Tema 5:

Arquitectura del procesador

Fundamentos de computadores

Daniel Mozos Muñoz
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Índice

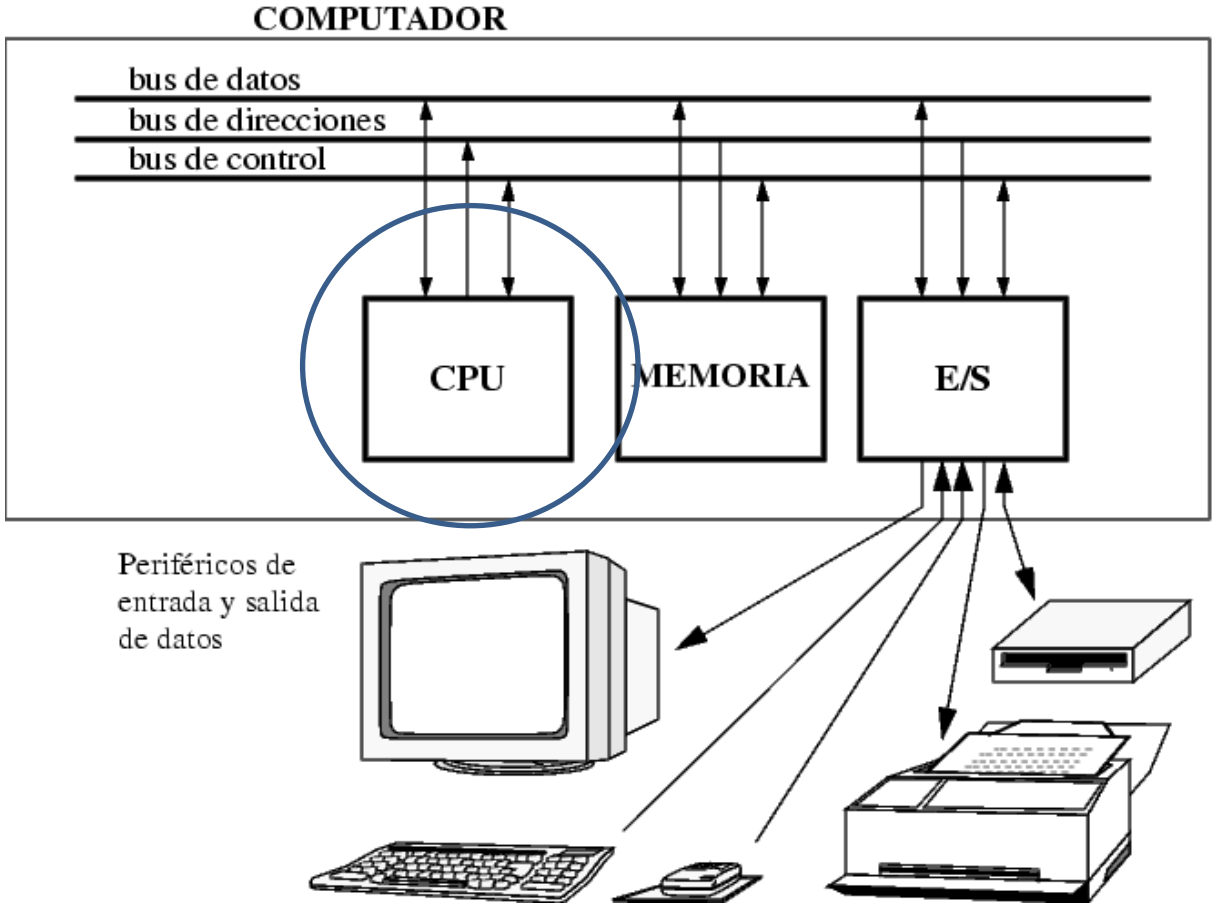
- Arquitectura del procesador: Decisiones de diseño
- RISC-V:
  - Almacenamiento
  - Instrucciones
  - Modos de direccionamiento
- Programación en ensamblador
- Acceso a datos
- Sentencias condicionales y bucles
- Llamadas a función
- Instrucciones en Punto flotante

Bibliografía:

•D.A. Patterson y J.L. Hennessy. *Computer Organization and Design. The Hardware/software interface. RISC-V edition, Morgan Kaufmann, 2018*

•*S.L. Harris y D.M. Harris. Digital design and computer architecture. RISC-V edition. Morgan Kaufmann, 2021*

¿Qué vamos a estudiar en este tema?



Modelo Von Neumann

Modelo de memoria

- En el **Modelo Von Neuman** la memoria almacena:
  - Instrucciones: programa almacenado
  - Datos
- Puede ser modelada como una secuencia de bytes (*tabla*)
  - Cada ***byte*** tiene asignada una **dirección** de memoria
  - Si disponemos de *k* bits para la dirección podremos acceder a 2^k bytes
  - **Palabra:** parámetro arquitectónico, típicamente 8, 16, 32 ó 64 bits
  - La memoria suele estar optimizada para transferencias de tamaño palabra o múltiplo de ella

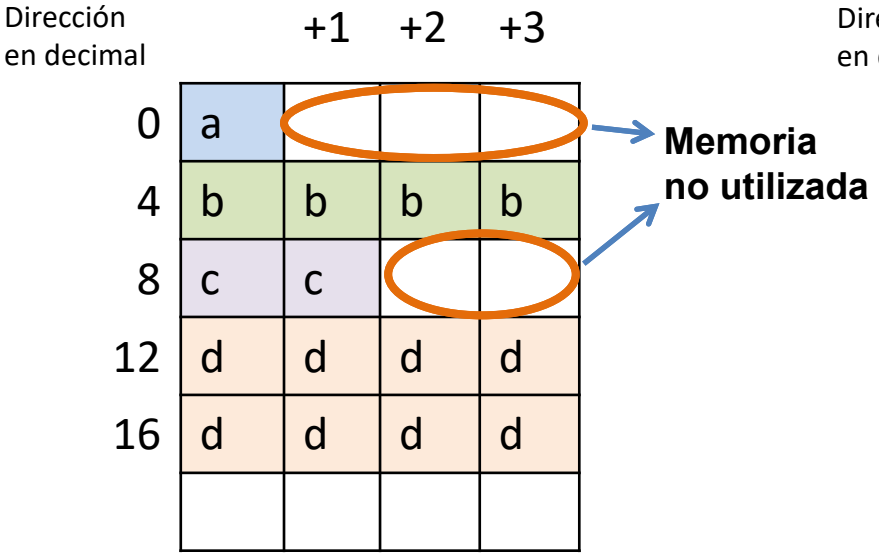
Alineamiento en memoria

- Muchas arquitecturas imponen restricciones a las direcciones donde pueden estar situadas las palabras en memoria
  - Un acceso a una palabra en memoria de tamaño N bytes debe hacerse sobre una dirección múltiplo de N
  - Limita las posibles direcciones de variables
    - **1 byte** (char): puede colocarse en **cualquier dirección**
    - **2 bytes** (short int): en direcciones **múltiplo de 2**
    - **4 bytes** (int, float, instrucciones): en direcciones **múltiplo de 4**
    - **8 bytes** (double float): en direcciones **múltiplo de 8**

Alineamiento de variables

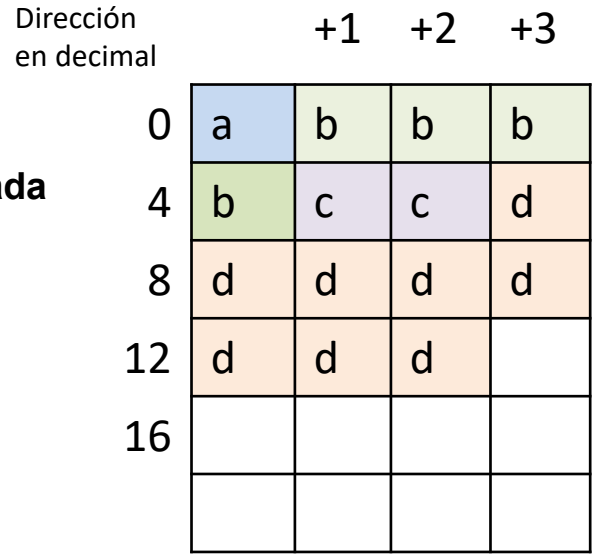
- Ejemplo. Declaramos cuatro variables:

  char a; int b; short int c; double d;



Variables alineadas

Direcciones de comienzo de las variables:
a => 0
b => 4
c => 8
d => 12



Variables no alineadas

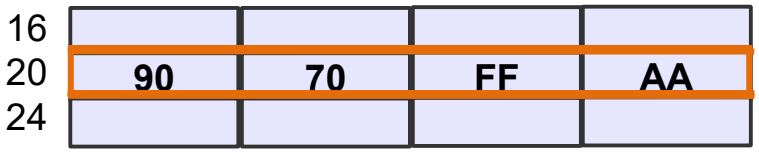
Direcciones de comienzo de las variables:
a => 0
b => 1
c => 5
d => 7

Orden de bytes en memoria

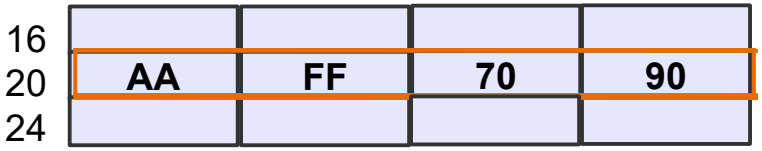
- **Ejemplo:** dato de 4 bytes (0x9070FFAA) almacenado en la dirección 20.

El dato ocupará los bytes 20, 21, 22 y 23. ¿Qué byte se pone en la dirección 20, cuál en la dirección 21, …?

- **BIG-ENDIAN**: La dirección de la variable coincide con la dirección del **byte MÁS significativo**
- **LITTLE-ENDIAN**: La dirección de la variable coincide con la dirección del **byte MENOS significativo**
- La mayor parte de los procesadores actuales pueden ser configurados para funcionar como little o big endian



BIG-ENDIAN



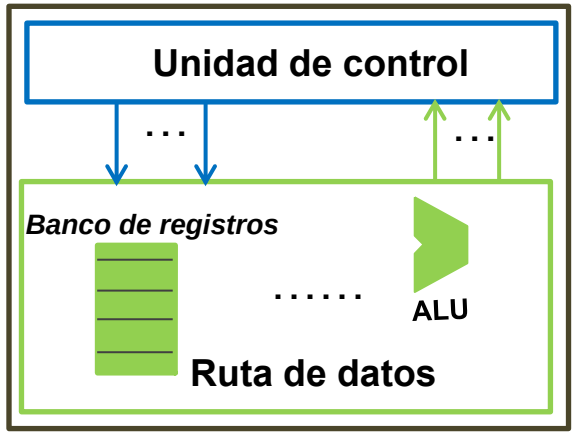
LITTLE-ENDIAN

CPU

- La CPU es un sistema digital cuya funcionalidad podríamos describir con la siguiente secuencia cíclica:

1. Leer desde memoria la siguiente instrucción
2. Decodificar la instrucción
3. Obtener los operandos de la instrucción
4. Ejecutar la instrucción (realizar la operación)
5. Almacenar el resultado
6. Volver a 1

CPU



- **Unidad de proceso (Ruta de datos o Data-path)**
  - Circuito que realiza las operaciones que indican las instrucciones del programa
  - Está formada por módulos combinacionales y secuenciales
    - ALU, Registros especiales, Multiplexores, Banco de registros …

- **Unidad de control**
  - Genera las señales necesarias para que la unidad de proceso realice en cada momento las operaciones correspondientes
  - Varias alternativas. En este curso estudiaremos su diseño como máquina de estados.

Arquitectura del procesador

- Definición de **Arquitectura del procesador**
  - Es el conjuntos de atributos de un computador que son visibles a:
    - El programador en lenguaje máquina
    - El sistema operativo
    - El compilador
  - Engloba los siguientes elementos:
    - Conjunto de **instrucciones** que puede ejecutar el procesador
    - **Tipos básicos de datos** soportados por las instrucciones
    - **Modos de direccionamiento.** Especifican dónde se encuentran los operandos y cómo acceder a ellos
    - Conjunto de **registros** visibles al programador
      - Registros de datos, direcciones, estado, contador de programa (PC)
    - **Memoria** accesible al programador
    - Mecanismos de **E/S**

Decisiones de diseño de la CPU

- **¿TODOS los datos** que manipula un programa se almacenan en la **memoria** o se va a disponer de **un banco de registros ?**
  - **La memoria**
    - Tiene más capacidad de almacenaje
    - Acceso a los datos lento
  - **El banco de registros**
    - Tiene menos capacidad de almacenaje
    - Acceso a los datos muy rápido

Decisiones de diseño de la CPU

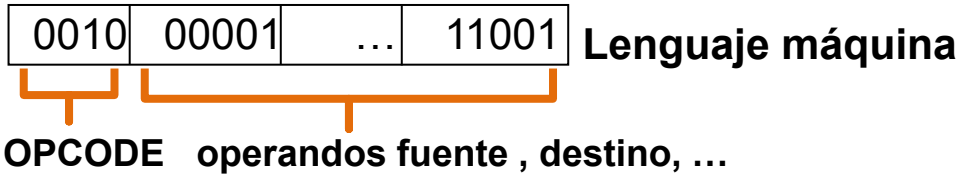
- Sobre el **Repertorio de instrucciones**
  - **¿ RISC o CISC ?**
    - RISC: *Reduced Instruction Set Computer*
    - CISC: *Complex Instruction Set Computer*
  - **Ancho de la instrucción:**
    - Número fijo o variable de bits que podemos utilizar para codificar la instrucción
  - **Tipos de instrucciones**
    - **Instrucciones aritmético-lógicas**
      - Realizar las operaciones matemáticas elementales
    - **Instrucciones de acceso a memoria**
      - Obtener los datos que utiliza la instrucción y almacenar los resultados de las operaciones
    - **Instrucciones de salto**
      - Modificar el flujo secuencial del programa
    - **Otras dependiendo de las características particulares del procesador**
  - **Número de instrucciones de cada tipo**

Decisiones de diseño de la CPU

- **Modos de direccionamiento**
  - Es la forma de especificar dentro de la instrucción:
    - **dónde están las datos**
      - Memoria
      - Registros
      - En la propia instrucción
    - **cómo acceder a ellos**
  - Cada familia de procesadores ofrece distintas posibilidades:
    - Inmediato
    - Absoluto
    - Directo de Registro
    - Indirecto de Registro
    - Indirecto de Registro con desplazamiento
    - …

Decisiones de diseño de la CPU

- **Formato de las instrucciones**
  - Las instrucciones tienen que contener la siguiente información:
    - **Tipo de instrucción** → OPCODE (Código de Operación)
    - **Dónde se encuentra el valor de cada operando** → Modos de direccionamiento (MDs)



    - **Número de operandos explícitos** de cada instrucción: 0, 1, 2, 3

Familia de procesadores

- Se denomina familia al conjunto de procesadores con la **misma arquitectura pero distinta implementación**
- Las familias de computadores hacen posible que:
  - existan máquinas de la misma familia con distinta:
    - Tecnología, velocidad, consumo, precio, etc.
  - las máquinas de una misma familia sean compatibles entre sí
    - La compatibilidad suele ser hacia arriba (upward compatibility)
    - Todos los miembros nuevos de una misma familia pueden ejecutar los programas creados para los miembros más antiguos

Los procesadores RISC-V

- Es el primer repertorio de instrucciones open-source con soporte empresarial.
- La arquitectura RISC-V se definió en 2010 en la University of California, Berkeley por Krste Asanović, Andrew Waterman, David Patterson, y otros
- RISC-V no es usual dado que su naturaleza open-source hace que se pueda usar libremente, y tiene capacidades comparables con arquitecturas comerciales clásicas como ARM y x896
- Algunas empresas como SiFive y Western Digital ya fabrican chips con esta arquitectura.

Lenguaje Ensamblador

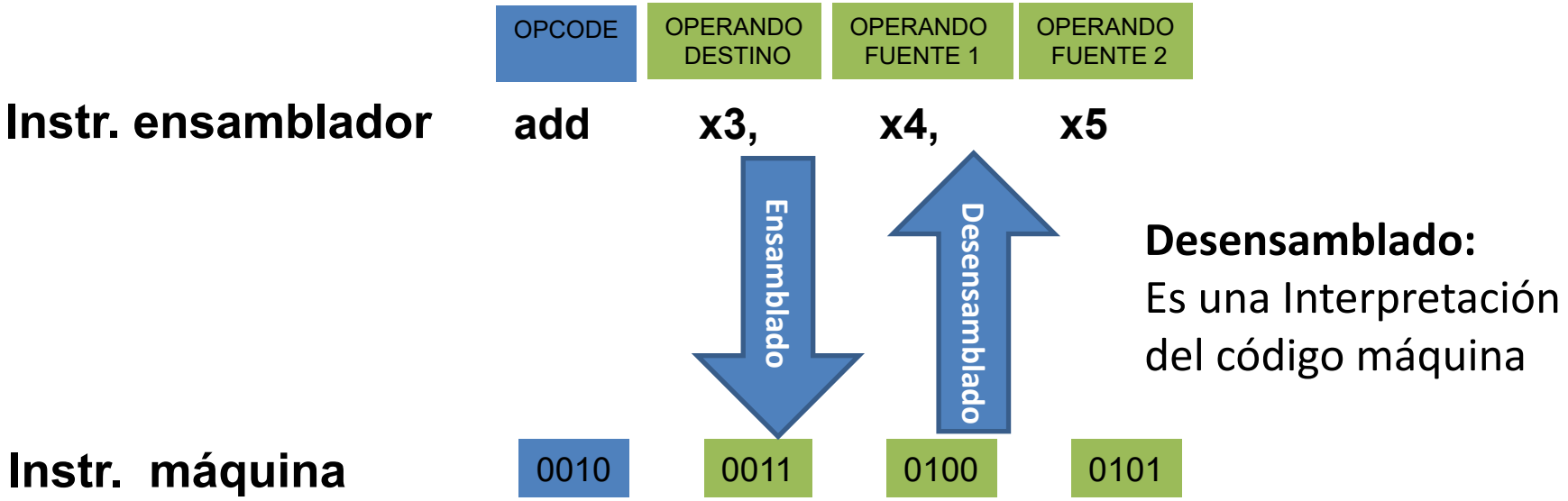
- LENGUAJE ENSAMBLADOR: Es la versión legible por un humano del lenguaje nativo del computador.

- Conjunto de instrucciones, símbolos y reglas sintácticas y semánticas con el que se puede programar un ordenador para que resuelva un problema, realice una tarea/algoritmo, etc.

- Podemos decir que el código ensamblador es un conjunto de expresiones fácilmente recordables por el programador, en las que además se tiene en cuenta la arquitectura del procesador:
  - No se puede utilizar cualquier expresión
  - Hay que considerar dónde se encuentran **físicamente** los datos

Lenguaje Ensamblador

- Un computador NO entiende código ensamblador.
  - Sólo entiende ceros y unos (código máquina)
- Cada instrucción escrita en código ensamblador y cada etiqueta, son traducidos mediante el ensamblado y el enlazado en código máquina:



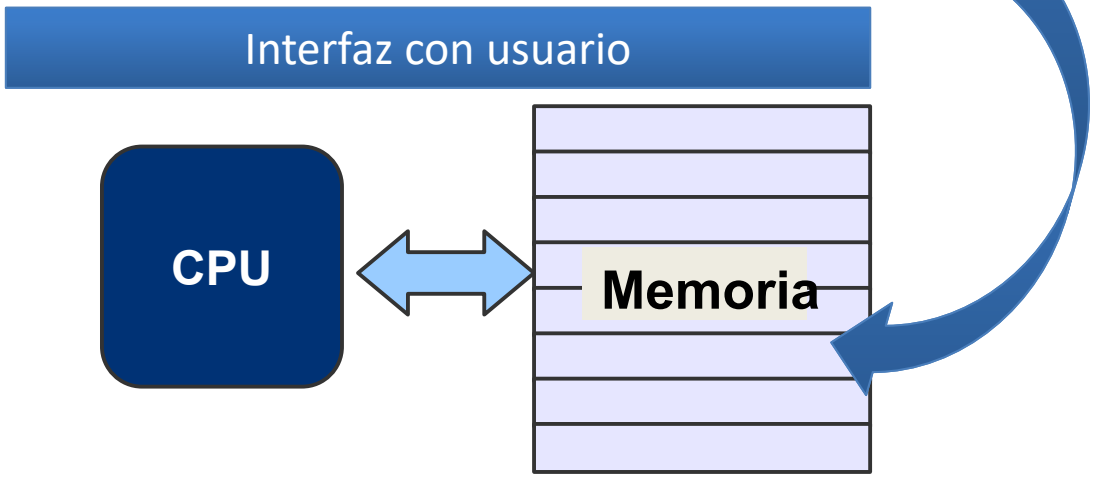
Desensamblado:
Es una Interpretación del código máquina

Lenguaje Ensamblador

```
program simple;
var
   a, b, p:
integer
begin
   a := 5;
   b := 3;
   p := a * b;
end
```

```
addi x1, x0, 5
addi x2 , x0, 3
sub x3, x1, x0
```

```
457f 464c 0101 0001 0000 0000 0000 0000
0002 0003 0001 0000 8280 0804 0034 0000
0df0 0000 0000 0000 0034 0020 0007 0028
0022 001f 0006 0000 0034 0000 8034 0804
8034 0804 00e0 0000 00e0 0000 0005 0000
0004 0000 0003 0000 0114 0000 8114 0804
8114 0804 0013 0000 0013 0000 0004 0000
0001 0000 0001 0000 0000 0000 8000 0804
8000 0804 046c 0000 046c 0000 0005 0000
1000 0000 0001 0000 046c 0000 946c 0804
946c 0804 0100 0000 0104 0000 0006 0000
1000 0000 0002 0000 0480 0000 9480 0804
9480 0804 00c8 0000 00c8 0000 0006 0000
0004 0000 0004 0000 0128 0000 8128 0804
8128 0804 0020 0000 0020 0000 0004 0000
```



Principios de diseño

- Hennessy y Patterson propusieron los siguientes principios de diseño de los procesadores RISC:

1. **La regularidad facilita la simplicidad de diseño**
2. **Hacer rápido el caso común**
3. **Más pequeño es más rápido**
4. **Buen diseño exige buenos compromisos**

RISC-V: Banco de registros

Más pequeño es más rápido:

- RISC-V tiene 32 registros de propósito general (x0-x31) agrupados en un Banco de Registros, cada uno de ellos de 32 bits (existen versiones de 64 y 128 bits)

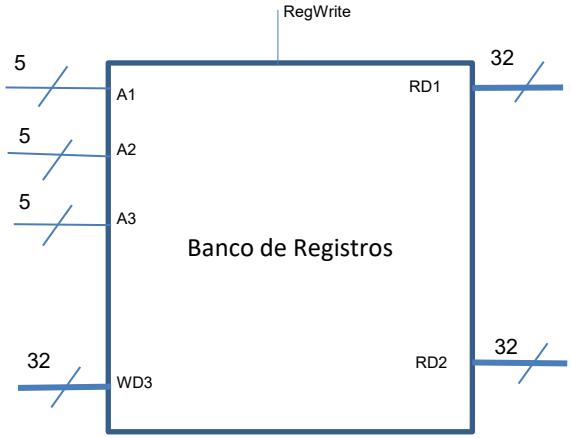
La regularidad facilita la simplicidad de diseño:

Todos los registros tienen 32-bits de anchura

- El Banco de Registros permite acceder

a la vez a 2 registros fuente y 1 registro

destino

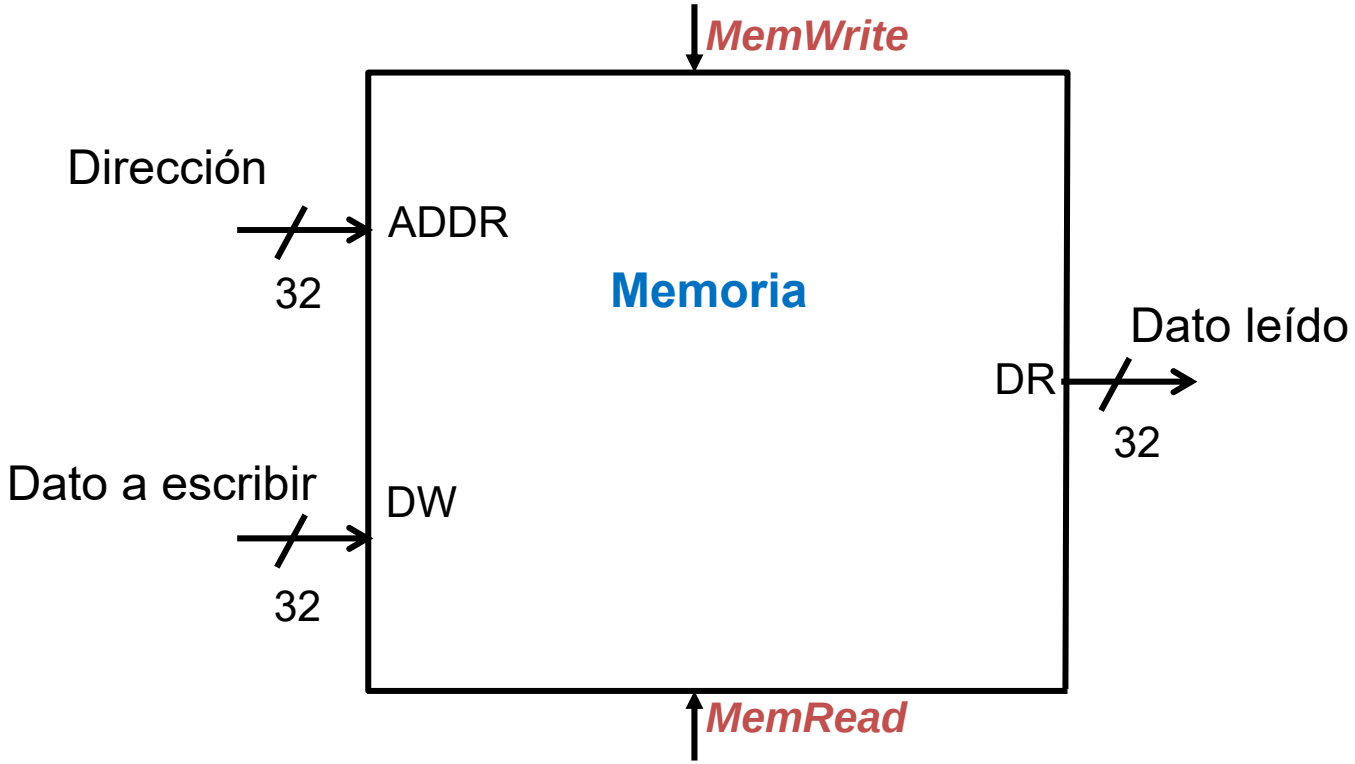


RISC-V: Banco de registros

Nombre	Número de Registro	Uso
zero	x0	Valor constante 0
ra	x1	Dirección de retorno
sp	x2	Puntero de pila
gp	x3	Puntero global
tp	x4	Puntero de hilo
t0-2	x5-x7	Registros temporales
s0/fp	x8	Registro salvado/puntero de frame
s1	x9	Registro salvado
a0-1	x10-11	Argumentos de función/Valores devueltos
a2-7	x12-17	Argumentos de función
s2-11	x18-27	Registros salvados
t3-6	x28-31	Registros temporales

RISC-V: Memoria

- Memoria accesible por bytes
- 32 bits de dirección => 2^{32} bytes direccionables
- Palabras de 4 bytes. Cada acceso a memoria lee o escribe 32 bits
- Accesos alineados (RISC-V suele usar accesos no alineados)
- Little endian



RISC-V: Instrucciones

- Un computador debe ser capaz de:
  - Realizar las operaciones de procesamiento de datos elementales: **Instrucciones aritméticas y lógicas.**
  - Leer/escribir datos desde/en la memoria: **Instrucciones de acceso a memoria.**
  - Modificar el flujo secuencial del programa: **Instrucciones de salto.**
  - Otras dependiendo de las características particulares de la arquitectura: DSP, FP, SIMD, etc

RISC-V: Instrucciones aritméticas-lógicas

- Operan con tres operandos: dos fuentes y un destino:

  add x1, x2, x3 (x1<= x2 + x3)

  sub x1, x2, x3 (x1<= x2 - x3)

Código C:

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

Código RISC-V compilado:

(x5, x6, x7, x8 y x9 contienen g, h, i, j y f respectivamente)

```
add x10, x5, x6   // temp x10 = g + h
add x11, x7, x8   // temp x11 = i + j
sub x9, x10, x11  // f = x10 – x11
```

- El segundo operando puede ser una constante (valor inmediato) de 12 bits en C2 a la que se extiende el signo hasta 32 bits. (Las constantes en hexadecimal van precedidas de 0x y en binario de 0b)

  addi x1, x2, 12 (x1 <= x2 + 12)

RISC-V: Instrucciones lógicas

- Realizan las operaciones lógicas: AND, OR o XOR bit a bit.
- Existe la versión inmediata (ANDI, ORI, XORI con operando como constante de 8 bits con signo extendido.

	Registros fuente			
x1	0100 0110	1010 0001	1111 0001	1011 0111
x2	1111 1111	1111 1111	0000 0000	0000 0000

Código ensamblador		Resultado				
and	**x3, x1, x2**	x3	0100 0110	1010 0001	0000 0000	0000 0000
or	**x4, x1, x2**	x4	1111 1111	1111 1111	1111 0001	1011 0111
xor	**x5, x1, x2**	x5	1011 1001	0101 1110	1111 0001	1011 0111

- Un NOT lógico puede hacerse con una XORI:

  xori x3, x4, 0xFF

RISC-V: Instrucciones de desplazamiento

- **sll**: shift left logical. Desplazamiento lógico a la izquierda. (Entran ceros)

  sll x5, x6, x7

- **srl**: shift right logical. Desplazamiento lógico a la derecha. (Entran ceros)

  srl x5, x6, x7

- **sra**: shift right arithmetic. Desplazamiento aritmético a la derecha. (Mantiene el signo)

  sra x5, x6, x7

- Existe la versión inmediata (**slli**, **srli**, **srai**)

RISC-V : Instrucciones de desplazamiento

- Cantidad de desplazamiento en el caso inmediato aparece como un campo de 5 bits sin signo.

	Registros fuente			
x8	1000 1000	0001 1100	0001 0110	1110 0111

Código ensamblador		Resultado				
slli	**x4, x8, 6**	x4	0000 0111	0000 0101	1011 1001	1100 0000
srli	**x5, x8, 4**	x5	0000 1000	1000 0001	1100 0001	0110 1110
srai	**x6, x8, 3**	x6	1111 0001	0000 0011	1000 0010	1101 1100

RISC-V: Instrucciones de Multiplicación

- **mul**: Multiplicación de 32 × 32, resultado 32 bits menos significativos

  `mul x1, x2, x3`

  Resultado: $x1 = (x2 * x3)_{31:0}$

- **mulh**: Multiplicación con signo de 32 × 32, resultado 32 bits más significativos

  `mulh x2, x3, x4`

  Resultado: $x2 = (x3 \times x4)_{63:32}$

- **mulhsu**: Multiplicación primer operando con signo y segundo sin signo de 32 × 32, resultado 32 bits más significativos

  `mulhsu x2, x3, x4`

  Resultado: $x2 = (x3 \times x4)_{63:32}$

- **mulhu**: Multiplicación sin signo. de 32 × 32, resultado 32 bits más significativos

  `mulhsu x2, x3, x4`

  Resultado: $x2 = (x3 \times x4)_{63:32}$

mulh x1, x3, x4
mul x2, x3, x4

?

RISC-V: Instrucción de acceso a memoria

- LOAD:

  lw x3, 8(x7) => x3 = MEM[x7+8]

  lee el contenido de la posición de memoria obtenida al sumar el contenido del registro x7 con el valor 8, y lo almacena en el registro x3

- STORE:

  sw x3, 8(x7) => MEM[x7+8] = x3

  Almacena el contenido del registro x3 en la posición de memoria obtenida al sumar el contenido del registro x7 con el valor 8.

RISC-V: Instrucción load

Estado inicial

Registros fuente				
x1	0000 0000	0000 0000	0000 0001	0000 0000
x2	0000 0000	0000 0000	1111 0001	0011 0010
x3	0000 0000	0000 0000	0000 0000	0000 0000

Dirección de memoria	Contenido de la memoria			
0x0000_0100	1010 1010	1011 1011	1100 1100	1101 1101
0x0000_0104	0001 0001	0010 0010	0011 0011	0100 0100
0x0000_0108	0000 0000	1111 1111	0101 0101	1110 1110

Código ensamblador	Resultado				
lw x3, 0(x1)	x3	1010 1010	1011 1011	1100 1100	1101 1101
lw x3, 8(x1)	x3	0000 0000	1111 1111	0101 0101	1110 1110

RISC-V: Instrucción store

Estado inicial

Registros fuente

x1	0000 0000	0000 0000	0000 0001	0000 0000
x2	0000 0000	0000 0000	1111 0001	0011 0010
x3	0000 0000	0000 0000	0000 0000	0000 0000

Dirección de memoria	Contenido de la memoria			
0x0000_0100	1010 1010	1011 1011	1100 1100	1101 1101
0x0000_0104	0001 0001	0010 0010	0011 0011	0100 0100
0x0000_0108	0000 0000	1111 1111	0101 0101	1110 1110

Código ensamblador	Dirección de memoria	Resultado
sw x3, 0(x1)	0x0000_0100	0000 0000 0000 0000 0000 0000 0000 0000
sw x3, 8(x1)	0x0000_0108	0000 0000 0000 0000 0000 0000 0000 0000

RISC-V: Instrucciones load y store

- RISC-V dispone de instrucciones de load y store en tamaño byte y halfword
  - Load byte/halfword/word: Extendido el signo a 32 bits en rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word sin signo: Extensión con ceros a 32 bits en rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Almacena los 8/16/32 bits menos significativos
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

RISC-V: Instrucciones load y store

Estado inicial

Registros fuente				
x1	1010 1010	1011 1011	1100 1100	1101 1101
x2	1010 1010	1011 1011	1100 1100	1101 1101
x3	1011 1011	1100 1100	1101 1101	1010 1010
x4	0000 0000	0000 0000	0000 0000	0000 0000

LittleEndian

Dirección de memoria	Contenido de la memoria (+3)	(+2)	(+1)	
0x0000_0000	1000 0011	0100 0010	1000 1100	1111 0111

Código ensamblador		Resultado			
lh x1, 2(x4)	x1	1111 1111	1111 1111	1000 0011	0100 0010
lbu x2, 0(x4)	x2	0000 0000	0000 0000	0000 0000	1111 0111

Código ensamblador	Dirección de memoria	Resultado
sb x3, 0(x4)	0x0000_0000	0000 0000 0000 0000 0000 0000 1010 1010

Generación de constantes

- Para inicializar registros con valores constantes usamos el inmediato de 12 bits de la instrucción **addi**

Código C

```
// int es una palabra
   de 32 bits con
   signo
int a = -372;
int b = a + 6;
```

Código ensamblador RISC-V

```
# s0 = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

Generación de constantes

- Para generar constantes más grandes usaremos load upper immediate (**`lui`**) y **`addi`**
- **`lui`**: coloca un inmediato en los 20 bits más significativos de un registro destino y 0's en los 12 bits menos significativos

Código C

```
int a = 0xFEDC8765;
```

Código ensamblador RISC-V

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

Recordar que `addi` **extiende el signo** de su inmediato de 12 bits.

Generación de constantes

- Si el **bit 11** de la constante de 32 bits es **1**, se debe incrementar los 20 bits más significativos en 1 en **lui**

Código C

```
int a = 0xFEDC8EAB;
```

Código ensamblador RISC-V

```
# s0 = a
lui  s0, 0xFEDC9        # s0 = 0xFEDC9000
addi s0, s0, -341       # s0 = 0xFEDC9000 + 0xFFFFFEAB
                        #    = 0xFEDC8EAB
```

RISC-V: Instrucciones de salto

- Los saltos permiten la ejecución de instrucciones fuera de la secuencia normal.
- RISC-V tiene seis tipos de saltos condicionales, todos ellos usan dos registros fuente y una etiqueta que indica a qué instrucción saltar.
- La etiqueta va seguida de :

beq	beq x2, x3, L1	Salta a L1 si x2 es igual a x3
bne	bne x2, x3, L1	Salta a L1 si x2 no es igual a x3
blt	blt x2, x3, L1	Salta a L1 si x2 es menor que x3 (x2 y x3 con signo)
bge	bge x2, x3, L1	Salta a L1 si x2 es mayor o igual que x3 (x2 y x3 con signo)
bltu	bltu x2, x3, L1	Salta a L1 si x2 es menor que x3 (x2 y x3 sin signo)
bgeu	bgeu x2, x3, L1	Salta a L1 si x2 es mayor o igual que x3 (x2 y x3 sin signo)

RISC-V: Instrucciones de salto

Ejemplo de salto

```
        addi s0, zero, 4        # s0 = 0 + 4 = 4
        addi s1, zero, 1        # s1 = 0 + 1 = 1
        slli s1, s1, 2          # s1 = 1 << 2 = 4
        beq s0, s1, target      # s0 == s1, se realiza el salto
        addi s1, s1, 1          # no ejecutada
        sub s1, s1, s0          # no ejecutada
target:                         # etiqueta
        jadd s1, s1, s0         # s1 = 4 + 4 = 8
```

RISC-V: Instrucciones de salto

- Tres instrucciones de salto incondicional:

j etiqueta # PC = etiqueta

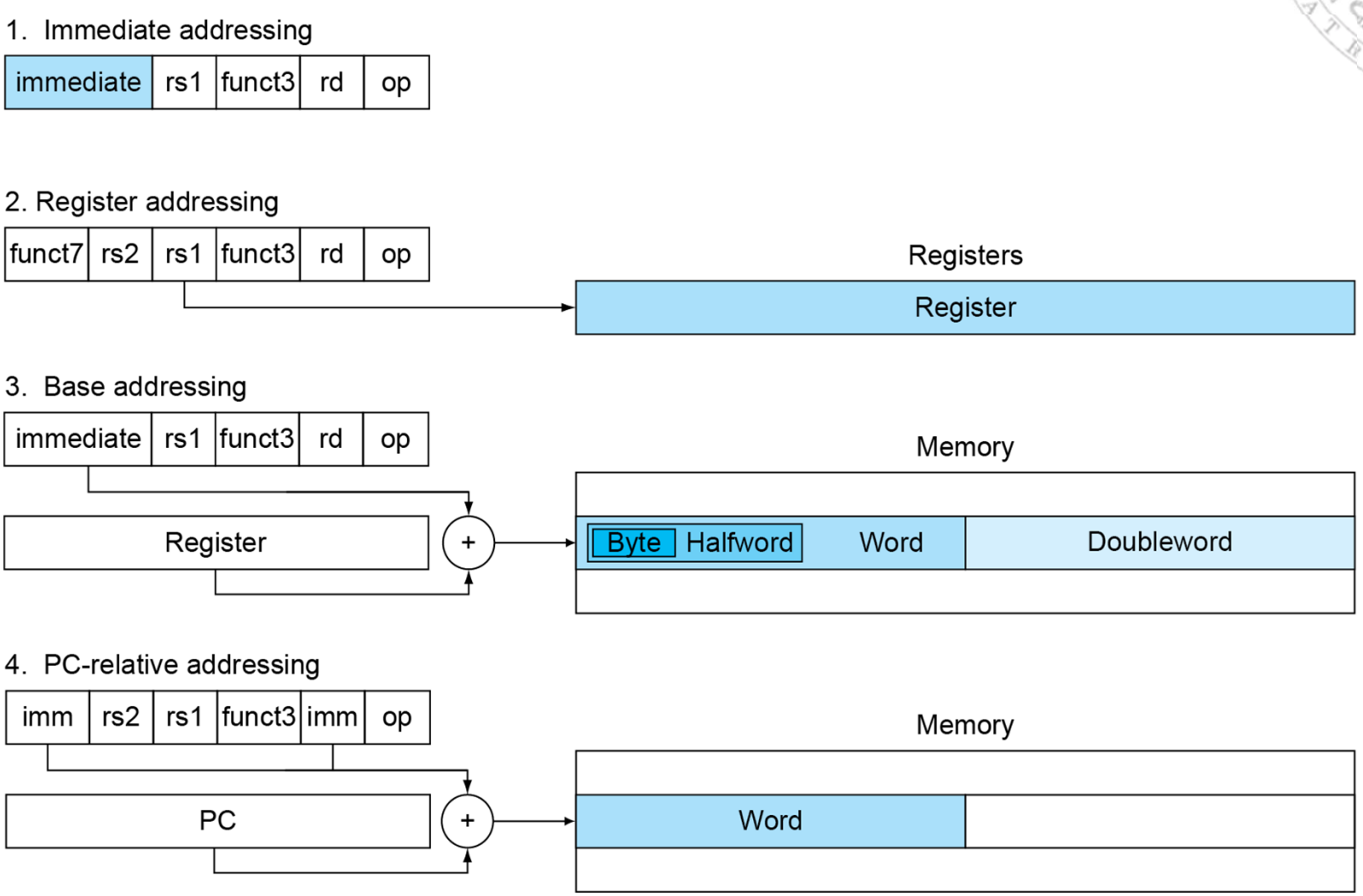
jal etiqueta # ra = PC + 4
PC = etiqueta
salto a función

jr ra # PC = ra
retorno de función

Ejemplo de salto

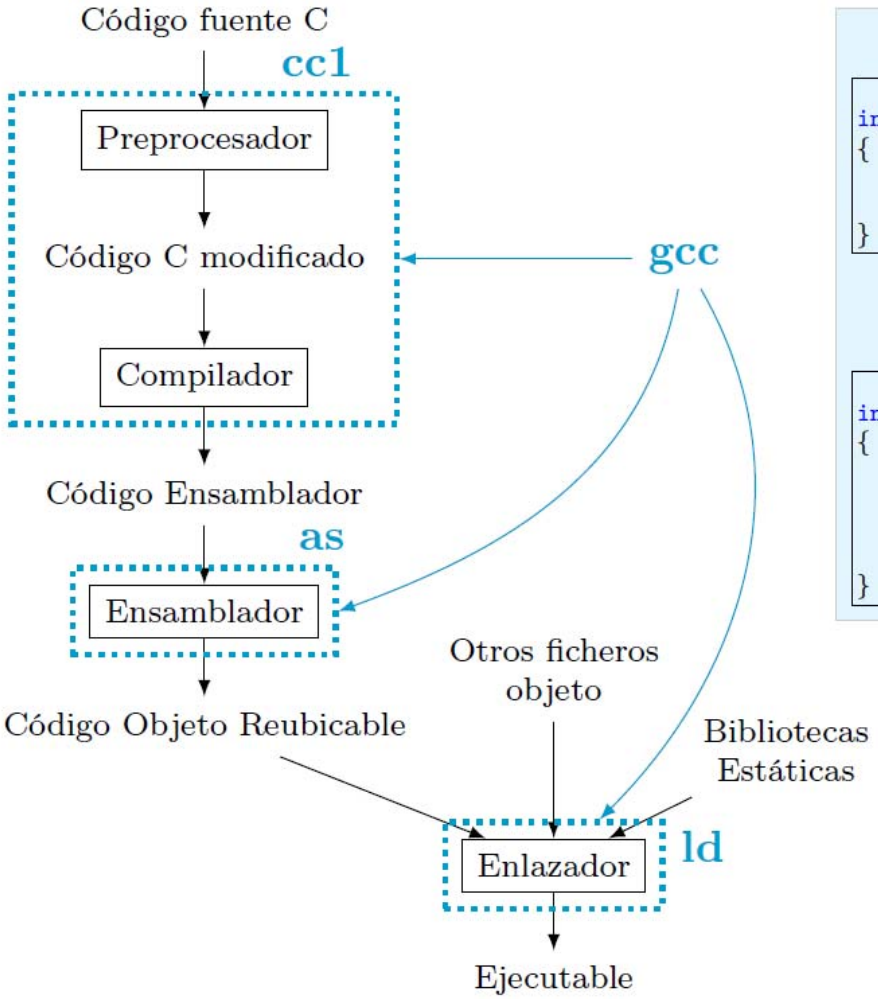
```
        j target          # salta a target
        srai s1, s1, 2    # no ejecutada
        addi s1, s1, 1    # no ejecutada
        sub s1, s1, s0    # no ejecutada
target:
        add s1, s1, s0    # s1 = s1 + s0
```

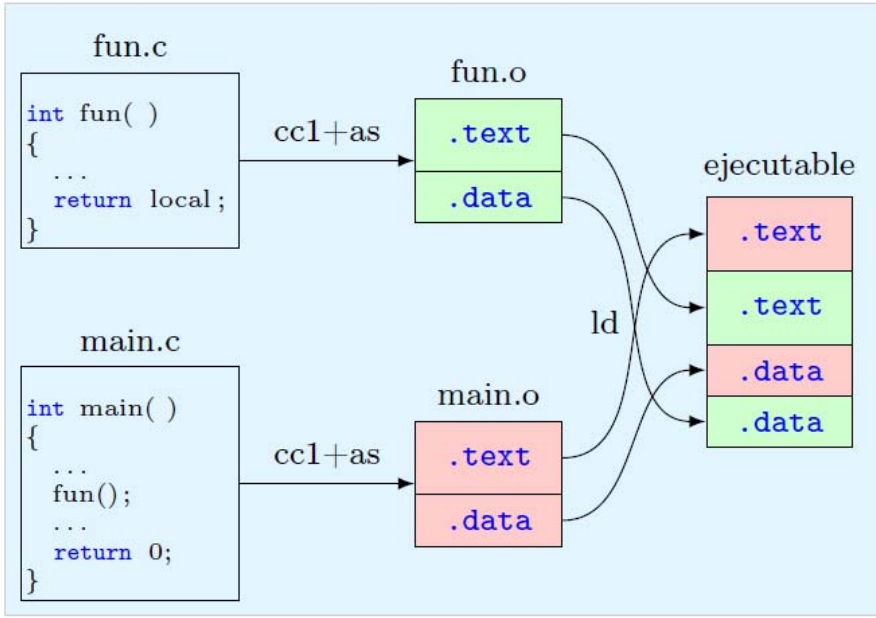
RISC-V: Modos de direccionamiento



Ensamblador

- Etapas de compilación, ensamblado y enlazado





Ensamblador

- Cada línea del programa puede tener los siguientes campos:

Etiqueta	Instrucción/Directiva	Operandos	Comentarios

- **Etiqueta**: Referencias simbólicas de posiciones de memoria
- **Directiva**: acciones auxiliares durante el ensamblado (e.g. reserva de memoria)
- **Instrucción**: del repertorio del RISC-V
- **Operandos**:
  - Registros
  - Constantes
  - Etiquetas
- **Comentarios:** caracteres que siguen a #. Pueden aparecer solos en una línea.

```
        .globl start
        .equ    ONE, 0x01   #Constant
        .data               #Data
DOS:    .word 0x02          #Variable
        .bss
RES:    .space 4
        .text               #Program
start:  addi x5, x0, #ONE
        ld x1, =DOS
        ld x2, 0(x1)
        add  x3, x5, x2
        ld x4, =RES
        st x3, 0(x4)
END:    .end
```

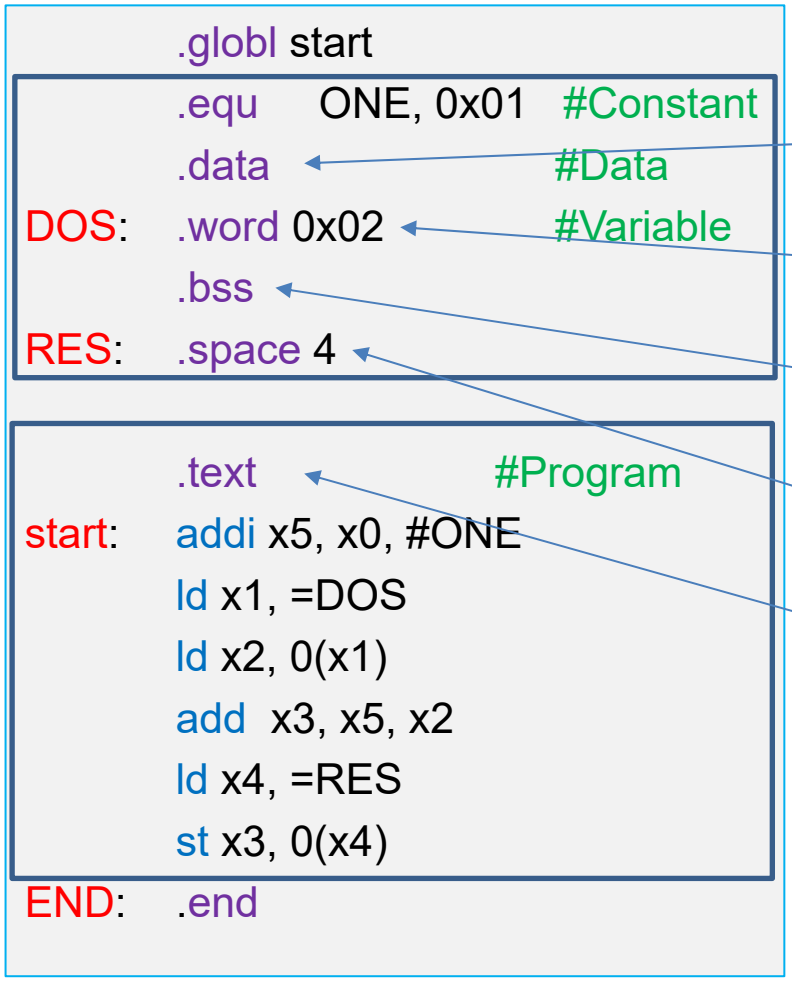
Directivas de ensamblado

Directiva	Propósito
`.text`	Declara el comienzo de la sección de instrucciones
`.data`	Declara el comienzo de la sección de variables globales **con** valor inicial
`.bss`	Declara el comienzo de la sección de variables globales con valor inicial 0
`.word w1,...,wn`	Reserva *n* **palabras** en memoria e inicializa el contenido a *w1,...,wn*
`.byte b1,…, bn`	Reserva *n* **bytes** en memoria e inicializa el contenido a *b1,...,bn*
`.space n`	Reserva *n* **bytes** de memoria
`.string “str”`	Almacena la cadena “str” en memoria
`.equ nom, valor`	Define una constante llamada *nom* como *valor*
`.globl sym`	La etiqueta *sym* es global
`.end`	Fin del código ensamblador

Pseudo-instrucciones de load y store

Pseudoinstrucción	Instrucciones RISC-V	Descripción	Operación
j label	jal zero, label	salto	PC = label
jr ra	jalr zero, 0(ra)	Salto con registro	PC = ra
mv t5, s3	addi t5, s3, 0	move	s5 = t3
not s7, t2	xori s7, t2, -1	Negación Complemento a 1	s7 = ~t2
nop	addi zero, zero, 0	No operación	
li ss8, 0x7ef	addi s8, zero, 0x7EF	Carga inmediato de 12 bits	s8 = 0x7EF
li s8, 0x56789DEF	lui s8, 0x5678A addi s8, s8, 0xDEF	Carga inmediato de 32 bits	s8 = 0x56789DEF
la rd, symbol	auipc rd, $symbol_{31:12}$ addi rd, rd, $symbol_{11:0}$	Carga dirección de variable global	rd = dirección de symbol
bgt s1, t3, L3	blt s1, t3, L3	Saltar si >	Si (s1>t3), PC=L3
bgez t2, L7	bge t2, zero, L7	Saltar si >=0	Si (t2>=0), PC=L7
call L1	jal L1	Llama a función próxima	PC =L1 ra = PC+4
call L5	auipc ra, $inm_{31:12}$ jalr ra, ra, $inm_{11:0}$	Llama a función lejana	PC = L5 ra = PC+4
ret	jalr zero, 0(ra)	Retorna de función	PC = ra

Zonas de un programa ensamblador



A partir de .data se enumeran las variables de entrada

Se define la variable DOS y se inicializa a dos en hexadecimal

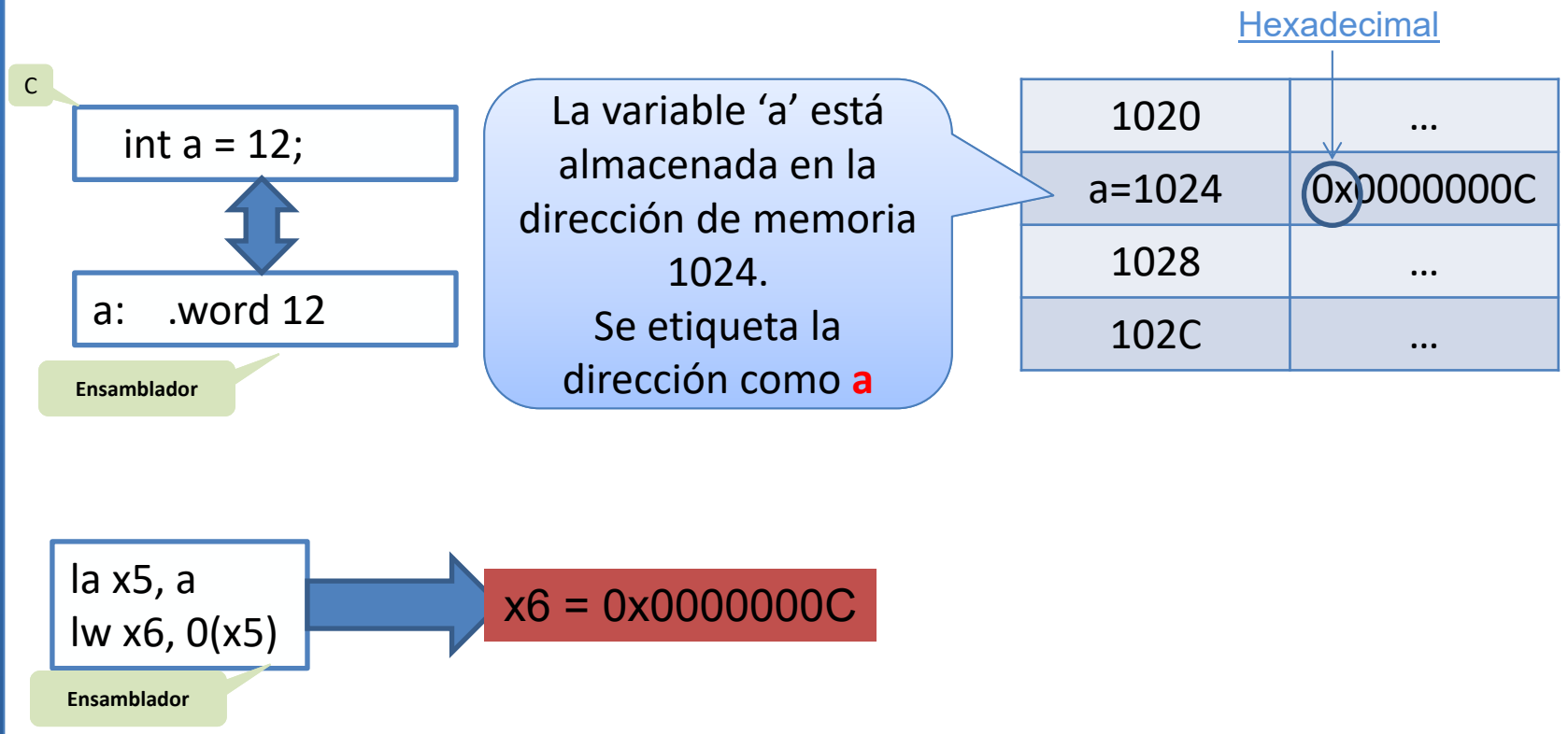
A partir de .bss se enumeran las variables de salida

Se define la variable RES y se reservan 4 bytes en memoria para almacenarla

Comienzo del programa

¿Cómo acceder a una variable?

- **Direccionamiento absoluto**:
  - El operando está en la dirección de memoria indicada.
- **Pseudo-instrucción**: No la proporciona el repertorio de instrucciones del RISC-V



¿Cómo acceder a un array?

C

```
int a[10];


a[0] = 1;
a[1] = 3;
a[2] = 7;
a[3] = 5;
a[4] = -2;
…
a[9] = 4;
```

```
a:   .word   1, 3, 7, 5, -2, …, 4
```

Ensamblador

a = 0x00001020	0x00000001
0x00001024	0x00000003
0x00001028	0x00000007
0x0000102C	0x00000005
0x00001030	0xFFFFFFFE
…	…
0x00001044	0x00000004

Ensamblador

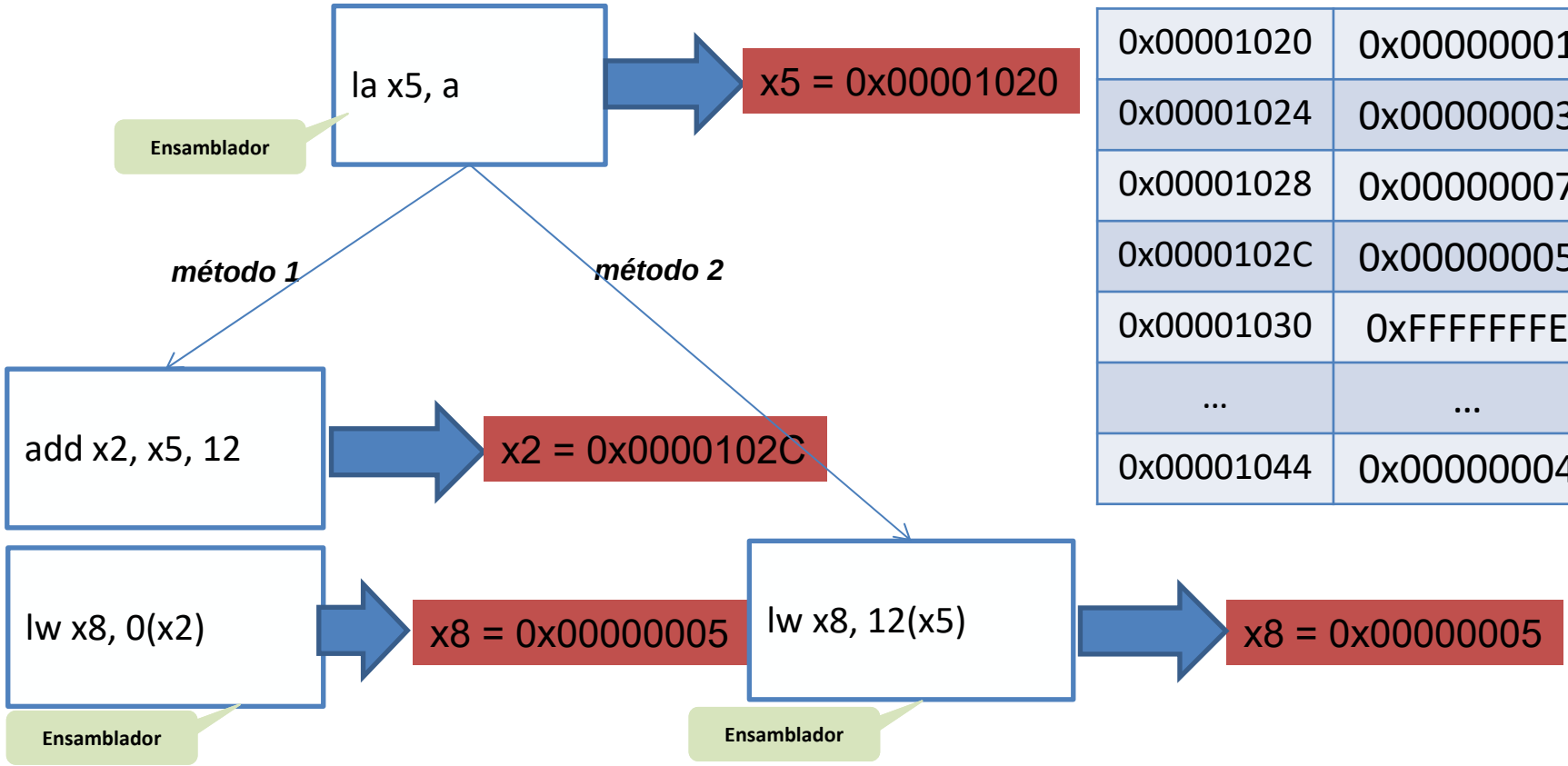
```
la x5, a
lw x6, 0(x5)
```

x6 = 0x00000001

¿Y si quiero acceder a otra componente?

¿Cómo acceder a un array?

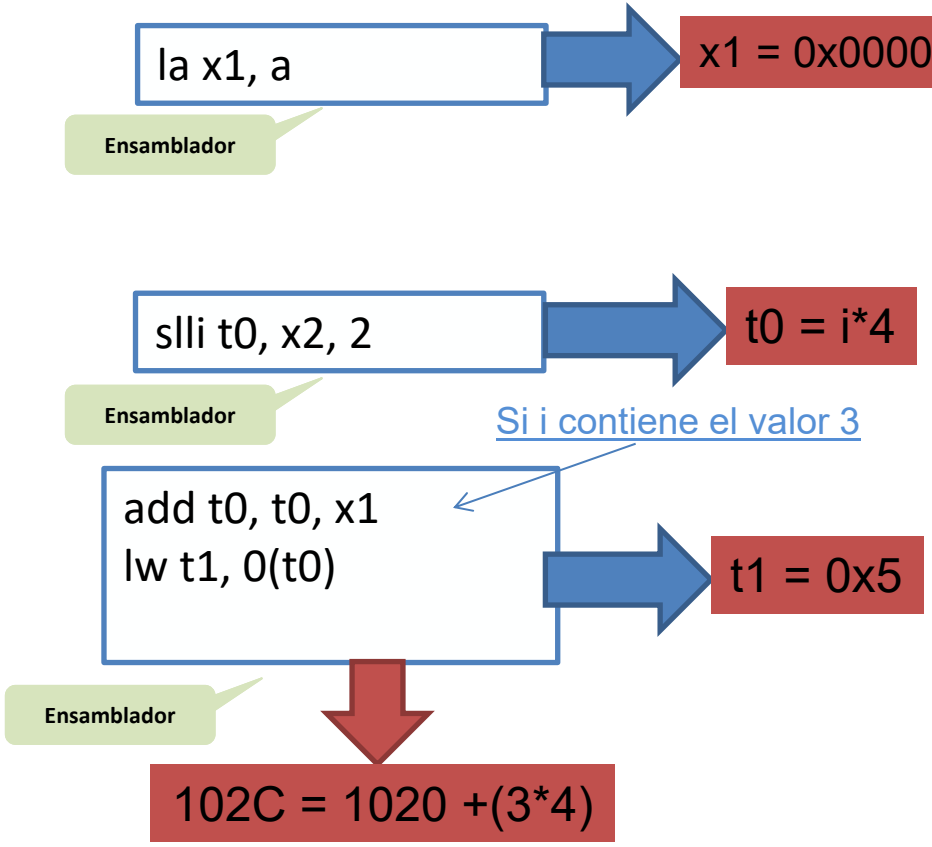
¿Dónde está la componente 3? **0x0000102C**



0x00001020	0x00000001
0x00001024	0x00000003
0x00001028	0x00000007
0x0000102C	0x00000005
0x00001030	0xFFFFFFFE
...	...
0x00001044	0x00000004

¿Cómo acceder a un array?

¿Dónde está la componente i del array? **1020 + 4*i**
x2 contiene i



0x00001020	0x00000001
0x00001024	0x00000003
0x00001028	0x00000007
0x0000102C	0x00000005
0x00001030	0xFFFFFFFE
...	...
0x00001044	0x00000004

Sentencias if

Código C

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

Código ensamblador RISC-V

```
@ x5=f, x1=g, x2=h, x3=i, x4=j

 bne x3, x4, L1    # si i!=j, salta el bloque
                   # if. Se comprueba
                   # la condición contraria
                   # del if
 add x5, x1, x2    # f = g + h

L1
 sub x5, x5, x3    # f = f - i
```

Sentencias if/else

Código C

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

Código ensamblador RISC-V

```
#x5=f, x1=g, x2=h, x3=i, x4=j

 bne x3, x4, L1    # si i!=j, salta el bloque if
 add x5, x1, x2    # f = g + h
 j   L2            # salta el bloque else
L1
 sub x5, x5, x3    # f = f - i
L2
```

Bucles while

Código C

```
// calcula la potencia
// de x,  2^x = 128

int pot = 1;
int x   = 0;
while (pot != 128)
   {
   pot = pot * 2;
   x = x + 1;
   }
```

Código ensamblador RISC-V

```
# R2 = pot, R1 = x
   addi x2, x0, 1   # pot = 1
   add x1, x0, x0   # x = 0

WHILE
   beq x2, 128, DONE      # si(pot==128
                          # salir del
   bucle
   sll x2, x2, 1    # pot=pot*2
   add x1, x1, 1    # x=x+1
   j   WHILE        # repetir bucle
DONE
```

Bucles for

```
for (inicialización; condición; iteración)
        {
        sentencias
        }
```

- **inicialización :** se ejecuta antes de que comience el bucle
- **condición:** se comprueba al principio de cada iteración
- **iteración:** se ejecuta al final de cada iteración
- **sentencias:** se ejecutan cada vez que la condición se cumple

Bucles for

Código C

```
@ suma los números del
   1 al 9
int sum = 0

for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

Código ensamblador RISC-V

```
# x2 = i, x1 = sum
   addi x2, x0, 1   # i = 1
   addi x1, x0, 0   # sum = 0

FOR
   beq x2, 10,  DONE      # si (i==10)
                          # salir del bucle
   add x1, x1, x2   # sum=sum + i
   add x2, x2, 1    # i = i + 1
   j   FOR          # repetir bucle

DONE
```

Llamadas a función (método)

- Los lenguajes de alto nivel disponen de funciones para reusar código que funcionan del siguiente modo:

Código C

```
void main()
{
  int y;
  y = sum(42,7);
  …
}

int sum(int a, int b)
{
  return (a + b);
}
```

- **Función que llama:**
  - Pasa **argumentos** a la función llamada.
  - Salta a la función llamada.

- **Función llamada:**
  - **Ejecuta** la función.
  - **Devuelve** el resultado a quien hizo la llamada.
  - **Retorna** al punto de llamada.
  - **No debe sobreescribir** registros o memoria que se puedan necesitar por parte de quien hizo la llamada.

Llamadas a función

- **¿Cómo se hacen las llamadas a función en ensamblador?**
  - **Función que llama:** Se usa la instrucción jump and link

    `jal` TARGET

    Esta instrucción guarda el PC en el Link register, ra (x1) y actualiza el PC con la dirección TARGET para saltar a la dirección de comienzo de la función:
    - `ra` = PC + 4
    - salta a TARGET
  - **Retorno** desde la función llamada: la última instrucción de la función llamada mueve el **link register** al **PC**:

    `jr ra`
    - PC = `ra`
  - **Argumentos:** Se suelen poner en: `a0` – `a7`
  - **Valor retornado:** Se suele poner en: `a0`

Llamadas a función

Código C

```
int main() {
   int resultado;
   result =suma(7, 16);
}

int suma(int a, int b)
   {int r;
   r = a + b;
   return r;
}
```

Código ensamblador RISC-V

```
0x00000200 MAIN      addi a1, x0, 7
0x00000204           addi a2, x0, 16
0x00000208           jal  SUMA
...

0x00401020 SUMA      add a0, a1, a2
                     jr ra
```

jal SUMA salta a `SUMA` (dirección 0x00401020)
`ra` = `PC` + 4 = 0x0000020C

jr ra `PC` = `ra`
(la siguiente instrucción ejecutada está en 0x0000020C)

Paso de argumentos y devolución de resultados

Código C

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);
// 4 argumentos
  ...
}

int diffofsums(int f, int g, int h,
    int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;
}
```

Ensamblador RISC-V

```
# s7 = y
main:
. . .
addi a0, zero, 2  # argumento 0 = 2
addi a1, zero, 3  # argumento 1 = 3
addi a2, zero, 4  # argumento 2 = 4
addi a3, zero, 5  # argumento 3 = 5
jal  diffofsums   # call function
add  s7, a0, zero # y =valor devuelto
. . .
# s3 = result
diffofsums:
add  t0, a0, a1   # t0 = f + g
add  t1, a2, a3   # t1 = h + i
sub  s3, t0, t1   # result = (f + g)
- (h + i)
add  a0, s3, zero # retorna valor en a0
jr   ra           # Vuelve a main
```

Paso de argumentos y devolución de resultados

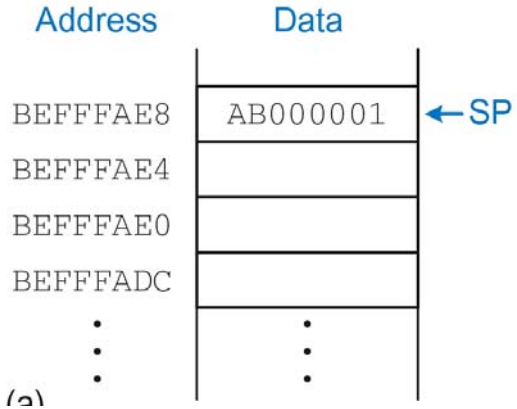
Código ensamblador RISC-V

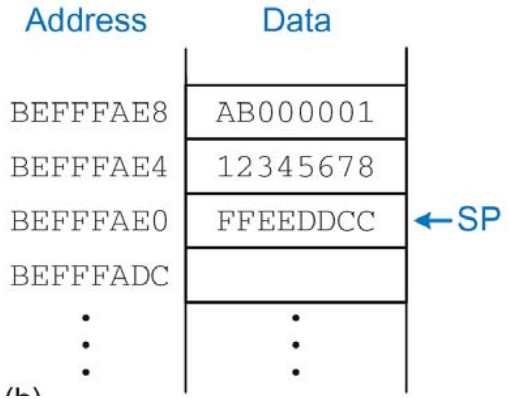
```
; R4 = result
DIFFOFSUMS
  add  t0, a0, a1    # t0 = f + g
  add  t1, a2, a3    # t1 = h + i
  sub  s3, t0, t1    # result = (f + g) - (h + i)
  add  a0, s3, zero  # put return value in a0
  jr   ra
```

- *diffofsums* sobrescribe 3 registros: t0, t1, s3.
- Si estos registros son utilizados por el programa principal (el que llama a la función) deberían ser protegidos.
- Para ello, *diffofsums* puede usar la **pila** para almacenar temporalmente el valor de los registros

La pila

- La pila es Memoria usada para guardar temporalmente variables dentro de una función.
- Cola *last-in-first-out* (LIFO) .
- ***Expandir***: usa más memoria cuando se necesita más espacio.
- ***Contraer***: usa menos memoria cuando ya no se necesita más el espacio.
- Las pilas crecen de las direcciones de memoria más altas hacia las más bajas.
- Stack pointer (puntero de pila): sp (x2) apunta al tope de la pila, la última posición ocupada.





¿Cómo usan las funciones la pila?

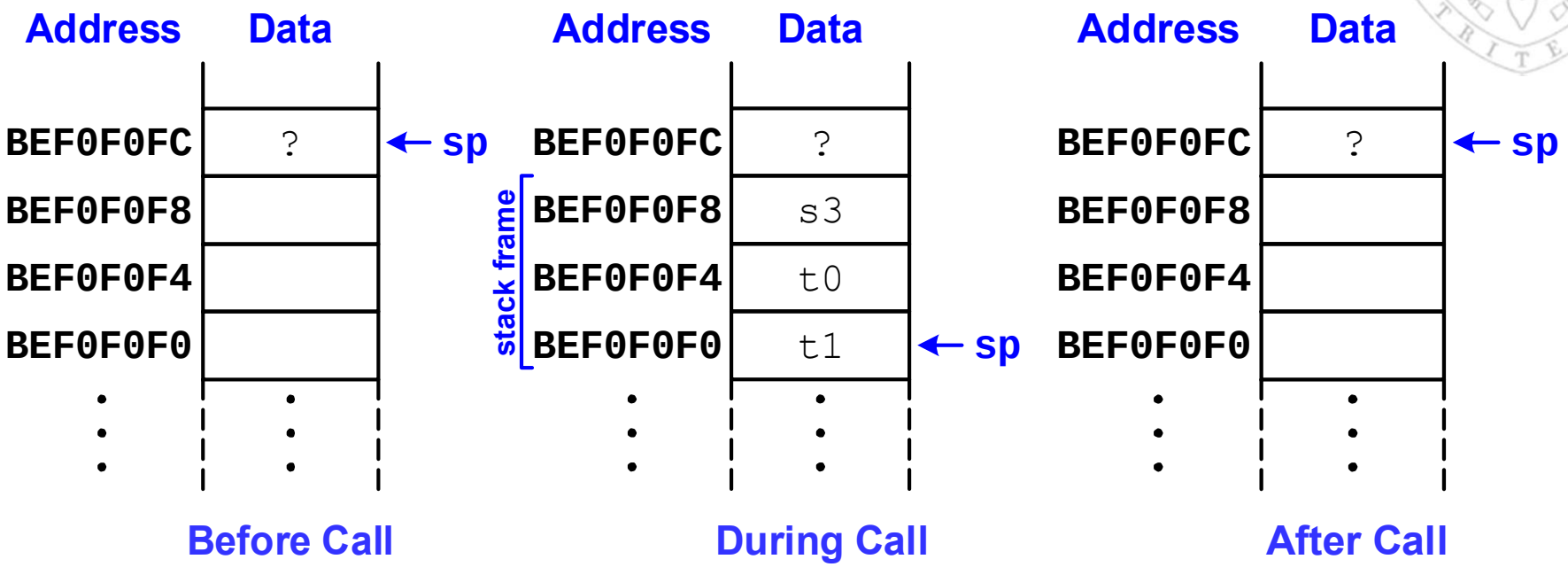
- Las funciones llamadas no deben tener efectos colaterales no deseados.
- Por ello todos los registros modificados en la función deberían protegerse.
- El proceso es:
  1. Reservar espacio en la pila para almacenar los valores de los registros
  2. Almacenar los valores de los registros en la pila
  3. Ejecutar el código de la función usando los registros
  4. Recuperar los valores originales de los registros de la pila
  5. Liberar el espacio usado en la pila para almacenar los valores de los registros

¿Cómo usan las funciones la pila?

- `diffofsums` sobrescribía 3 registros: `R4`, `R8`, `R9`

```
# s3 = result
diffofsums:
  addi sp, sp, -12        # reserva espacio en la pila para
                           #3 registros
  sw   s3, 8(sp)          # salva s3 en la pila
  sw   t0, 4(sp)          # salva t0 en la pila
  sw   t1, 0(sp)          # salva t1 en la pila
  add  t0, a0, a1         # t0 = f + g
  add  t1, a2, a3         # t1 = h + i
  sub  s3, t0, t1         # result = (f + g) - (h + i)
  add  a0, s3, zero       # coloca el valor de retorno en a0
  lw   s3, 8(sp)          # restaura s3 de la pila
  lw   t0, 4(sp)          # restaura t0 de la pila
  lw   t1, 0(sp)          # restaura t1 de la pila
  addi sp, sp, 12         # libera el espacio de pila
  jr   ra                 # vuelve al programa que llama
```

La pila durante la llamada a función



RISC-V: Punto flotante

- RISC-V ofrece tres extensiones para gestionar números en PF:
  - **RVF:** precisión simple (32-bit)
    - 8 bits de exponente, 23 bits de mantisa
  - **RVD:** doble precisión (64-bit)
    - 11 bits de exponente, 52 bits de mantisa
  - **RVQ:** cuadruple precisión (128-bit)
    - 15 bits de exponente, 112 bits de mantisa
- Registros en PF:
  - 32 registros en Punto flotante
  - Anchura de 32, 64 o 128 bits
  - Cuando se implementan cvarias extensions de punto flotante, los valores de menos precision ocupan los bits menos significativos del registro

Registros en Punto flotante

Name	Register Number	Usage
`ft0-7`	f0-7	Variables temporales
`fs0-1`	f8-9	Variables salvadas
`fa0-1`	f10-11	Argumentos de función/valor de retorno
`fa2-7`	f12-17	Argumentos de función
`fs2-11`	f18-27	Variables salvadas
`ft8-11`	f28-31	Variables temporales

Instrucciones en PF

- Añadir .s (single), .d (double), .q (quad) para precision `fadd.s`, `fadd.d`, and `fadd.q`
- **Operaciones aritméticas:**

  `fadd`, `fsub`, `fdiv`, `fsqrt`, `fmin`, `fmax`, multiply-add (`fmadd`, `fmsub`, `fnmadd`, `fnmsub`)
- **Otras instrucciones:**

  mover (`fmv.x.w`, `fmv.w.x`)
  convertir (`fcvt.w.s`, `fcvt.s.w`, etc.)
  comparación (`feq`, `flt`, `fle`)
  clasificar (`fclass`)
  sign injection (`fsgnj`, `fsgnjn`, `fsgnjx`)
- **fmadd** es la instrucción más crítica para programas de procesado de señales

  Requiere cuatro registros

```
fmadd.f f1, f2, f3, f4      # f1 = f2 x f3 + f4
```