ARQUITECTURA DE COMPUTADORES

# Lenguaje ensamblador

## Diseño de un repertorio de instrucciones

**Katia Leal Algara, URJC**
**Juan González Gómez, URJC**

# Tipos de repertorios

- Según el tipo de repertorio, las máquinas se pueden clasificar en:
  - **CISC**, ***C****omplex* ***I****nstruction* ***S****et* ***C****omputer*
  - **RISC**, ***R****educed* ***I****nstruction* ***S****et* ***C****omputer*

# Máquinas CISC

- Tienen un conjunto de instrucciones que se caracteriza por ser muy amplio y permitir operaciones complejas entre operandos situados en la memoria o en los registros internos, en contraposición a la arquitectura RISC
- Pertenecen a la primera corriente de construcción de procesadores, antes del desarrollo de los RISC

# Máquinas RISC

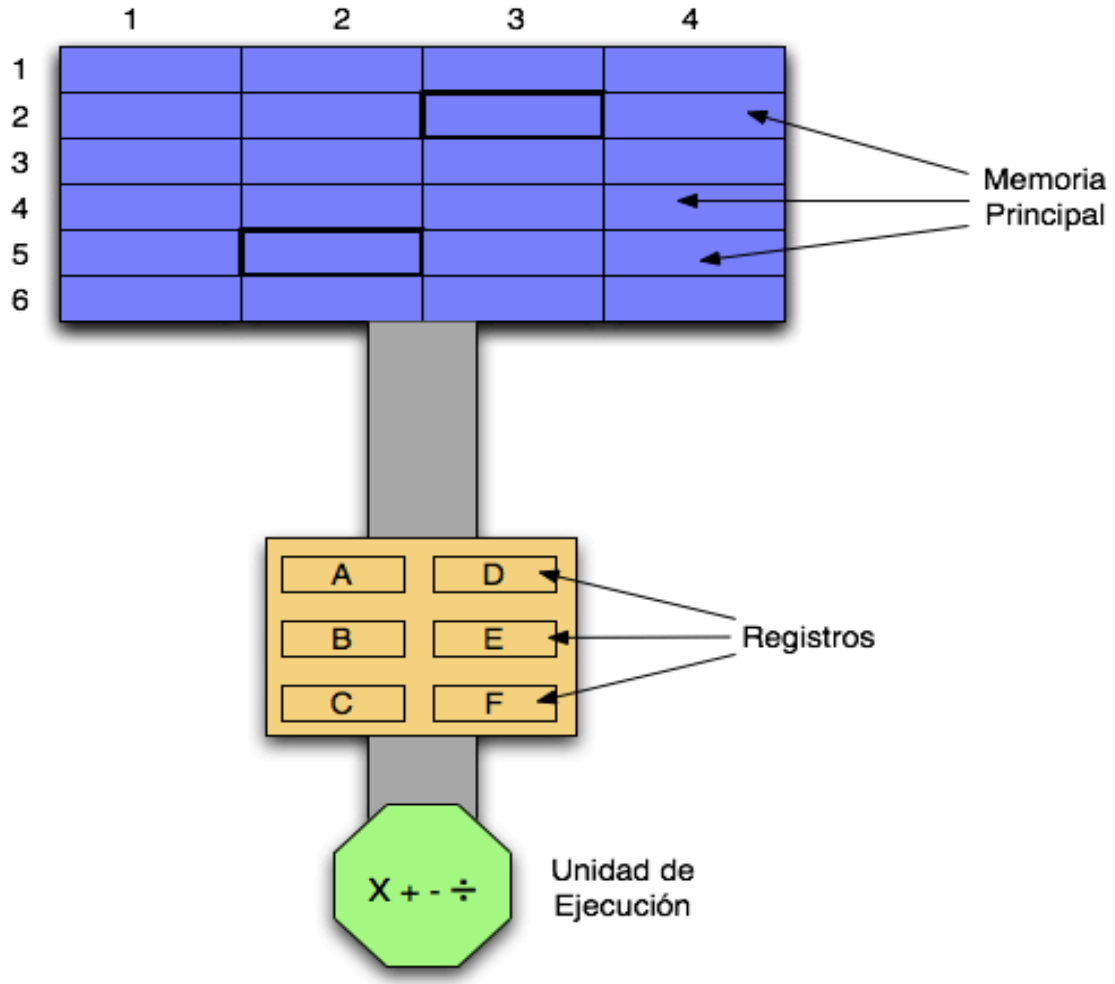
- Características fundamentales:
  - Instrucciones de tamaño fijo y presentadas en un reducido número de formatos
  - Solo las instrucciones de carga y almacenamiento acceden a la memoria de datos
- Objetivos:
  - Posibilitar la segmentación y el paralelismo en la ejecución de instrucciones
  - Reducir los accesos a memoria
- Arquitecturas de carga-almacenamiento

# CISC Vs RISC

| CISC | RISC |
|---|---|
| Conjunto *grande* de instrucciones: operaciones complejas y muchos modos de direccionamiento | Conjunto *pequeño* de instrucciones: pocos tipos de datos y modos de direccionamiento |
| Programa objeto *corto* | Programa objeto *largo* |
| Compilador más *sencillo* | Compilador más *complejo* |
| Codificación *variable* | Codificación *fija* |
| Hardware más *complejo*, más *lento*, más *caro* | Hardware más *sencillo*, más *rápido*, más *barato* |
| *Difícilmente* optimizable | *Fácilmente* optimizable |

# Ejemplo: RISC Vs CISC

- (2:3) X (5:2), almacenar el resultado en (2:3) de memoria

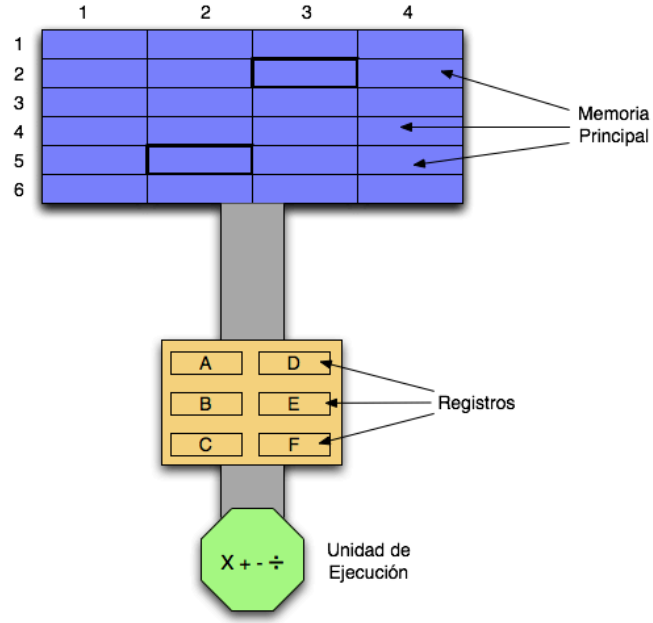


# Solución CISC

- **Objetivo**: completar una tarea en el menor número de líneas de código ensamblador posibles, ¿por qué?
- Construcción de un microprocesador capaz de comprender y ejecutar una serie de operaciones complejas
- La tarea completa puede ser llevada a cabo con una única instrucción específica, *MUL*
- (2:3) X (5:2) => (2:3)
- Lenguaje de alto nivel:

  a = a * b;
- Lenguaje ensamblador (CISC):

  MUL (2:3),(5:2)



# Solución RISC

- **Objetivo**: usan instrucciones sencillas que se puedan ejecutar rápidamente
- Arquitecturas basadas en **registros de propósito general** (GPR) que operan siempre sobre operandos almacenados en el procesador (registros), cerca de la unidad de ejecución (ALU)
- Lenguaje de alto nivel:

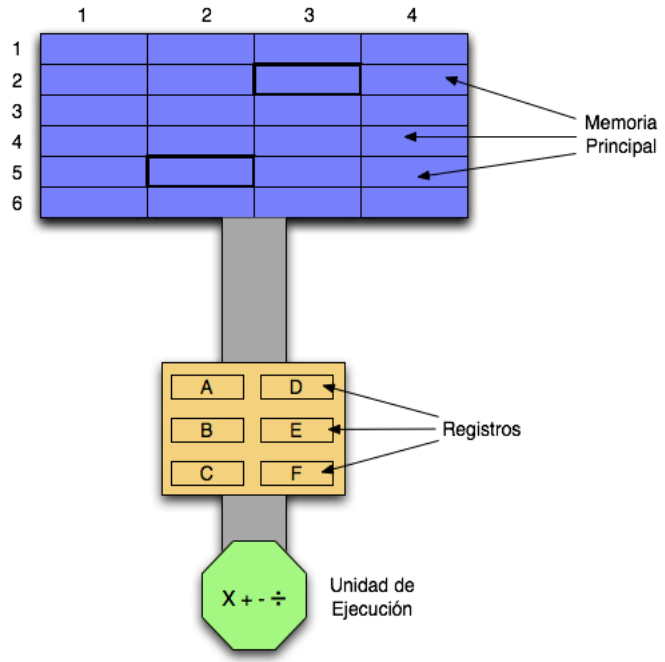
  a = a * b;
- Lenguaje ensamblador (RISC):

  LOAD A,(2:3)

  LOAD B,(5:2)

  MUL  A,B

  STORE (2:3),A



# ¿Por qué RISC-V?

- Muchas alternativas RISC: ARM o MIPS, ¿por qué RISC-V?
- **Una cuestión de docencia:**
  - Hardware abierto, *open-source hardware*
  - Similar a MIPS, que ha sido la arquitectura de referencia utilizada en la mayor parte de los textos docentes disponibles para la enseñanza de Arquitectura de Computadores
- Los repertorios de microprocesadores RISC son muy parecidos entre sí, conociendo uno podemos entender fácilmente otros identificando sus particularidades

# Investigadores pioneros

- **John L. Hennessy**
  - En 1981 el doctor Hennesy reunió a varios investigadores para centrarse en la arquitectura de computadores RISC
  - Hennesy ayudó a transferir esta tecnología a la industria y a darla a conocer con sus múliples publicaciones
  - En 1984, **cofundó la compañía MIPS** Computer Systems, hoy día MIPS Technologies
  - En los últimos años, su investigación se ha centrado en la Arquitectura de Computadores de altas prestaciones
- **David A. Patterson**
  - Dirigió el diseño e **implementación del RISC 1**, probablemente el primer procesador RISC producido en VLSI (*Very Large Integration Scale*)
  - Su enseñanza ha sido galardonada en múltiples ocasiones por el ACM, el IEEE y la Universidad de California

# RISC-V

- RISC-V, desarrollado originalmente en la Universidad de California en Berkeley, ofrece una versión simple, elegante y moderna de cómo deberían ser los repertorios de instrucciones actuales
- Más de 40 compañías se han unido a la Fundación RISC-V (riscv.org). La lista incluye a las más importantes, a excepción de ARM and Intel, incluyendo AMD, Google, Hewlett Packard Enterprise, IBM, Microsoft, NVIDIA, Oracle, and Qualcomm

## Diseño de un repertorio de instrucciones RISC

- Decisiones:
  - Tipo de almacenamiento de los operandos
  - Ordenación de los bytes
  - Alineamiento
  - Modos de direccionamiento soportados
  - Codificación de las instrucciones

# Tipo de almacenamiento de los operandos

- Los distintos repertorios se diferencian en el tipo de almacenamiento interno que utilizan:
  - Pila
  - Acumulador
  - Registros de propósito general (General Purpose Registers, GPR)

## Tipo de almacenamiento de los operandos: *Pila*

- Pila: los operandos son implícitos, siempre en la parte superior de la pila (Top of Stack, TOS)
  - No es necesario indicar dónde se encuentran los operandos

```
PUSH AX
PUSH BX
ADD
POP CX
```

## Tipo de almacenamiento de los operandos: *Registro Acumulador*

- Registro acumulador: uno de los operandos es implícito
  - El otro se debe especificar de forma explícita

```
LOAD A
ADD B
STORE C
```

## Tipo de almacenamiento de los operandos: *General Purpose Registers, GPR*

- Los operandos se especifican de forma explícita
  - **Registro-Registro** de 3 operandos, todos deben estar en registros
    - Se utilizan instrucciones de carga (load) y almacenamiento (store)
  - Registro-Memoria de 2 operandos, al menos uno de los operandos debe estar en registro
  - Memoria-Memoria de 2 o 3 operandos, todos ellos en memoria

## Tipo de almacenamiento de los operandos: *General Purpose Registers, GPR*

- Casi todas las arquitecturas se basan en GPR
- Los registros son más rápidos
- Son utilizados de manera mucho más eficiente por los compiladores
  - Almacenamiento temporal de variables

## Tipo de almacenamiento de los operandos: *General Purpose Registers, GPR*

- En arquitecturas *registro-registro*:
  - La codificación es sencilla, siempre hay que especificar el identificador de tres registros
  - Pero los programas ocupan más, ¿por qué?
- En arquitecturas *memoria-memoria*:
  - Código más compacto, menos instrucciones
  - La memoria, un cuello de botella, ¿por qué?
  - Grandes diferencias entre la longitud de las instrucciones y entre su duración
  - Se complica la codificación de las instrucciones y puede variar mucho el CPI (Ciclos Por Instrucción) entre instrucciones

# Tipo de almacenamiento de los operandos: *General Purpose Registers, GPR*

- **Ejercicio 1**. Vamos a evaluar dos alternativas:
  - Registro-Registro de 3 operandos con 8 registros de propósito general
  - Memoria-Memoria de 3 operandos
- Se desea realizar las siguientes operaciones:

  R = X AND Y

  Z = X OR Y

  Y = R AND X
- El *código de operación* ocupa 1 Byte
- Las direcciones de memoria y los operandos ocupan 4 Bytes
- Los operandos están almacenados inicialmente en memoria

# Tipo de almacenamiento de los operandos: *General Purpose Registers, GPR*

| Registro-Registro | Tráfico con la memoria de instrucciones | Tráfico con la memoria de datos |
|---|---|---|
| load r1,X | opcode + dir + registro = 6B | 1 operando = 4B |
| load r2,Y | opcode + dir + registro = 6B | 1 operando = 4B |
| and r3,r1,r2 | opcode + 3 registro = 3B | - |
| or r4,r1,r2 | opcode + 3 registro = 3B | - |
| and r2,r3,r1 | opcode + 3 registro = 3B | - |
| store r3,R | opcode + dir + registro = 6B | 1 operando = 4B |
| store r4,Z | opcode + dir + registro = 6B | 1 operando = 4B |
| store r2,Y | opcode + dir + registro = 6B | 1 operando = 4B |
| | Total, **39B** | Total, **20B** |

| Memoria-Memoria | Tráfico con la memoria de instrucciones | Tráfico con la memoria de datos |
|---|---|---|
| and R,X,Y | opcode + 3 direcciones = 13B | 3 operandos = 12B |
| or Z,X,Y | opcode + 3 direcciones = 13B | 3 operandos = 12B |
| and Y,R,X | opcode+3 direcciones=13B | 3 operandos = 12B |
| | Total, **39B** | Total, **36B** |

- Operaciones:

  R = X AND Y

  Z = X OR Y

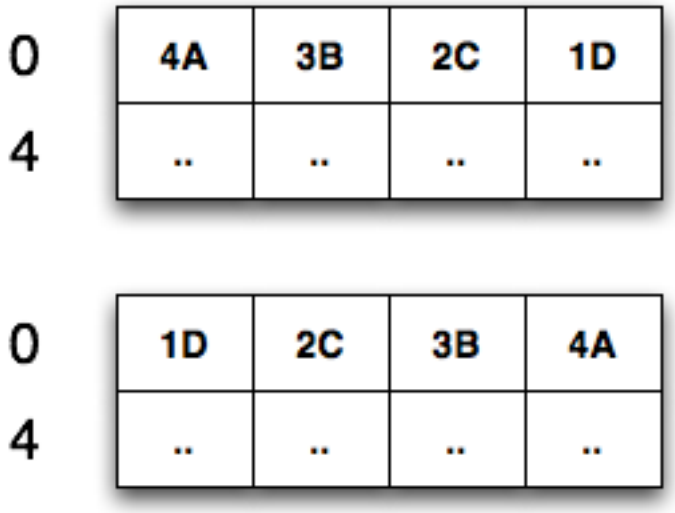
  Y = R AND X

# Ordenación

- La mayoría de las máquinas están direccionadas por bytes
  - Proporcionan acceso a bytes (8 bits), medias palabras (16 bits), palabras (32 bits) y dobles palabras (64 bits)
- **Little Endian**, "little-end-in", de comienzo por el extremo pequeño
  - Coloca el byte menos significativo en la posición más significativa de la palabra
  - La dirección de un dato es la del byte menos significativo
- **Big Endian**, "big-end-in", de comienzo por el extremo grande
  - Coloca el byte menos significativo en la posición menos significativa de la palabra
  - La dirección de un dato es la del byte más significativo
- **Middle Endian**, arquitectura capaz de trabajar con ambas ordenaciones, como por ejemplo los procesadores MIPS o Power PC

# Ordenación

- **Ejemplo**: el valor hexadecimal 0x4A3B2C1D se codificaría en memoria en la secuencia:
  - {4A, 3B, 2C, 1D} en *big-endian*
  - {1D, 2C, 3B, 4A} en *little-endian*



# Ordenación

- **Ejemplo**: tras la ejecución del siguiente código, en unas máquinas se imprime un mensaje y en otras otro. Debes indicar la ordenación de los datos en memoria teniendo en cuenta que un entero ocupa 16 bits y un char 8 bits. Además, debes indicar el valor de p[0] y p[1] en cada caso.

```
#include <stdio.h>
int main(void){
    int i = 1;
    char *p = (char *) &i;
    if ( p[0] == 1 )
        printf("Little Endian\n");
    else
        printf("Big Endian\n");
    return 0;
}
```

# Alineamiento

- En RISC-V, la información almacenada debe comenzar en direcciones múltiplo del tamaño de la información almacenada
- Estos accesos deben estar alineados, es decir, un acceso a una información de *s* bytes en la dirección del byte *B* está alineado si:

$$B \text{ módulo } s = 0$$

- Estas restricciones de alineamiento se deben a que las memorias, físicamente, están diseñadas para hacer accesos alineados
- Un acceso no alineado o mal alineado, supone varios accesos alineados a la memoria

# Alineamiento

```
        .data
str:    .asciiz “Hola”
num:    .word 10
```

- Segmento de datos con acceso alineado:
  - Por cada referencia a memoria para recuperar *num*, un solo acceso

| Dirección | Contenido bytes +0 +1 +2 +3 | Contenido bytes +4 +5 +6 +7 |
|---|---|---|
| 0 | 61 6C 6F 48 | 00 00 00 00 |
| 8 | 00 00 00 0A | 00 00 00 00 |

- Segmento de datos con acceso NO alineado:
  - Por cada referencia a memoria para recuperar *num*, dos accesos

| Dirección | Contenido bytes +0 +1 +2 +3 | Contenido bytes +4 +5 +6 +7 |
|---|---|---|
| 0 | 61 6C 6F 48 | 00 00 00 00 |
| 8 | 0A 00 00 00 | 00 00 00 00 |

# Hexadecimal

- Base 16
  - 4 bits por cada dígito hexadecimal

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | C | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | D | 1101 |
| 2 | 0010 | 6 | 0110 | A | 1010 | E | 1110 |
| 3 | 0011 | 7 | 0111 | B | 1011 | F | 1111 |

- Ejemplo: ECA8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

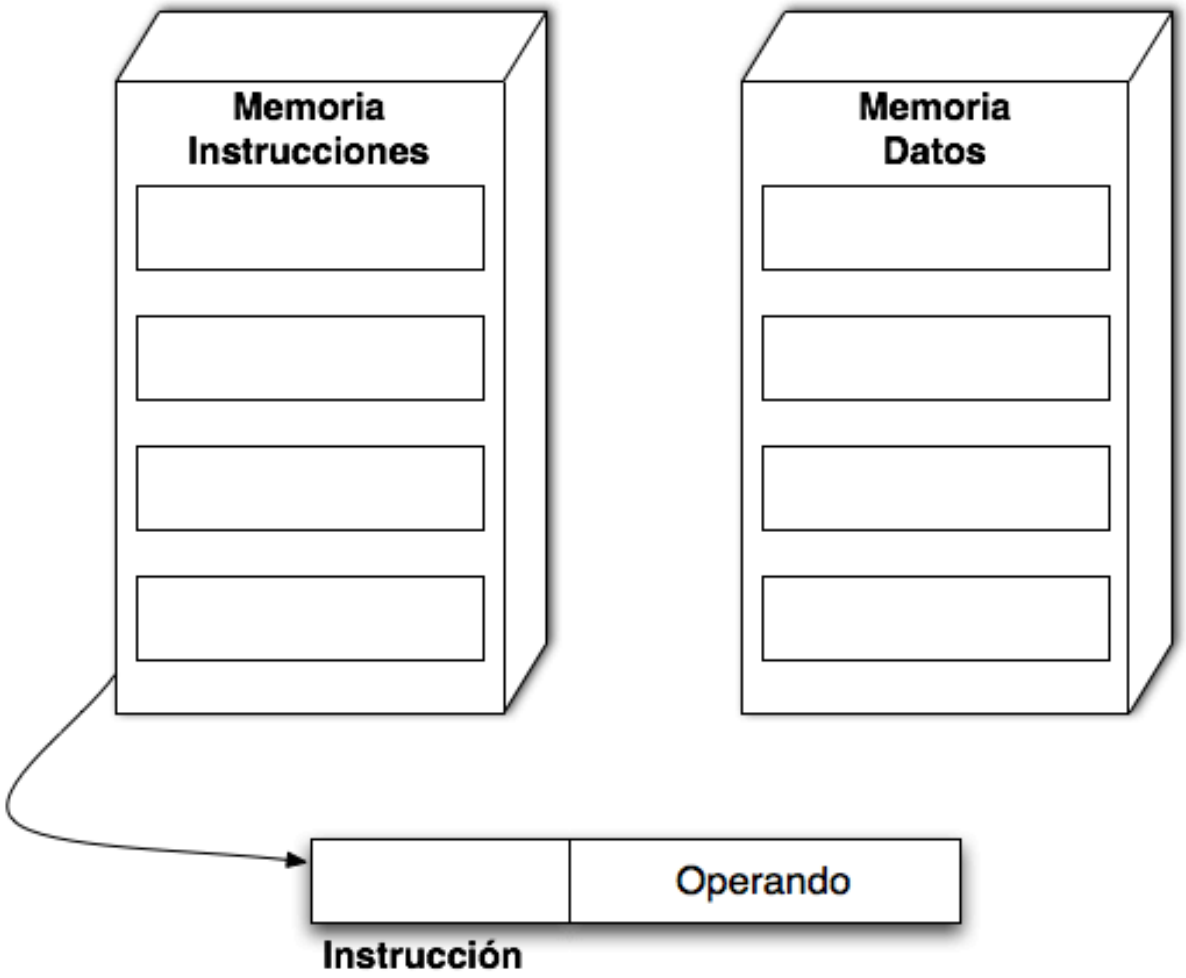
# Modos de direccionamiento

- Los modos de direccionamiento determinan la ubicación de los operandos
- ¿Dónde se pueden ubicar los operandos?
  - En la propia instrucción
  - En un registro
  - En memoria principal

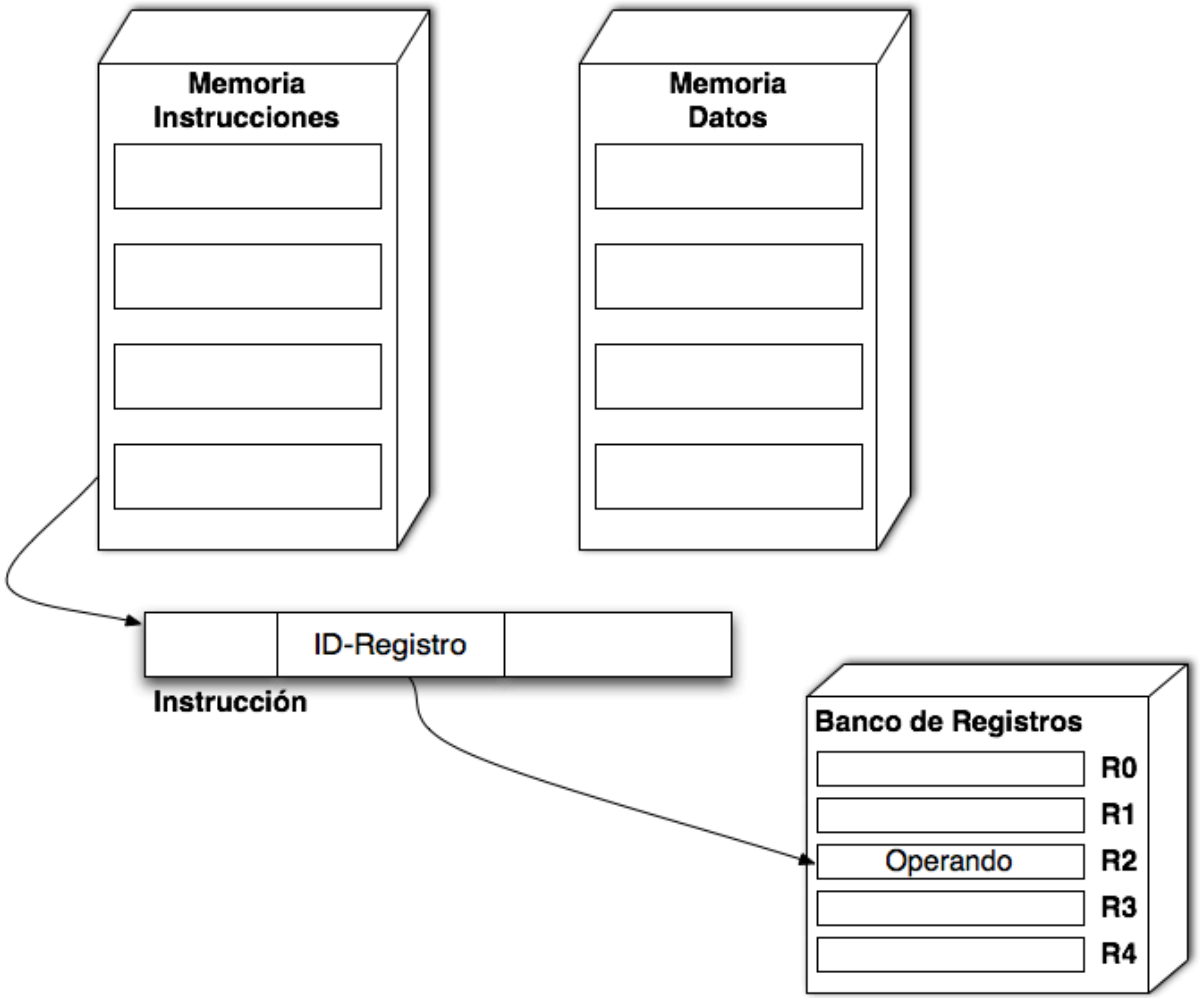
# Modos de direccionamiento básicos

- **Direccionamiento inmediato**:
  - el operando se codifica dentro de la instrucción
- **Direccionamiento a registro**:
  - el operando se encuentra en un registro
  - se incluye el identificador del registro que amacena el operando.
- Si el operando se ubica en memoria:
  - **Directo** o **Absoluto**: se incluye la dirección de memoria en la que está almacenado el operando
  - **Indirecto**: se indica el registro que almacena la dirección de memoria en la que se encuentra el operando
  - **Indirecto con desplazamiento**: se suma un operando inmediato al contenido del registro para obtener la dirección de memoria en la que se encuentra el operando

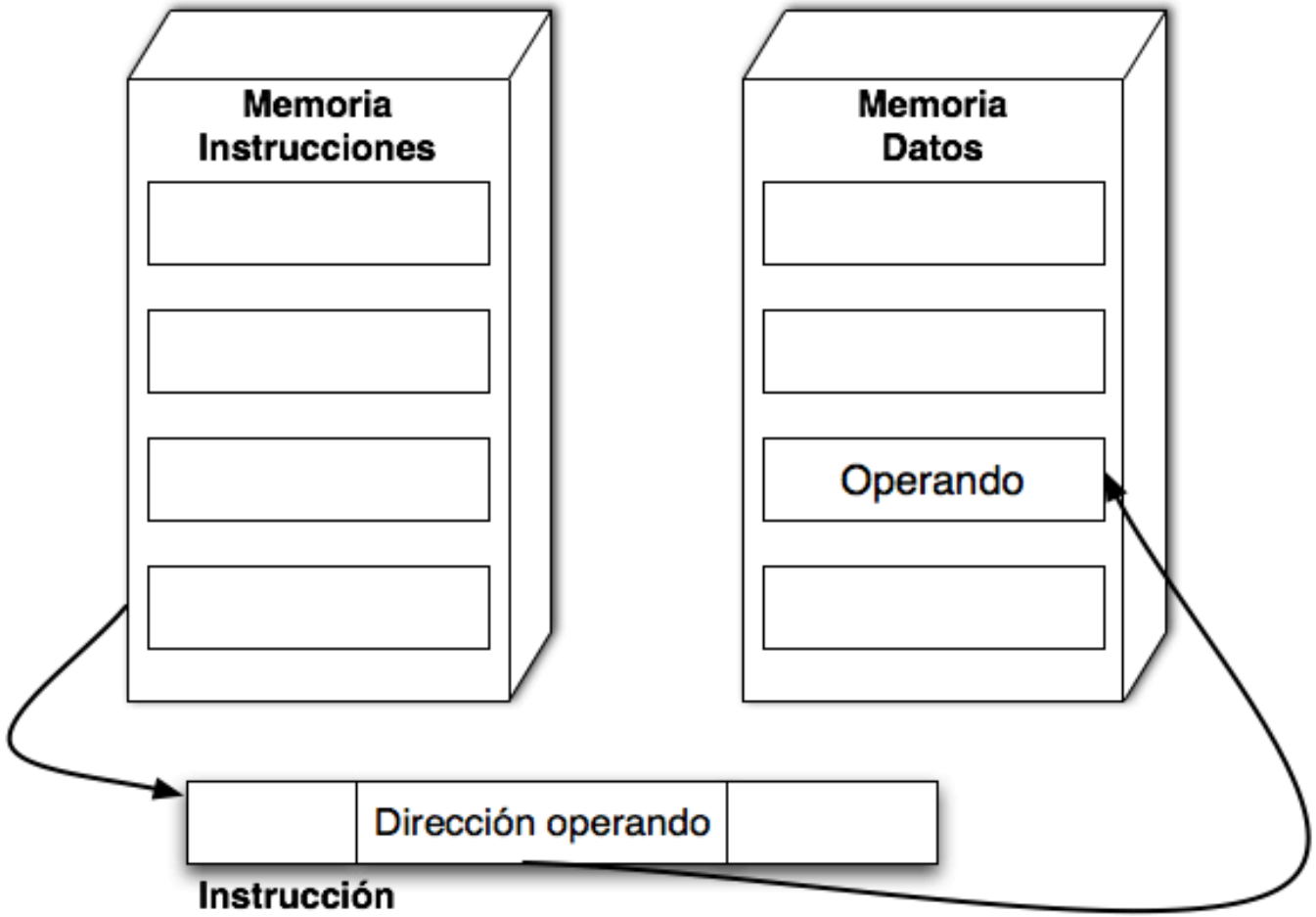
# Modos de direccionamiento básicos:
# - *Direccionamiento inmediato*



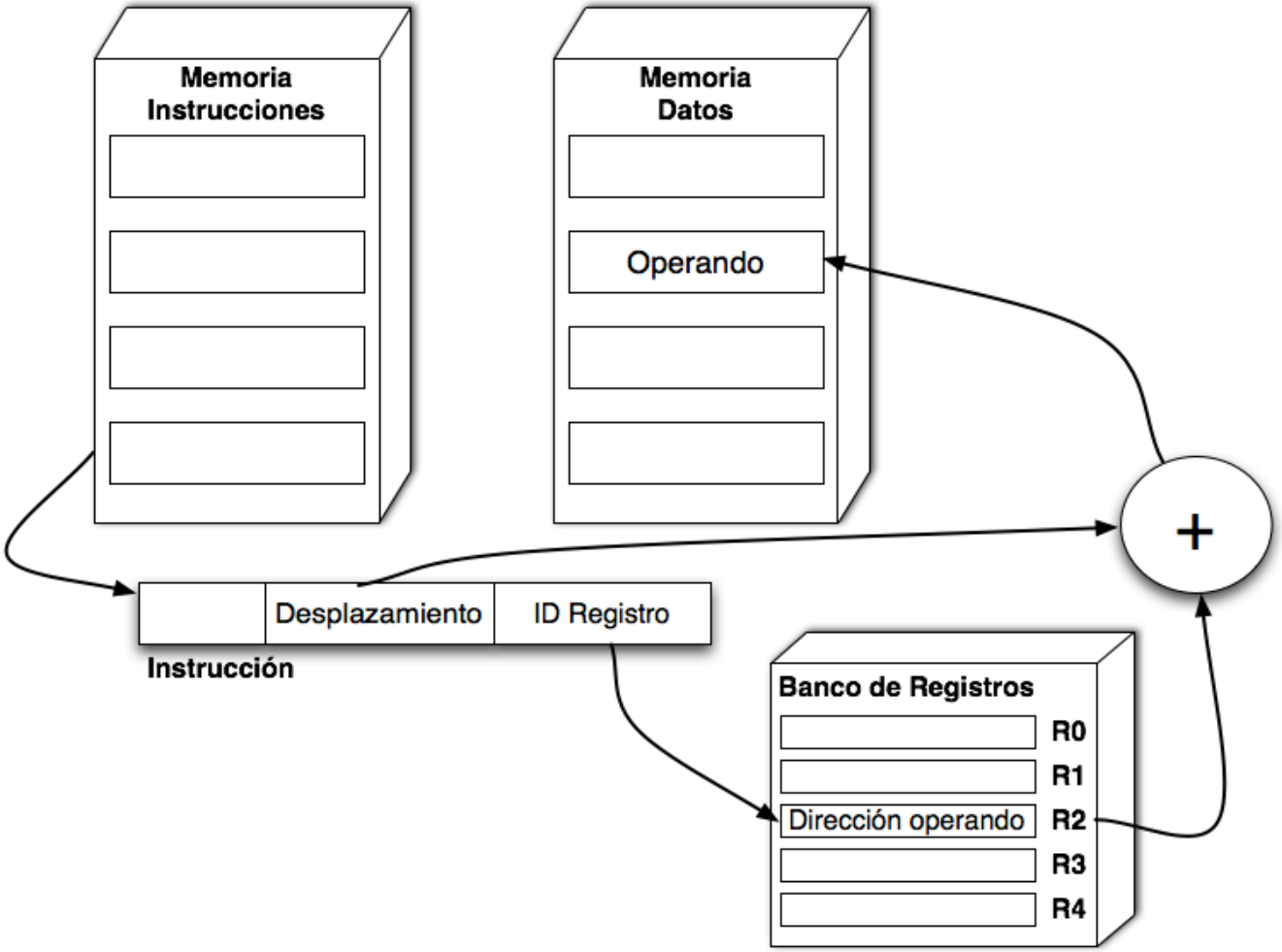
# Modos de direccionamiento básicos:
# *- Direccionamiento a registro*



# Modos de direccionamiento básicos:
# *- Direccionamiento directo*



# Modos de direccionamiento básicos:
## *- Direccionamiento indirecto con desplazamiento*



# Modos de direccionamiento en un repertorio RISC

- Los repertorios RISC incluyen como mínimo direccionamiento inmediato e indirecto con desplazamiento
- **Direccionamiento inmediato**, a la hora de diseñar el repertorio hay que decidir sí:
  1. Todas las instrucciones deben soportar este modo o sólo un subconjunto
  2. El rango de valores del operando inmediato, ¿en qué influye esto?
- **Direccionamiento indirecto con desplazamiento**: la decisión más importante consiste en determinar el rango de valores que puede tomar el desplazamiento

# Otras consideraciones

- **Tipo y tamaño de los operandos**:
  - ¿Qué tipo de datos se soportan? ¿Con qué tamaños? Carácter, entero, coma flotante, etc
  - El código de operación, opcode, indicará el tipo de los operandos implicados en la ejecución de la instrucción
  - … también lo pueden indicar los operandos mediante etiquetas, ¿inconvenientes?
- **Conjunto de operaciones soportadas**:
  - ¿Qué tipo de operaciones van a realizar las instrucciones del repertorio?
  - Un conjunto sencillo: aritmético-lógicas (ALU), de acceso a memoria, de control de flujo (saltos) y llamadas al sistema operativo
- **Tratamiento de las instrucciones de control de flujo**: modifican el flujo de control de un código

# Instrucciones control de flujo

- **Saltos condicionales**:
  - ¿Cómo se especifica la condición?
  - ¿Cómo se indica la dirección destino de salto?
- **Saltos incondicionales**:
  - ¿Cómo se indica el destino?
- **Direccionamiento relativo al PC, *branch***
  - Se conoce el destino de salto en tiempo de compilación
  - Los destinos de los saltos están cercanos al salto
  - Código reubicable
  - ¿Cuántos bits se necesitan para el desplazamiento?
- **Direccionamiento indirecto con registro, *jump***
  - No se conoce la dirección de salto o su valor excede del que se puede indicar con el desplazamiento
  - Se indica el identificador del registro que contiene la dirección destino de salto

# Codificación del repertorio de instrucciones

- **Longitud variable**
  - Soporta cualquier número de operandos y cualquier combinación instrucción/modo de direccionamiento
  - ***Etiquetas*** que indican el modo
  - Se añaden tantos campos como sean necesarios + las etiquetas que permiten su interpretación
- **Longitud fija**
  - El **código de operación** especifica el modo de direccionamiento
  - Sólo se permiten unas combinaciones determinadas de operaciones + modos
  - Los campos de la instrucción son siempre los mismos
- **Híbrida**
  - Longitud fija
  - Sólo se permiten unos determinados formatos de instrucción, que incluyen un número variable de modos y operandos

# RISC-V: generalidades

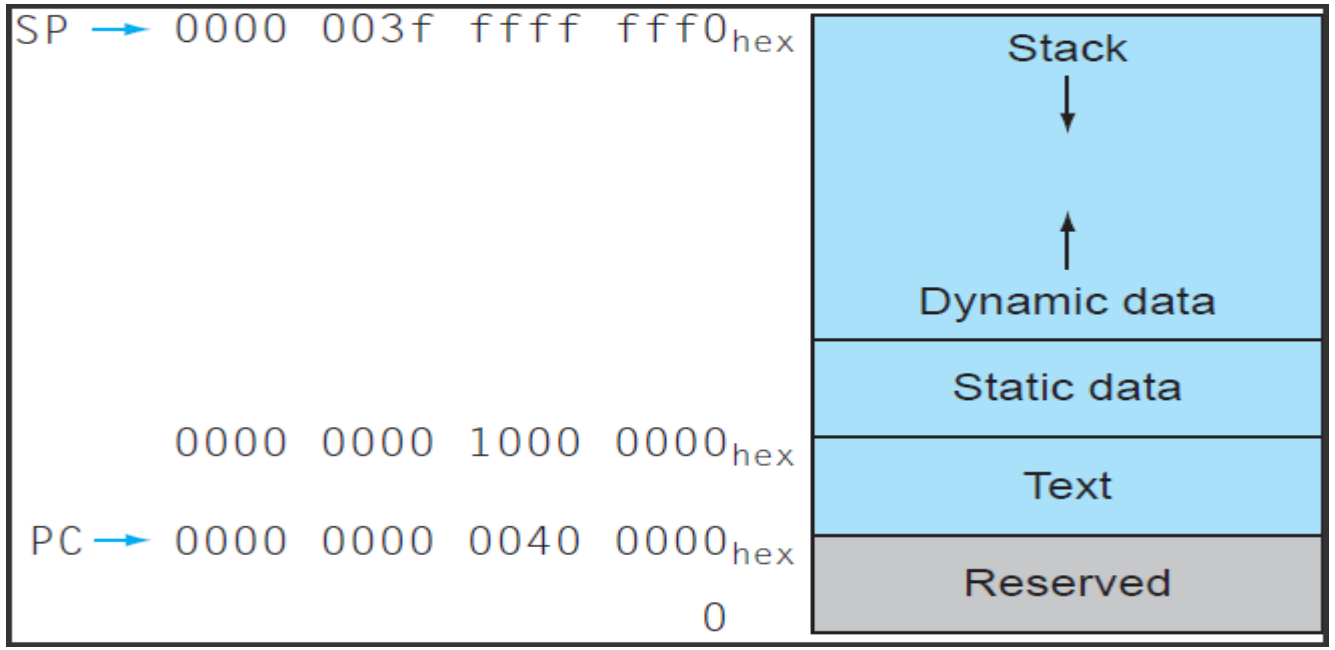
- RICS-V, arquitectura RISC basada en **registros de propósito general** de tipo carga/ almacenamiento
- Los **operandos** de una instrucción siempre deben estar almacenados en registros dentro del procesador (no puedan estar en memoria)
- Los resultados siempre se devuelven a registros dentro del procesador
- La arquitectura del RISC-V se basa en un juego de instrucciones de longitud fija
- **32 registros de propósito general** de 64 bits

# RISC-V: banco de registros

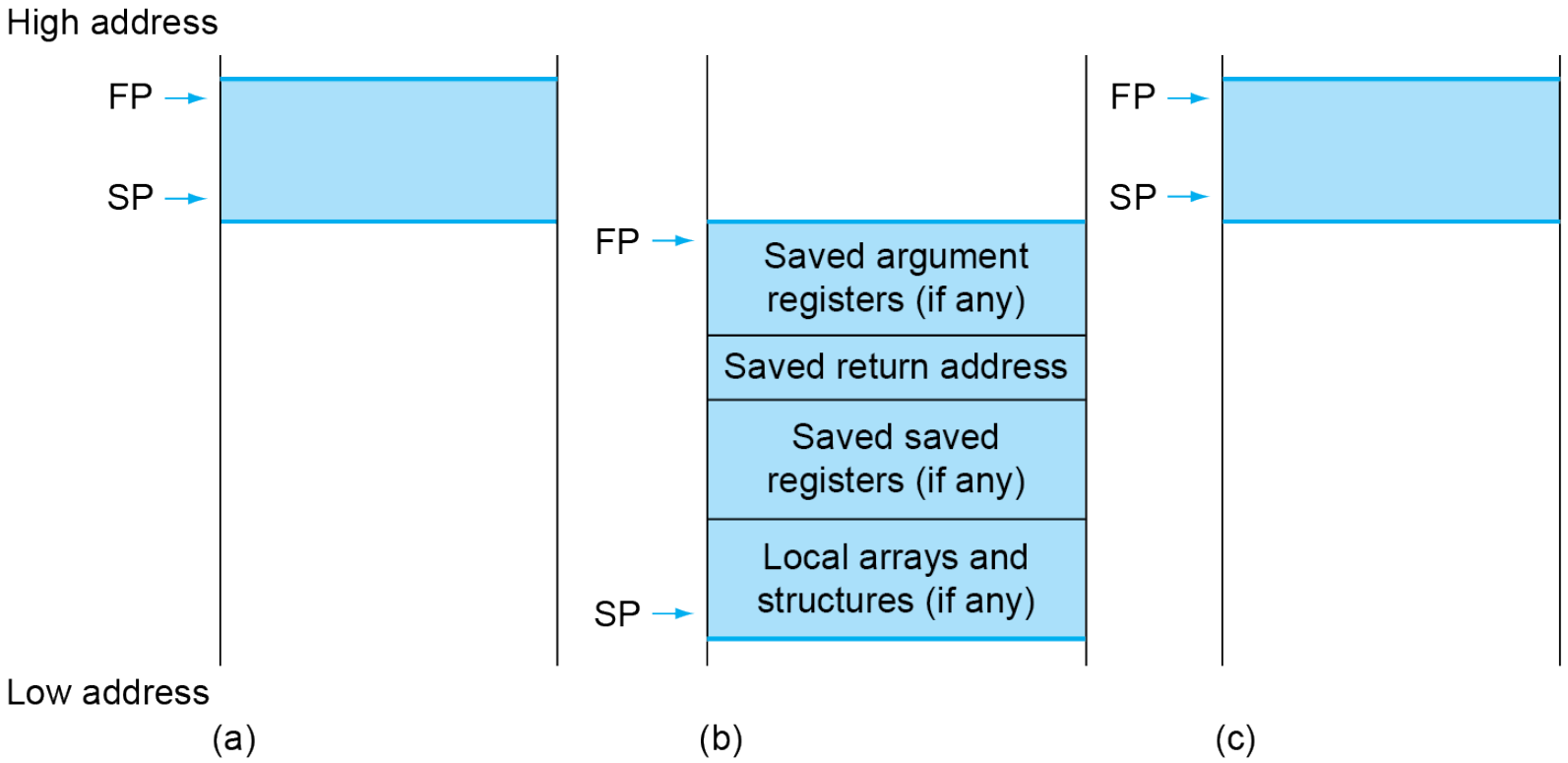
| Name | Register number | Usage | Preserved on call? |
|---|---|---|---|
| x0 | 0 | The constant value 0 | n.a. |
| x1 (ra) | 1 | Return address (link register) | yes |
| x2 (sp) | 2 | Stack pointer | yes |
| x3 (gp) | 3 | Global pointer | yes |
| x4 (tp) | 4 | Thread pointer | yes |
| x5-x7 | 5–7 | Temporaries | no |
| x8-x9 | 8–9 | Saved | yes |
| x10-x17 | 10–17 | Arguments/results | no |
| x18-x27 | 18–27 | Saved | yes |
| x28-x31 | 28–31 | Temporaries | no |

# RISC-V: diseño de la memoria

- **Text**: almacena el código del programa (código ejecutable)
- **Static** data: almacena variables declaradas para las que se reserva memoria en tiempo de compilación
- **Heap**: almacena datos para los que no se ha reservado memoria en tiempo de compilación, son datos creados en tiempo de ejecución
  - E.g., malloc in C, new in Java
- **Stack**: invocación de subrutina



# RISC-V: datos locales *pila*



# RISC-V: datos locales *pila*

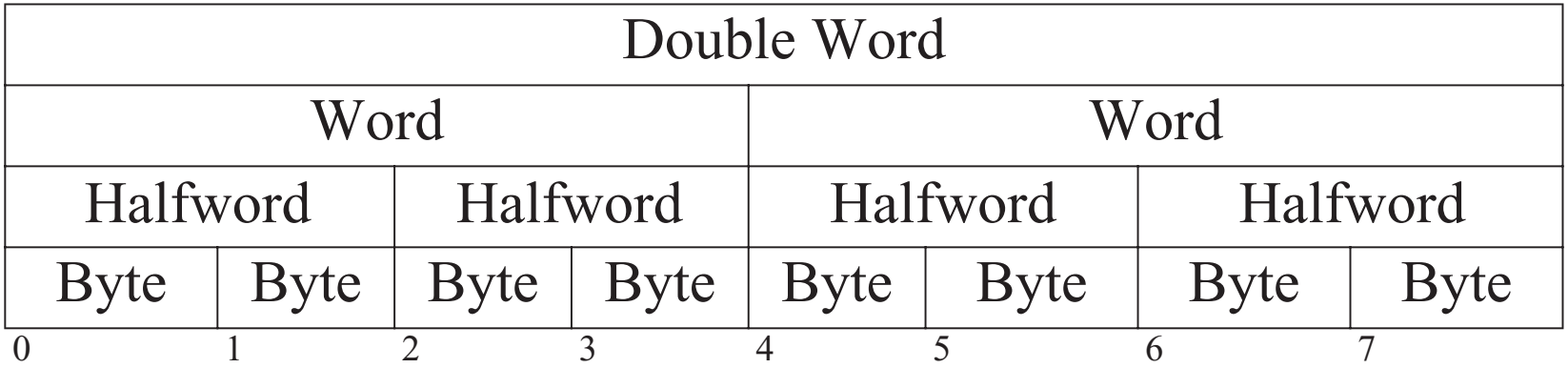
- Convenio de llamada a subrutina

| Preserved | Not preserved |
|---|---|
| Saved registers: x8-x9, x18-x27 | Temporary registers: x5-x7, x28-x31 |
| Stack pointer register: x2(sp) | Argument/result registers: x10-x17 |
| Frame pointer: x8(fp) | |
| Return address: x1(ra) | |
| Stack above the stack pointer | Stack below the stack pointer |

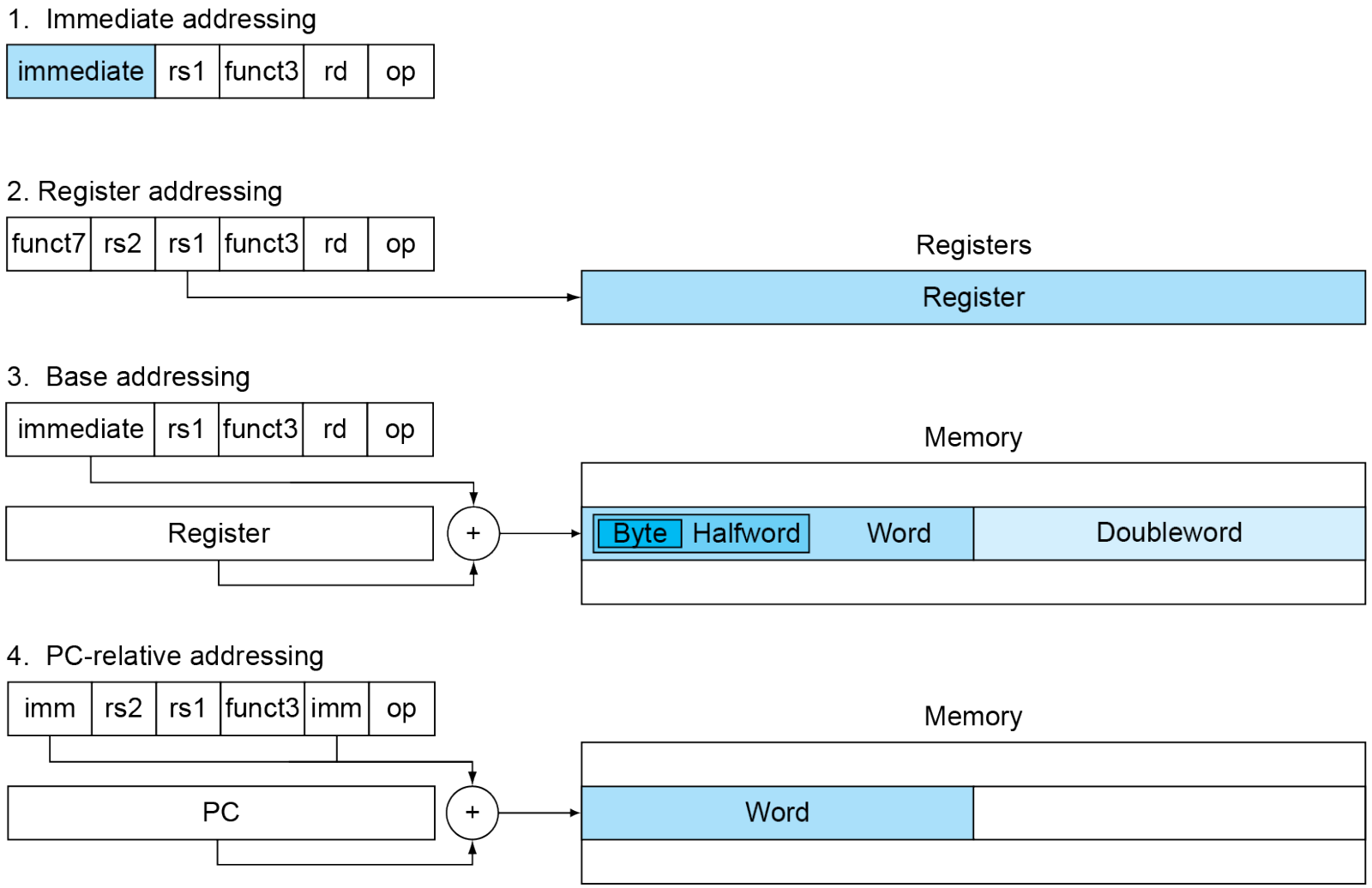
# RISC-V: alineamiento de los datos

- **Byte:** 1 byte, 8 bits, ***b***
- **Halfword:** media palabra, 2 bytes, 16 bits, ***h***
- **Word:** palabra, 4 bytes, 32 bits, ***w***
- **Double Word:** doble palabra, 8 bytes, 64 bits, ***d***

**DATA ALIGNMENT**



# RISC-V: modos de direccionamiento



# RISC-V: tipos de instrucciones

| Name (Field Size) | Field 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits | Comments |
|---|---|---|---|---|---|---|---|
| R-type | funct7 | rs2 | rs1 | funct3 | rd | opcode | Arithmetic instruction format |
| I-type | immediate[11:0] | | rs1 | funct3 | rd | opcode | Loads & immediate arithmetic |
| S-type | immed[11:5] | rs2 | rs1 | funct3 | immed[4:0] | opcode | Stores |
| SB-type | immed[12,10:5] | rs2 | rs1 | funct3 | immed[4:1,11] | opcode | Conditional branch format |
| UJ-type | immediate[20,10:1,11,19:12] | | | | rd | opcode | Unconditional jump format |
| U-type | immediate[31:12] | | | | rd | opcode | Upper immediate format |

# Operaciones aritméticas

- Código C:

```
a = b + c + d + e;
```

- RISC-V emplea una notación rígida:
  - Cada instrucción aritmética realiza una única operación
  - Siempre tiene tres operandos: dos fuente y uno destino
- Si queremos sumar b, c, d y e en a:

```
add a, b, c
add a, a, d
add a, a, e
```

- Cada línea de este lenguaje contiene una única instrucción

# Operaciones aritméticas

- El hardware para un número variable de operandos es más complicado que el hardware para un número fijo de operandos
- Cuatro principios de diseño del hardware
- ***Principio de diseño I***:
  - La regularidad favorece la simplicidad
  - La simplicidad permite un mayor rendimiento a un menor coste

# Operaciones aritméticas

- Código C:

```
f = (g + h) - (i + j);
```

- Código compilado RISC-V:

```
add t0, g, h   // temp t0 = g + h

add t1, i, j   // temp t1 = i + j

sub f, t0, t1  // f = t0 - t1
```

# Operandos en registros

- Los **operandos** de las instrucciones aritméticas son limitados y se corresponden físicamente con una parte del hardware del computador denominada **registros**
- Los registros son una parte del hardware visible a los programadores

# Operandos en registros

- A diferencia de las variables de un programa, los registros tienen un número limitado
- En MIPS hay 32 registros
- Esta limitación en el número de registro viene impuesta por el:
- ***Principio de diseño II:***
  - Más pequeño es más rápido
- Un mayor número de registros supone incrementar el ciclo de reloj puesto que a las señales electrónicas les lleva más tiempo recorrer distancias más largas

# Registros de RISC-V

- **x0**: valor constante 0, registro cableado a 0
- x1: dirección de retorno
- **x2**: puntero de pila
- x3: puntero a zona de varibales estáticas, global
- x4: puntero de hilo
- **x5 – x7, x28 – x31**: registros temporales
- **x8**: puntero de marco de pila
- **x9, x18 – x27**: registros estáticos
- **x10 – x11**: argumentos subrutina/valor retorno subrutina
- **x12 – x17**: argumentos subrutina

# Registros de RISC-V

| Name | Number | Value |
|---|---|---|
| zero | 0 | 0x00000000 |
| ra | 1 | 0x00000000 |
| sp | 2 | 0x7fffeffc |
| gp | 3 | 0x10008000 |
| tp | 4 | 0x00000000 |
| t0 | 5 | 0x00000000 |
| t1 | 6 | 0x00000000 |
| t2 | 7 | 0x00000000 |
| s0 | 8 | 0x00000000 |
| s1 | 9 | 0x00000000 |
| a0 | 10 | 0x00000000 |
| a1 | 11 | 0x00000000 |
| a2 | 12 | 0x00000000 |
| a3 | 13 | 0x00000000 |
| a4 | 14 | 0x00000000 |
| a5 | 15 | 0x00000000 |
| a6 | 16 | 0x00000000 |
| a7 | 17 | 0x00000000 |
| s2 | 18 | 0x00000000 |
| s3 | 19 | 0x00000000 |
| s4 | 20 | 0x00000000 |
| s5 | 21 | 0x00000000 |
| s6 | 22 | 0x00000000 |
| s7 | 23 | 0x00000000 |
| s8 | 24 | 0x00000000 |
| s9 | 25 | 0x00000000 |
| s10 | 26 | 0x00000000 |
| s11 | 27 | 0x00000000 |
| t3 | 28 | 0x00000000 |
| t4 | 29 | 0x00000000 |
| t5 | 30 | 0x00000000 |
| t6 | 31 | 0x00000000 |
| pc | 32 | 0x00400000 |

# Ejemplo operandos en registros

- Código C:

```
f = (g + h) - (i + j);
```

  - f, g, h, i y j se asignan a s0, s1, s2, s3 y s4

- Código compilado RISC-V:

```
add t0, s1, s2
add t1, s3, s4
sub s0, t0, t1
```

# Operandos en memoria

- Memoria principal para datos complejos
  - Arrays, estructuras, datos dinámicos
- Para poder realizar operaciones aritméticas:
  - Cargar/**L**oad los valores de memoria a registros
  - Almacenar/**S**tore el resultado de registro a memoria
- La memoria se direcciona por bytes
  - Cara dirección identifica un 8-bit byte
- RISC-V es Little Endian
  - Byte menos significativo en dirección más significativa
- En RISC-V no es obligatorio que los datos estén alineados en memoria
  - A diferencia de otros ISAs

# Ejemplo operandos en memoria

- Código C:

  `A[12] = h + A[8];`

  - h se asigna a s2
  - la dirección base de A se asigna a s3
- Código compilado RISC-V:
  - El elemento 8 tiene un *offset* de 64 bytes con respecto a la dirección base del array A
    - 8 bytes por doble palabra, *doubleword*

```
ld    t0,64(s3) # t0 = A[8]
add   t0,s2,t0  # t0 = h + A[8]
sd    t0,96(s3) # A[12] = h + A[8]
```

# Ejercicio 1

- Código C:

  `A[12] = h + A[8];`

  - h se asigna a s2
  - la dirección base de A se asigna a s3
- ¿Cuál será el código compilado RISC-V correspondiente si los elementos del array son de tamaño palabra, media palabra y byte?

# Ejercicio 2

- Código C:

  `A[12] = h + A[i];`

  - i se asigna a s1
  - h se asigna a s2
  - la dirección base de A se asigna a s3
- ¿Cuál será el código compilado RISC-V correspondiente si los elementos del array son de tamaño doble palabra, palabra, media palabra y byte?

# Ejercicio 3

- Código C:

  ```
  B[12] = h - A[i];
  ```

  - i se asigna a s1
  - h se asigna a s2
  - la dirección base de A se asigna a s3
  - la dirección base de B se asigna a s4
- ¿Cuál será el código compilado RISC-V correspondiente si los elementos del array son de tamaño doble palabra, palabra, media palabra y byte?

# Registros Vs Memoria

- Los registros son más rápidos que la memoria
  - Se necesita menos energía para acceder a los registros que a la memoria
- Operar con datos en memoria requiere de load y store
  - Se debe ejecutar un mayor número de instrucciones más lentas
- El compilador debe usar los registros todo lo posible
  - Usar memoria solo para variables poco utilizadas
  - La optimización de los registros es importante

# Operandos inmediatos

- Con las instrucciones que hemos visto hasta ahora …

```
lw  t4,AddrCons4(s1) # t4 = 4
add s3, s3, t4       # s3 = s3 + t4
```

- … en su lugar existe una versión de todas las instrucciones aritméticas en la que uno de los operandos es una constante

```
addi s3, s3, 4       # s3 = s3 + 4
```

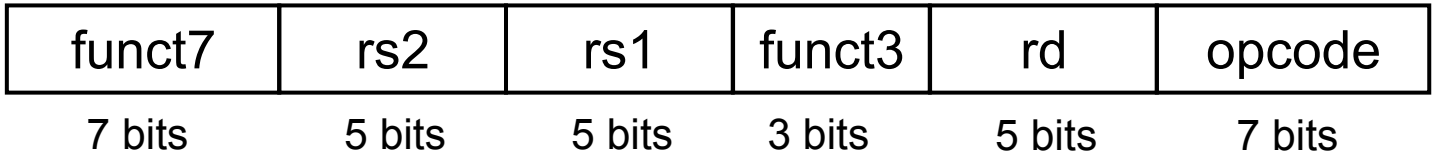
# Operandos inmediatos

- Las instrucciones inmediatas ilustran el tercer principio de diseño
- ***Principio de diseño III:***
  - Haz que el caso más común sea rápido
- La constante cero tiene el rol de simplificar el juego de instrucciones ofreciendo variaciones útiles
- RISC-V tiene un registro cableado a cero, se trata del registro zero que se corresponde con el registro número 0

# Formatos de instrucción

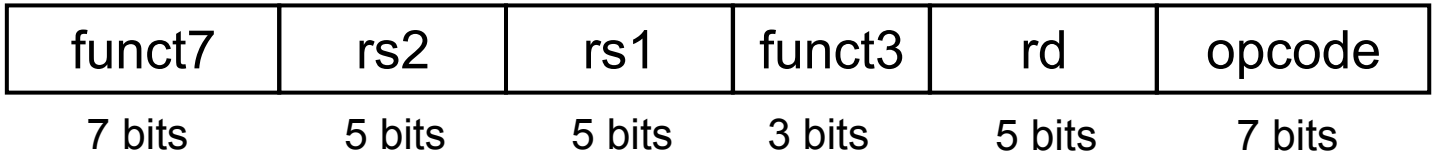
- *¿Qué pasa si una instrucción necesita campos más largos*? Conflicto entre el deseo de que todas las instrucciones tengan la misma longitud y el deseo de tener un único formato de instrucción
- ***Principio de diseño IV**:*
  - Un buen diseño demanda buenos compromisos
- El compromiso elegido por los diseñadores del RISC-V es mantener todas las instrucciones con la misma longitud, pero requiere de distintos formatos de instrucción para distintos tipos de instrucciones

# Instrucciones tipo R

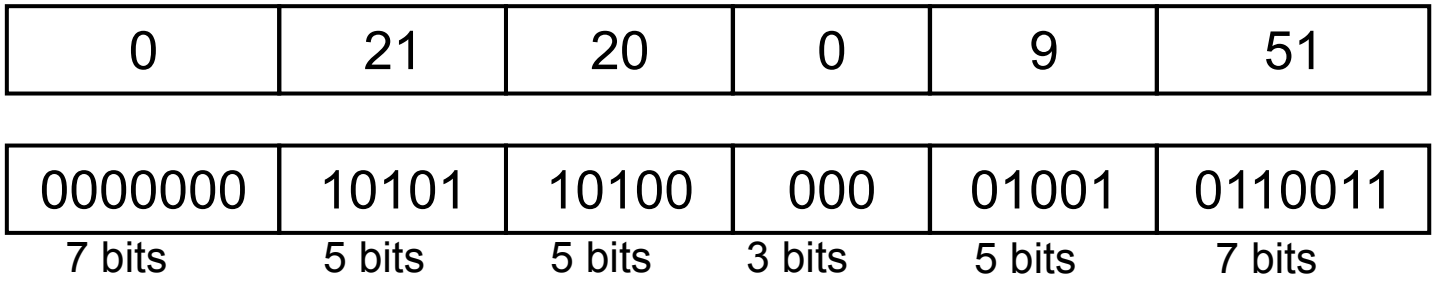



- Aritmético-lógicas registro-registro
  - **opcode**: código de operación
  - **rd**: número del resgistro destino
  - **funct3**: 3-bit código de función (adicional al código de operación)
  - **rs1**: número del primer registro fuente
  - **rs2**: número del segundo registro fuente
  - **funct7**: 7-bit código de función (adicional al código de operación)

# Ejemplo instrucción tipo R

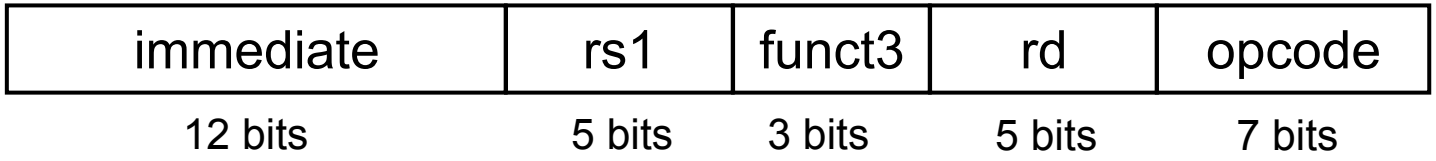



```
add x9,x20,x21
```



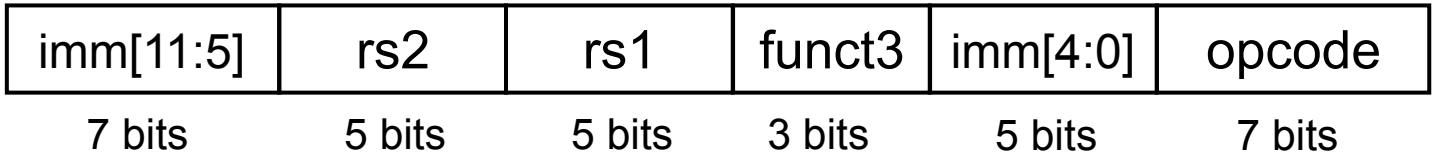

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

# Instrucciones tipo I



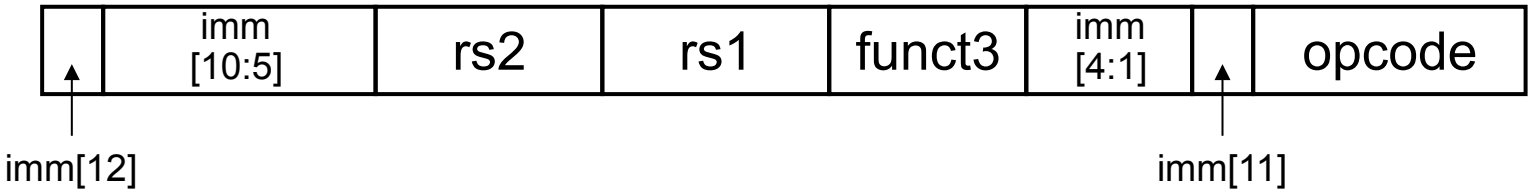

- Load
  - **rs1**: número del registro fuente, dirección base para el acceso a memoria
  - **rd**: número del resgistro destino
  - **immediate**: desplazamiento para el cálculo de la dirección de memoria a la que hay que acceder
- Aritmético-lógicas con operando inmediato
  - **rs1**: número de registro del primer operando fuente
  - **rd**: número del resgistro destino
  - **immediate**: valor del segundo operando Fuente
  - **funct3**: código de función de 3-bits (adicional al código de operación)

# Instrucciones tipo S




- Formato inmediato para instrucciones store
  - **opcode**: código de operación
  - **rs1**: número del registro fuente, dirección base para el acceso a memoria
  - **rs2**: número del resgistro destino
  - **immediate**: desplazamiento para el cálculo de la dirección de memoria a la que hay que acceder
    - Campo partido, para que los campos rs1 y rs2 siempre estén en el mismos sitio
  - **funct3**: código de función de 3-bits (adicional al código de operación)

# Instrucciones tipo SB




- Salto (cercano) condicional hacia delanta o hacia atrás
- Salto relativo al PC
  - **Dirección destino de salto = PC + immediate × 2**
  - **opcode**: código de operación
  - **rs1**: número del registro fuente, dirección base para el acceso a memoria
  - **rs2**: número del resgistro destino
  - **immediate**: desplazamiento para el cálculo de la dirección de memoria a la que hay que acceder
    - Campo partido, para que los campos rs1 y rs2 siempre estén en el mismos sitio
  - **funct3**: código de función de 3-bits (adicional al código de operación)