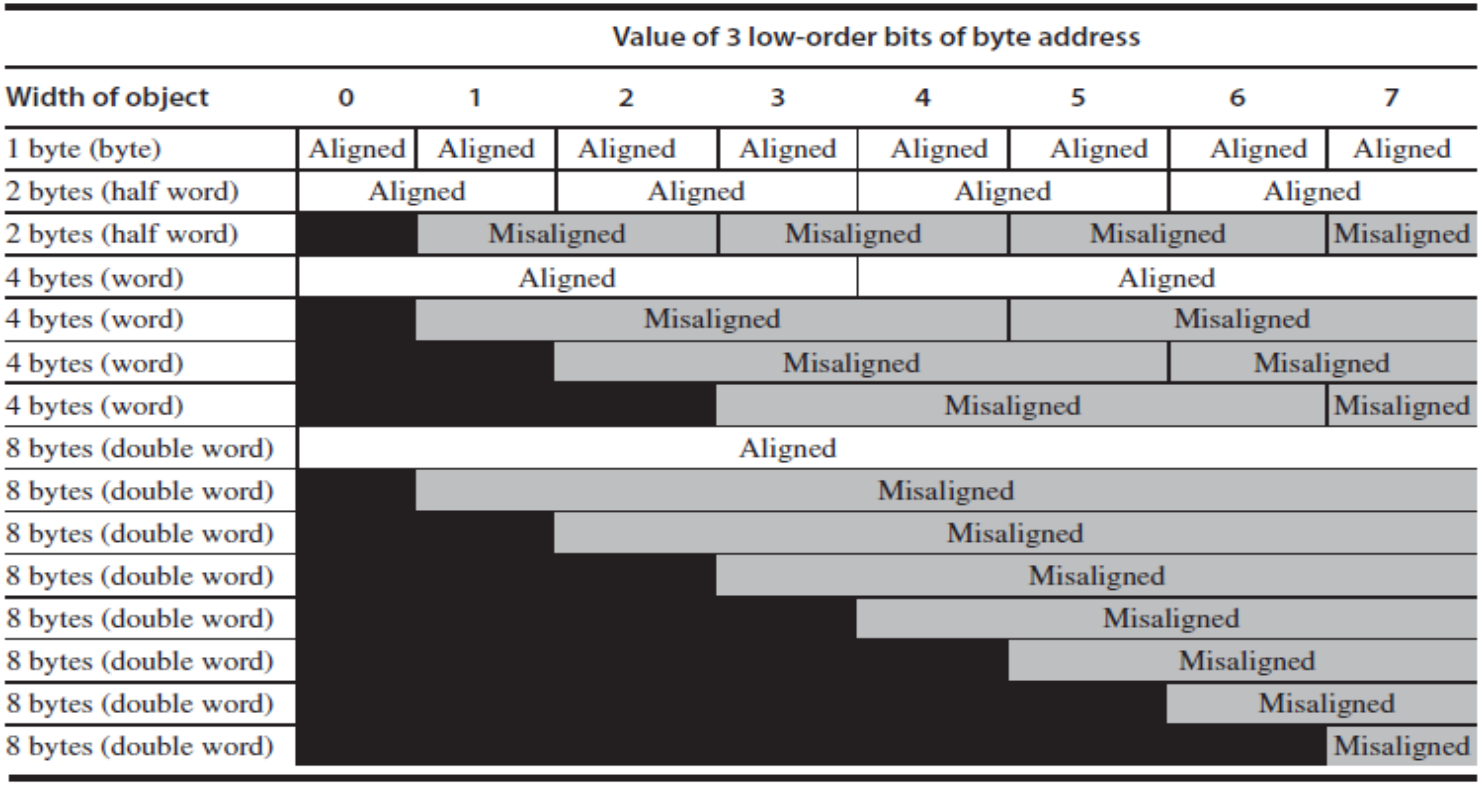
1. `0x12345678`
2. `0x87654321`
3. `0x78563412`
4. `0x21436587`

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.

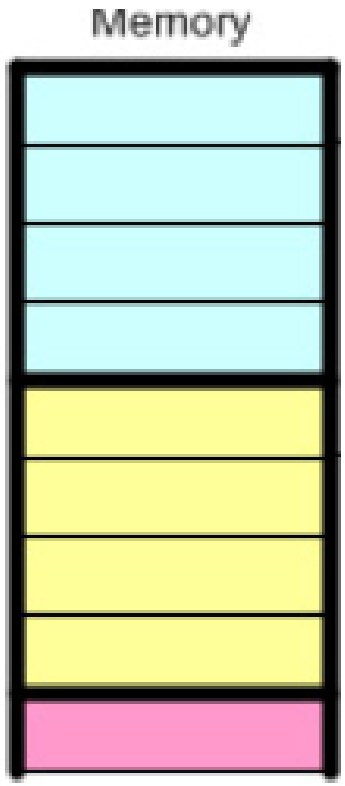


Alineamiento de memoria

Alineamiento de memoria

Problemas con accesos no alineados:

- CPU requiere más accesos a memoria.
- Manipulación extra es necesaria.
- Penalización de desempeño (dinero y memoria).



Alineamiento de memoria

Ejemplo

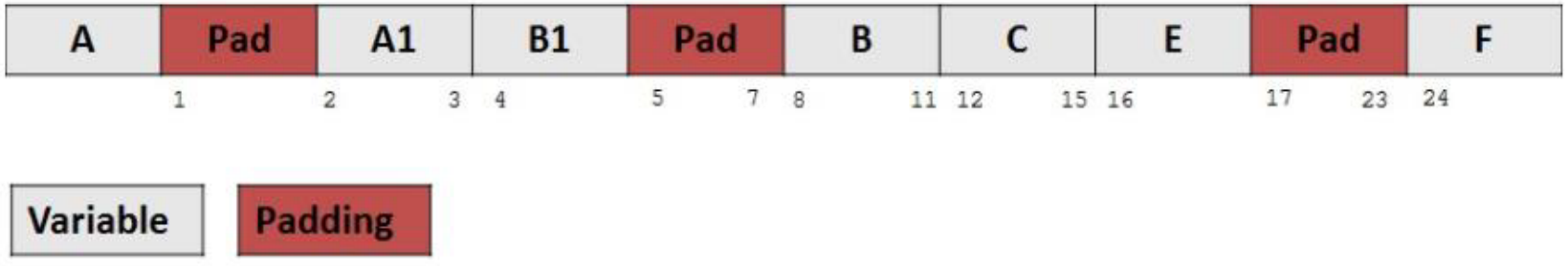
Ejemplo

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;
```

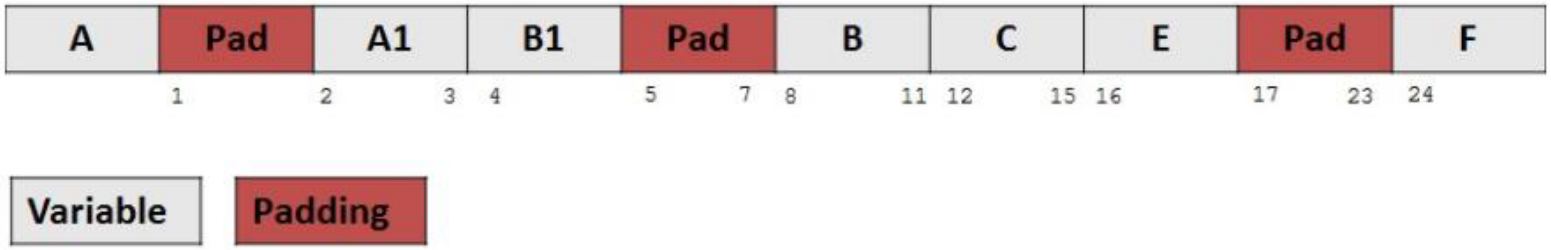
Data Type	32-bit (bytes)	64-bit (bytes)
char	1	1
short	2	2
int	4	4
long	8	8
float	4	4
double	8	8
long long	8	8
long double	4	16
Any pointer	4	8

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



Ejemplo

Ejemplo




```
Char a: 0x7ffec9734180
Short a1: 0x7ffec9734182
Char b1: 0x7ffec9734184
Float b: 0x7ffec9734188
Int c: 0x7ffec973418c
Char e: 0x7ffec9734190
Double f: 0x7ffec9734198
```

Enlace con código en C

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.

Modos de direccionamiento

Modo Registro

Cuando los operandos se encuentran en **registros**.

Ejemplo:

```
Add R4, R3
```

Lo que significa que:

Modo Inmediato

Modo utilizado para referenciar operaciones con **constantes**.

Ejemplo:

```
Add R4, #3
```

Lo que significa que:

Desplazamiento

El dato referencia se encuentra en la posición de memoria dada por la suma de una base (un inmediato) + un registro.

Ejemplo:

```
Add R4, 200(R1)
```

Lo que significa que:

Registro Indirecto

La dirección se determina por el contenido de un registro. Útil para utilizar punteros.

Ejemplo:

```
Add R4,(R1)
```

Lo que significa que:

Indexado : Base + índice

La dirección se calcula como la suma de un registro base + un registro índice. Utilizado para direccionar arreglos.
Ejemplo:

```
Add R4,(R1 + R2)
```

Lo que significa que:

Directo o Absoluto

La dirección efectiva se referencia directamente. Útil para acceder datos estáticos.
Ejemplo:

```
Add R4,(1001)
```

Lo que significa que:

Indirecto a Memoria

La dirección efectiva se calcula como el contenido de otra dirección de memoria.
Ejemplo:

```
Add R4,@(R1)
```

Lo que significa que:

Direccionamiento de memoria

Direccionamiento de memoria

x86

Al ser CISC permite operaciones que involucren memoria (ADD, MUL, etc).

Se emplean los registros base, punteros, y se pueden dar escalamientos y desplazamientos

```
1  MY_TABLE TIMES 10 DW 0 ; Allocates 10 words (2 bytes) each initialized to 0
2  MOV EBX, [MY_TABLE] ; Effective Address of MY_TABLE in EBX
3  MOV [EBX], 110 ; MY_TABLE[0] = 110
4  ADD EBX, 2 ; EBX = EBX +2
5  mov [EBX], 123 ; MY_TABLE[1] = 123
```

Direccionamiento de memoria

arm64

Define 3 modos "básicos" de direccionamiento.

1. Offset mode: El acceso a memoria esta dado por el valor del(os) registro(s) base y permite operaciones de desplazamiento.
2. Indexmode: Los registros base son modificados antes (pre-index) o después (post-index) del acceso a memoria.
3. PC-relative(literal): se indica PC+X donde X es un literal $< label >$.

Direccionamiento de memoria

arm64

Offset mode.

Example instruction	Description
LDR X0, [X1]	Load from the address in X1
LDR X0, [X1, #8]	Load from address X1 + 8
LDR X0, [X1, X2]	Load from address X1 + X2
LDR X0, [X1, X2, LSL, #3]	Load from address X1 + (X2 << 3)
LDR X0, [X1, W2, SXTW]	Load from address X1 + sign_extend(W2)
LDR X0, [X1, W2, SXTW, #3]	Load from address X1 + (sign_extend(W2) << 3)

```
1  // A C example showing accesses that a compiler is likely to generate.
2  void example_dup(int32_t a[], int32_t length) {
3      int32_t first = a[0];                          // LDR W3, [X0]
4      for (int32_t i = 1; i < length; i++) {
5          a[i] = first;                              // STR W3, [X0, W2, SXTW, #2]
6      }
7  }
```

Direccionamiento de memoria

arm64

Index mode.

Example instruction	Description
`LDR X0, [X1, #8]!`	Pre-index: Update X1 first (to X1 + #8), then load from the new address
`LDR X0, [X1], #8`	Post-index: Load from the unmodified address in X1 first, then update X1 (to X1 + #8)
`STP X0, X1, [SP, #-16]!`	Push X0 and X1 to the stack.
`LDP X0, X1, [SP], #16`	Pop X0 and X1 off the stack.

```
1  // A C example showing accesses that a compiler is likely to generate.
2  void example_strcpy(char * dst, const char * src)
3  {
4  char c;
5  do {
6      c = *(src++);              // LDRB W2, [X1], #1
7      *(dst++) = c;              // STRB W2, [X0], #1
8      } while (c != '\0');
9  }
```

Direccionamiento de memoria

arm64

PC-relative (literal)

Example instruction	Description
LDR W0, <label>	Load 4 bytes from <label> into W0
LDR X0, <label>	Load 8 bytes from <label> into X0
LDRSW X0, <label>	Load 4 bytes from <label> and sign-extend into X0
LDR S0, <label>	Load 4 bytes from <label> into S0
LDR D0, <label>	Load 8 bytes from <label> into D0
LDR Q0, <label>	Load 16 bytes from <label> into Q0

Referencias

J. Hennesy y D. Patterson (2012)

Computer Architecture: A Quantitative Approach. 5th Edition. Elsevier – Morgan Kaufmann.

J. González y R. García (2019)

Notas de clase de los profesores: Jeferson González y Ronald García.

ARMv8-A Architecture Reference Manual

Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4