



INSTITUTO POLITÉCNICO NACIONAL

ESCOM



Unidad de Aprendizaje

Arquitectura de Computadoras

Febrero 2025

Arquitectura RISC (Reduced Instruction Set Computer) ***Computadora de conjunto de instrucciones reducido***

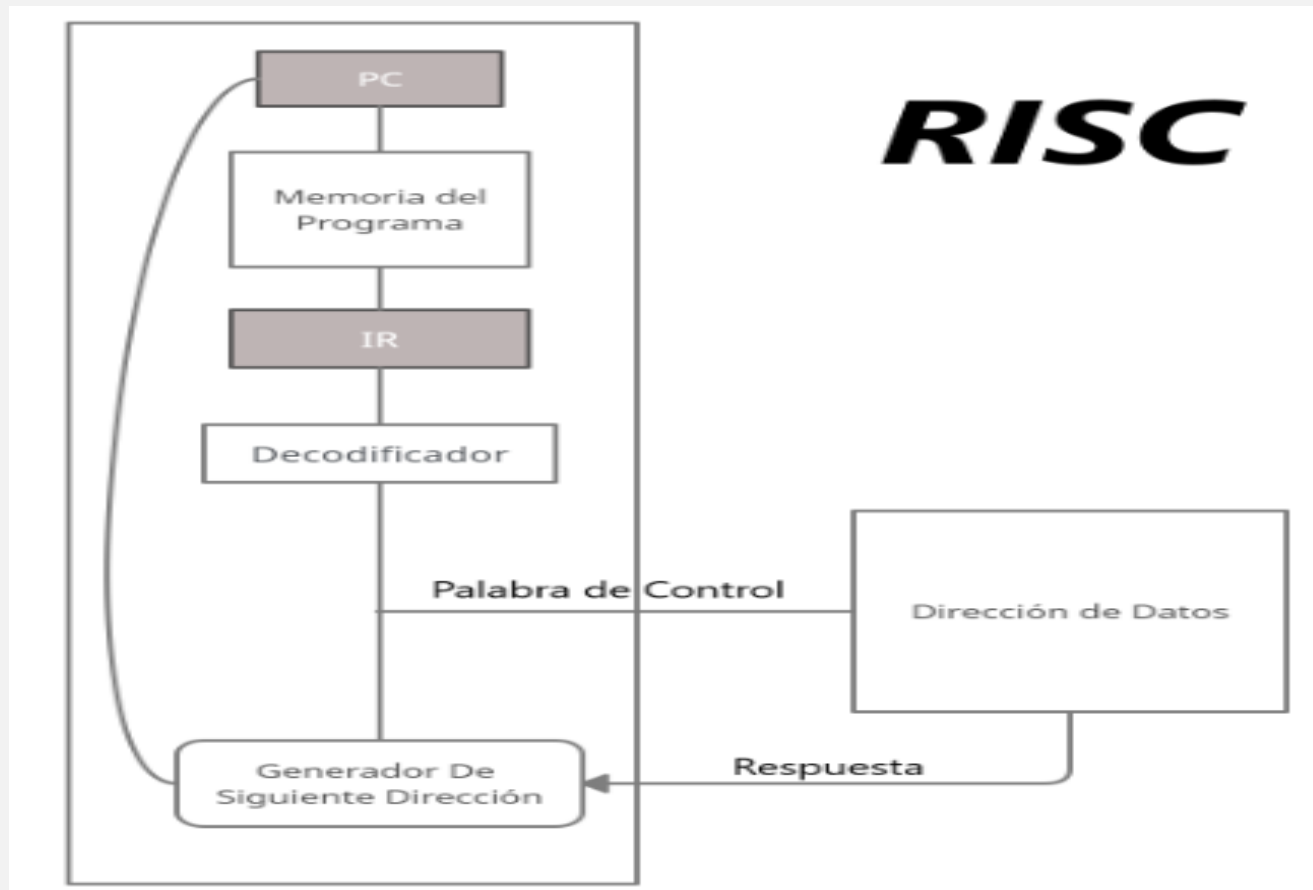
La arquitectura **RISC** utiliza un número reducido de instrucciones básicas, que son más simples y se pueden ejecutar en un solo ciclo de reloj.

La mayoría de las arquitecturas RISC usan instrucciones de longitud fija, lo que simplifica el decodificador de instrucciones y acelera el proceso de decodificación. El diseño es registro-registro y pocos modos de direccionamiento.

La RISC tiende a tener una mayor cantidad de registros internos para almacenar datos temporales, lo que reduce el número de accesos a la memoria principal, que es más lento que el acceso a los registros.

En RISC, las operaciones de carga y almacenamiento de datos desde/hacia la memoria son instrucciones separadas y explícitas, en lugar de estar integradas en otras operaciones como en CISC.

La simplicidad de las instrucciones facilita el pipelining, donde múltiples instrucciones pueden estar en diferentes etapas de ejecución simultáneamente, mejorando la eficiencia.



Ejemplos de Arquitecturas RISC

ARM (usado en muchos dispositivos móviles y microcontroladores)

MIPS (utilizado en ciertos sistemas embebidos y en educación)

RISC-V (una arquitectura de código abierto que está ganando popularidad)

SPARC (utilizada por Sun Microsystems en sus estaciones de trabajo y servidores)

Arquitectura ARM-Advanced RISC Machine

Máquina avanzada de RISC

El microprocesador de arquitectura ARM está basado en la arquitectura RISC y sus ventajas son la eficiencia energética, y son aptos para dispositivos móviles que funcionan con batería.

microprocesador de
Arquitectura ARM

Teléfonos inteligentes (Samsung y Apple)

Relojes inteligentes (Apple Watch)

Tabletas y computadoras portátiles
(MacBooks)

Tarjetas de desarrollo Raspberry Pi



ARQUITECTURA ARM-Advanced RISC Machine

Conjunto de Instrucciones (ISA, Instruction Set Architecture)

- Empresa británica Arm Holdings(antes Acorn Computers)
- Se crea en los años 1980.
- El impacto de ARM es con la llegada de dispositivos móviles, portátiles, equipos de escritorio y en el caso de Apple pequeños microservidores y supercomputadoras como Fugaku.
- Sus extensiones usan SIMD como NEON, SVE (conjunto variable en longitud entre 1280 bits hasta 2048 bits).
- ARM requiere el pago de regalías por cada chip fabricado que utilice su ISA, además de pagar licencias por la IP (propiedad intelectual) relacionada con los núcleos ARM.

Arquitecturas RISC-V y ARM.

Las dos ISAs compiten por el liderazgo en el mundo de los microprocesadores en casi todos los segmentos del mercado.



No obstante, cada una presenta sus propias fortalezas y estrategias.



Desde dispositivos IoT y de bajo consumo hasta el ámbito de la computación de alto rendimiento-HPC-High Performance Computing) (computadoras que trabajan juntas para resolución de problemas complejos o supercomputadoras).

Algunos ejemplos de estas HPC utilizan microprocesadores como Intel Xeon o AMD Epyc, IBM Power, Oracle Sparc o nuevas unidades basadas en ARM.

ARQUITECTURA RISC-V (*RISC: Reduced Instruction Set Computing*)

Conjunto de Instrucciones (ISA, Instruction Set Architecture)

- Reducidas y sencillas

- Conjunto de instrucciones (CI) de código abierto (BSD License)

Cualquier persona puede crear un procesador que ejecute instrucciones RISC-V sin pagarle a la organización que gestiona la arquitectura. *Los esquemas y especificaciones de RISC-V están disponibles públicamente.*

Las empresas pueden personalizar o agregar extensiones a la ISA para necesidades específicas, como mejorar el rendimiento en tareas concretas (como procesamiento gráfico, inteligencia artificial, etc.) sin pagar regalías.

- Conjunto de instrucciones creado en la Universidad de Berkeley 2010

y sus especificaciones se lanzaron en 2011. Se creó la RISC-V Foundation después RISC-V International, que cuenta con participantes como Google, Nvidia, Westerns Digital, Intel , Seagate, Qualcomm AMD, Analog Devices, Infineon, IBM, Mediatek, Nokia, Raspberry Pi Foundation, Arduino, Renesas, Sony, Siemens, Texas Instruments, ST Microelectronics, entre otras.

ARQUITECTURA RISC-V (RISC: Reduced Instruction Set Computing)

- Compatibilidad

los procesadores basados en RISC-V tienen la capacidad de ejecutar programas o software que han sido diseñados para otras implementaciones de RISC-V o incluso para otras arquitecturas de procesadores, como ARM o x86, bajo ciertas adaptaciones (emulación, compiladores y herramientas de desarrollo).

- Diseño modular (el núcleo del CI es estable, no cambia)

Permite la creación de sistemas o componentes que se pueden dividir en módulos con una función específica, personalizables e intercambiables. Se crean extensiones para implementar procesadores de distintas prestaciones (extensiones de multiplicaciones, punto flotante, etc)

- Conjunto de instrucciones base RV32I(47 instrucciones-reducibles a 38) además 4 tipos de instrucciones base (dos de ellas tienen variantes por lo que se consideran 6)



Tipo set instruc.	CISC	RISC	RISC
Bits	16, 32, 64	32, 64	32, 64, 128
Endianness	Little	Bi (Little por def.)	Bi (Little por def.)
Año lanzamiento	1978	1985	2010
Núm Instr. 32Bits	981 (+3684 variantes)	≈110 (Sin ext.)	49 (Set base)
Registros	8	16 (15 + pc)	32 (1 siempre a 0)
Licencia	Propietaria	Propietaria	Código abierto

Fuente: Universidad de Nebraska

https://www.unomaha.edu/college-of-information-science-and-technology/research-labs/_files/enumerating-x86-64-instructions.pdf

<http://www.riscbook.com/spanish/guia-practica-de-risc-v-1.0.5.pdf>

RISC-V en español #01-Presentación, José Antonio Puga

Etapas de diseño en niveles de abstracción

niveles de abstracción



La microarquitectura es la implementación del conjunto de instrucciones.

Conjunto de instrucciones en código máquina y registros de la computadora

Para el diseño de la microarquitectura se debe realizar el RTL (Nivel de transferencia de registros) con Verilog o VHDL.

En este nivel se definen las máquinas de estados, registros y bloques combinacionales, esto convierte el diseño al nivel de compuertas, con la implementación detallada del camino de los datos.



Principios de diseño de arquitectura de computadoras

- El rendimiento mediante la paralelismo (aumentar la velocidad de procesamiento usando múltiples unidades de ejecución).
- El rendimiento mediante la jerarquía de memoria (Se organiza la memoria en niveles (caché, RAM, almacenamiento secundario) para menor latencia.
- El rendimiento mediante la especulación (Uso de técnicas para predecir valores o caminos de ejecución y reducir tiempos de espera, como la predicción de saltos)
- Mejora el rendimiento con el uso de interfaces bien definida (modularidad y escalabilidad, Ley de Moore)
- Eficiencia mediante la mejora del caso común (Optimizar el diseño de acuerdo a los escenarios que ocurren con mayor frecuencia, como optimización de accesos a memoria y la ejecución de instrucciones frecuentes en menos ciclos)

Conjunto de instrucciones de RISC-V

- Existen versiones del conjunto de instrucciones:
32 bits, 64 bits, 128 bits
- Conjunto de instrucciones es modular, este se desarrolla a partir del conjunto base (operaciones aritmética y lógicas con enteros) con las extensiones que se requieran (puede incluir la multiplicación y división, operaciones de extensión de punto flotante en precisión simple o doble, operaciones atómicas para sincronizar sistemas multiprocesador, instrucciones de vector y comprimidas)

El tipo de conjunto de instrucciones a implementar se identifica con la nomenclatura:

- **RV32I**: 32 bits (operaciones aritmética y lógicas con enteros)
- **RV32IM**: 32 bits (soporta multiplicación y división con enteros)
- **RV64IMAFD**  **RV64G**

Set de instrucciones y nomenclatura

Nombre	Descripción	Inst.
RV32I	Set Instrucciones Base; Enteros 32 bits (Integers)	49
RV32E	Como RV32I para sistemas integrados. 16 registros.	49
RV64I	Set Instrucciones Base; Enteros 64 bits (Integers)	14(+)
RV128I	Set Instrucciones Base; Enteros 128 bits (Integers)	14(+)

Extensiones		Inst.
M	Para enteros, Multiplicación y División.	8
A	Instrucciones Atómicas.	11
F	Instrucciones coma Flotante Simple precisión.	25
D	Instrucciones coma flotante Doble precisión.	25
G	Abreviación para las 4 instrucciones anteriores juntas.	

Instrucciones atómicas están definidas en la extensión "A" (Atomic Instructions) de la ISA y se dividen en dos grandes categorías:

- Instrucciones de lectura-modificación-escritura (AMO, Atomic Memory Operations)

Leen un valor de la memoria, lo modifican y lo escriben de vuelta en una sola operación atómica, es decir, *se ejecuta completamente o no se ejecuta en absoluto*, sin posibilidad de interrupción.

Ejemplos:

AMOSWAP.W x1, x2, (x3) → Intercambia el valor de x2 con el valor en memoria en la dirección

AMOADD.W x1, x2, (x3) → Suma x2 al valor en x3 y guarda el resultado en x3, devolviendo el valor antiguo en x1.

- Instrucciones de carga y almacenamiento condicional (LR/SC, Load-Reserved/Store-Conditional)

Son usadas para implementar mecanismos de exclusión mutua, evita que dos hilos modifiquen una misma variable simultáneamente con resultados inconsistentes (condiciones de carrera)

Ejemplo:

LR.W x1, (x2) → Carga un valor de 32 bits desde la dirección x2 en x1 y establece una reserva en esa dirección.

SC.W x3, x4, (x2) → Intenta almacenar x4 en x2 solo si la reserva sigue activa. Si la reserva fue modificada por otro núcleo, la operación falla.

RISC-V incluye instrucciones para trabajar con números en **coma flotante** según el estándar *IEEE 754* y se agrupan en dos extensiones opcionales:

Extensión "F" → Soporta precisión simple (32 bits).

Extensión "D" → Soporta precisión doble (64 bits).

Set de instrucciones y nomenclatura

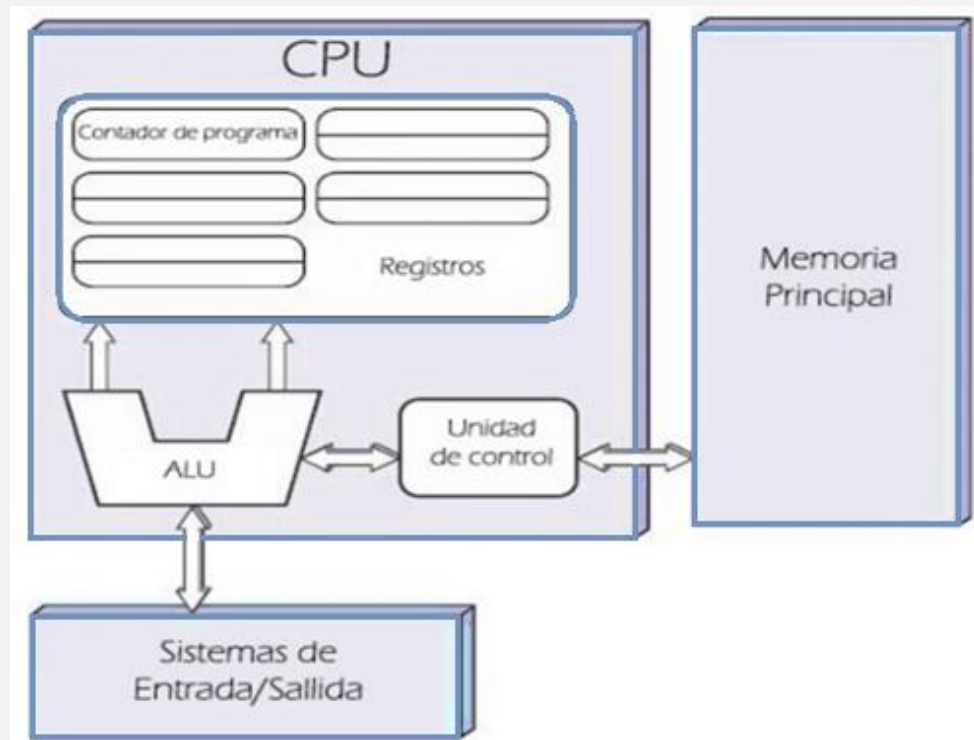
Set de instrucciones adicionales

(la mayoría no esán completamente implementadas al 2024)

Extensiones		Inst.
Q	Quad-Precisión coma flotante.	27
L	Decimal coma flotante	Sin Definir
C	Instrucciones Comprimidas.	36
B	Manipulación de Bits.	42
J	Dynamic Translated Languages.	Sin Definir
T	Memoria Transaccional.	Sin Definir
P	Instrucciones SIMD emPaquetadas.	Sin Definir
V	Operaciones con Vectores.	186
N	INTerrupciones a nivel usuario.	3
H	Hipervisor.	2
S	Instrucciones nivel Supervisor.	7

Registros y ABI (App. Binary Interface)

- ▶ Hay **32 registros** disponibles **x0..x31** (16 para el modo comprimido x0..x15).
- ▶ Para el módulo coma flotante (floating point) hay otros 32: **f0..f31**.
- ▶ No hay ninguna obligación de usar un registro de una forma concreta.
- ▶ Hay un ABI no obligatorio, son recomendaciones que todo el mundo usa.



Registros y ABI (App. Binary Interface)

Registros enteros del Conjunto Base (set base)

Reg.	ABI	Descripción	Guarda
x0	zero	Siempre vale cero. Sólo lectura.	
x1	ra	Return Address . Dirección de retorno en una subrutina.	
x2	sp	Stack Pointer . Puntero de la pila.	Sí
x3	gp	Global Pointer . Puntero global usado para acceder al ".bss"	
x4	tp	Thread Pointer . Análogo a gp, pero multihilo.	
x5..x7	t0..t2	Temporales . Se puede sobrescribir su valor sin restaurarlos.	
x8	s0/fp	Saved Register/Frame Pointer . Suele usarse para depurar (GDB).	Sí
x9	s1	Saved Register . Como Temporal, pero hay que restaurar valor.	Sí
x10..x11	a0..a1	Function Argument/Return Value . Devolver valor sin stack.	
x12..x17	a2..a7	Function Argument . Recibir valores sin usar el stack.	
x18..x27	s2..s11	Saved Register . (No disponibles en modo comprimido).	Sí
x28..x31	t3..t6	Temporales . (No disponibles en modo comprimido).	Sí

Codificación de instrucciones en 32 bits

Formato de instrucciones de 32 bits Risc-V

Format	Bit																																		
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
Register/ register	funct7							rs2					rs1					funct3			rd					opcode									
Immediate	imm[11:0]												rs1					funct3			rd					opcode									
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode									
Branch	[12]	imm[10:5]							rs2					rs1					funct3			imm[4:1]				[11]	opcode								
Upper immediate	imm[31:12]																				rd					opcode									
Jump	[20]	imm[10:1]											[11]	imm[19:12]							rd					opcode									

opcode (7 bits) : Especifica uno de los 6 tipos de formato de instrucción.

funct7 (7 bits) y funct3 (3 bits): Estos dos campos amplían el campo *opcode* para especificar la operación que se va a realizar.

rs1 (5 bits) y rs2 (5 bits): Especifica por medio del índice, el primer y el segundo registro operando respectivamente (son los registros fuente)

rd (5 bits): Especifica por medio del índice, el registro destino al cual será direccionado el resultado de la operación realizada.

Codificación de instrucciones en 32 bits

Codificación “R” Registro -Registro

Format	Bit																																
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Register/ register	funct7							rs2					rs1					funct3			rd					opcode							

- **Opcode + funct3 + funct7** son los que codifican la instrucción (tipo de formato y tipo de operación)
- **rd, rs1, rs2** son identificadores de registros. Los campos de dichos registros tienen 5 bits cada uno para direccionar hasta $2^5 = 32$. Esto significa, que se pueden seleccionar hasta **32 registros diferentes (x0 a x31)**, pero cada registro almacena palabras de 32, 64 o 128 bits según el ISA a utilizar.
- Siempre se usan 3 registros en operaciones con sólo registros.

Codificación de instrucciones en 32 bits

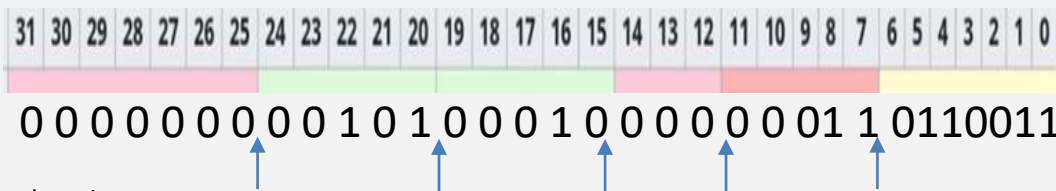
Codificación "R" Registro -Registro

Format	Bit																																																																								
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																																									
Register/ register	funct7							rs2						rs1						funct3			rd				opcode																																														
	31							25						24						20						19						15						14						12						11						7						6						0					
	0000000							rs2						rs1						000						rd						0110011								R add																																	
	0100000							rs2						rs1						000						rd						0110011								R sub																																	
	0000000							rs2						rs1						001						rd						0110011								R sll																																	

Ejemplo: **Instrucción add:** **add rd, rs1, rs2**

Se codifica como: **funct7 + funct3 + opcode** (utilizando los códigos se tiene)
 0000000 000 0110011

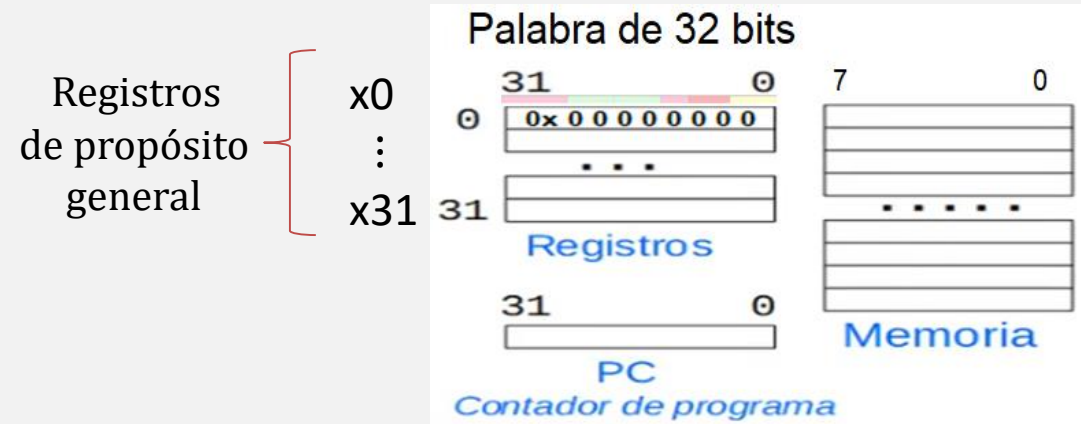
Si se tiene por ejemplo la instrucción: **add x3, x2, x5**, considerando los registros más la **codificación** el número total de bits de la instrucción es 32 (como se observa arriba en las figuras). Esto es:



 0b 0000000000101000100000000110110011 = 0x 005101b3

ISA RV32I: 32 bits operación con enteros

El número de registros en RISC-V depende del número de bits asignados para identificar los registros, esto es 5 bits ($2^5 = 32$ registros), cada uno de 32 bits de tamaño.



En **RISC-V con ISA de 32 bits (RV32I)** el tamaño de la *instrucción* y el tamaño de la **palabra en memoria** son conceptos diferentes.

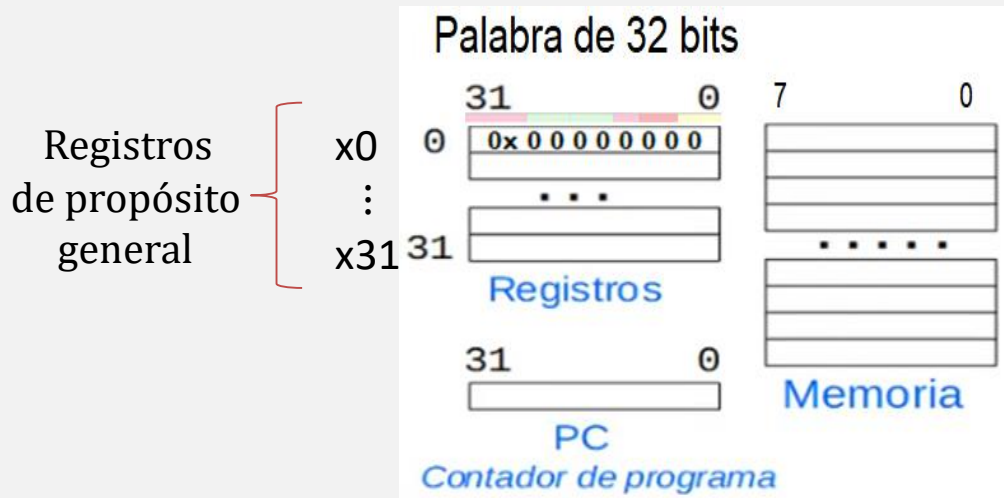
En **RV32I**:

Tamaño de la instrucción

Cada instrucción tiene **32 bits (4 bytes)**, lo que significa que cada instrucción ocupa **4 bytes** en memoria.

ISA RV32I: 32 bits operación con enteros

En **RV32I**:



Tamaño de la palabra en memoria

Una "**palabra**" se define como 32 bits (4 bytes) en una arquitectura de 32 bits.

Esto significa que cuando se usa **lw** (load word) se cargan 4 bytes desde la memoria al registro. Las direcciones de memoria avanzan en bytes, no en palabras.

Ejemplo:

La memoria está organizada en bytes (8 bits por dirección). Una palabra de 32 bits ocuparía direcciones consecutivas como:

Dirección
de memoria

0x1000 → byte 1

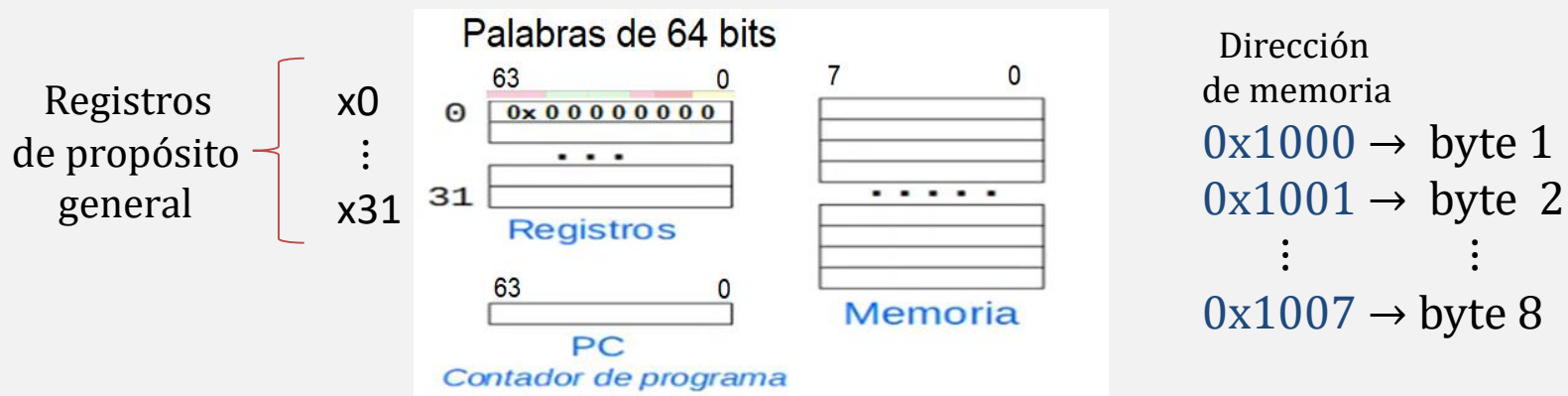
0x1001 → byte 2

0x1002 → byte 3

0x1003 → byte 4

RV64I: 64 bits operación con enteros

El **número de registros** en RISC-V de 64 bits, sigue siendo **32** debido a que número de bits asignados para identificar los registros, esto es 5 bits ($2^5 = 32$ registros), pero **cada uno es de 64 bits** de tamaño.



Tamaño de una palabra en RV64I

En RV64I

Una **palabra ("word")** sigue siendo de **32 bits (4 bytes)**, igual que en RV32I.

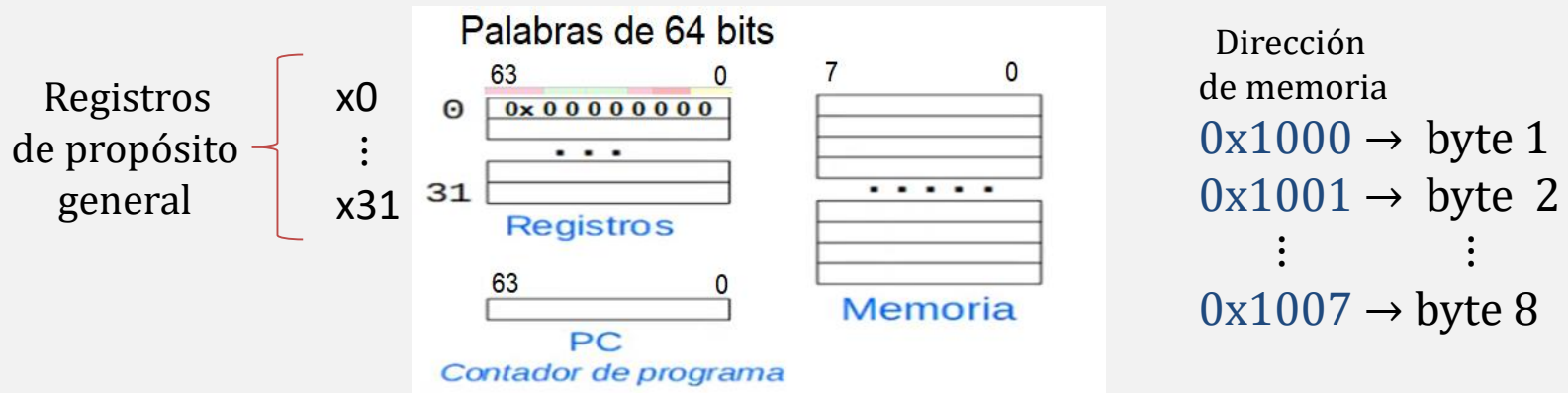
Sin embargo, en RV64I también existe el concepto de una **doble palabra ("double word")**, que equivale a **64 bits (8 bytes)**.

Tamaño de la localidad de memoria

La memoria sigue organizada en bytes (8 bits por dirección de memoria) igual que en RV32I.

Cada dirección de memoria representa **1 byte (8 bits)**, por lo que, si tenemos una **doble palabra**, ocupará **8 direcciones de memoria**.

RV64I: 64 bits operación con enteros



Existen distintos tamaños de acceso a la memoria:

Byte (8 bits) → **lb** (load byte)

Media palabra (16 bits) → **lh** (load half-word)

Palabra (32 bits) → **lw** (load word)

Doble palabra (64 bits) → **ld** (load double word)

Operaciones aritméticas (suma y resta)

Mnemónico

add a, b, c // a ← b+c
sub a, b, c // a ← b-c

En estas operaciones se observa simplicidad, pues hay dos valores como fuente y uno como destino, lo que hace un código uniforme y con la misma estructura.

Lenguaje C

f = (a+b) - (c+d)

t0 - t1

Lenguaje Ensamblador

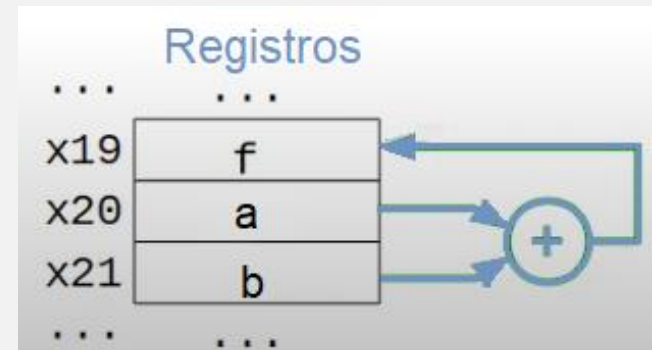
add t0, a, b // t0 ← a+b
add t1, c, d // t1 ← c+d
sub f, t0, t1 // f ← t0-t1

En realidad en ensamblador lo que se utiliza son registros. En RISC-V las operaciones aritméticas y cómputo se hacen entre registros (no se usa la memoria). Los registros se llaman x y el número del registro (algunos registros tienen propósito específico).

add x19, x20, x21 // x19 ← x20 + x21

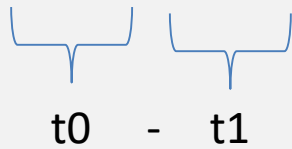
Lenguaje C Lenguaje Ensamblador

f = (a+b) add x19, x20, x21 // x19 ← x20 + x21



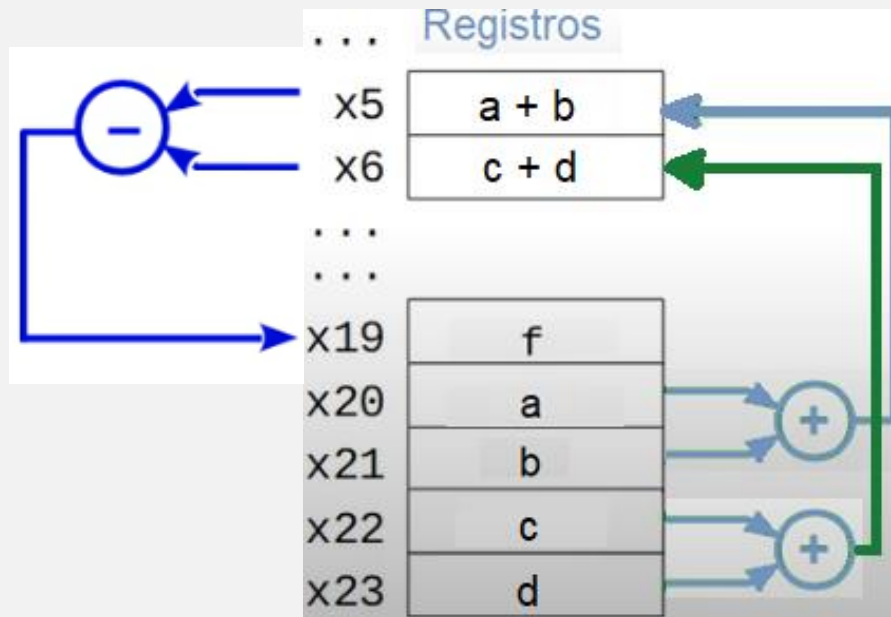
Lenguaje C

$f = (a+b) - (c+d)$



Lenguaje Ensamblador

```
add x5, x20, x21    // x5 ← x20 + x21
add x6, x22, x23    // x6 ← x22 + x23
sub x19, x5, x6      // x19 ← x5 - x6
```



RV32I: 32 bits (instrucción de **carga (load)** y **almacenamiento (store)**)

Operación LOAD

`lw x1, 0(x2)` # carga una palabra (32 bits) desde la dirección en x2 a x1

En esta instrucción:

lw (load Word-carga una palabra) es el opcode. La operación es para cargar una palabra de 32 bits desde una dirección de memoria.

x1 es el registro destino donde se almacenará el valor (de 32 bits) cargado desde la memoria.

0(x2) es la dirección de memoria desde donde se va a cargar la palabra de 32 bits. El valor dentro del registro x2 es tratado como una dirección base y el número es un desplazamiento (offset) a partir de esa dirección base. En este caso, no hay desplazamiento, la dirección es solamente el valor contenido en x2.

RV32I: 32 bits (instrucción de **carga (load)** y **almacenamiento (store)**)

Operación LOAD

`lw x1, 0(x2)` # carga una palabra (32 bits) desde la dirección en x2 a x1

Ejemplo:

Suponiendo que el valor en el registro **x2** es **0x1000** (dirección de memoria colocada en el x2, a la cual se debe acceder para obtener el dato) y en esa dirección está almacenado el valor **0x12345678** (una palabra de 32 bits).

Al ejecutar la instrucción, se lee el valor **0x12345678** desde la dirección **0x1000** en la memoria y se carga en el registro x1, por lo tanto el registro **x1** contendrá el valor **0x12345678**.

RV32I: 32 bits (instrucción de **carga (load)** y **almacenamiento (store)**)

Operación LOAD

Ejemplos:

```
lb  x5, 4(x10)  # Carga 1 byte desde la dirección (x10 + 4) a x5 (con signo)
lbu x5, 4(x10)  # Carga 1 byte sin signo
lh  x5, 4(x10)  # Carga 2 bytes (medio word) con signo
lhu x5, 4(x10)  # Carga 2 bytes sin signo
lw  x5, 4(x10)  # Carga 4 bytes (word) en x5
```

Acceso a la memoria:

Byte (8 bits) → **lb** (load byte)

Media palabra (16 bits) → **lh** (load half-word)

Palabra (32 bits) → **lw** (load word)

Doble palabra (64 bits) → **ld** (load double word)

RV32I: 32 bits (carga (load) y almacenamiento (store))

Operación STORE

sw x1, 0(x2) # Almacenar una palabra (32 bits) desde x1 en la dirección x2

sb x5, 4(x10) # Guarda 1 byte desde x5 en la dirección (x10 + 4)

sh x5, 4(x10) # Guarda 2 bytes (medio word)

sw x5, 4(x10) # Guarda 4 bytes (word)

Ejemplo:

lw x5, 0(x10) # Cargar el valor almacenado en la dirección x10 al reg. x5

addi x5, x5, 1 # Incrementar el valor (sumándole 1) en el reg. x5

sw x5, 0(x10) # Guardar el nuevo valor en la dirección x10