DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Sistemas Digitales

Arquitectura

Primer Cuatrimestre 2024

Sistemas Digitales
DC - UBA

Introducción

Clase de hoy

Hoy vamos a ver:

- Definición de **arquitecturas**.

Clase de hoy

Hoy vamos a ver:

- Definición de **arquitecturas**.
- El lenguaje ensamblador de `RISC V`.

Clase de hoy

Hoy vamos a ver:

- Definición de **arquitecturas**.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.

Clase de hoy

Hoy vamos a ver:

- Definición de **arquitecturas**.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.
- Compilación, ensamblado, vinculación y ejecución.

Clase de hoy

Hoy vamos a ver:

- Definición de **arquitecturas**.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.
- Compilación, ensamblado, vinculación y ejecución.
- Programa de ejemplo.

Introducción

¿Qué es la arquitectura, o mejor dicho, la **arquitectura de un procesador**? La **arquitectura** de un procesador se refiere a aquello con lo que podemos trabajar cuando escribimos un programa. Son las instrucciones, los registros y la forma de acceder a memoria.

Introducción

¿Cómo interactuamos con la **arquitectura de un procesador**? Escribiendo un programa en un **lenguaje ensamblador**, o sea, el lenguaje que el procesador entiende.

¿Qué cosa no es la **arquitectura de un procesador**? La implementación específica del procesador que le permite ejecutar estos programas. Puede haber varias implementaciones distinas de una misma arquitectura pertenecientes a una o varias empresas, para el programa, siempre y cuando respeten lo que la arquitectura define, van a ser intercambiables.

¿Qué constituye una arquitectura?

Introducción

¿Qué constituye una arquitectura?

- El conjunto de **instrucciones**.

Introducción

¿Qué constituye una arquitectura?

- El conjunto de **instrucciones**.
- El conjunto de **registros**.

¿Qué constituye una arquitectura?

- El conjunto de **instrucciones**.
- El conjunto de **registros**.
- La forma de acceder a la **memoria**.

Introducción

¿Qué constituye una arquitectura?

- El conjunto de **instrucciones**.
- El conjunto de **registros**.
- La forma de acceder a la **memoria**.

¿Qué es una instrucción, un registro o una memoria?

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Veamos:

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Veamos:

- ¿Qué comportamiento tiene este programa?

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Veamos:

- ¿Qué comportamiento tiene este programa?
- ¿Cómo interpreta el procesador la línea 5? ¿Esto se realiza en una o varias instrucciones de lenguaje máquina?

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Veamos:

- ¿Qué comportamiento tiene este programa?
- ¿Cómo interpreta el procesador la línea 5? ¿Esto se realiza en una o varias instrucciones de lenguaje máquina?
- ¿Cómo se representan y almacenan las variables `i` y `acumulador`?

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Veamos:

- ¿Qué comportamiento tiene este programa?
- ¿Cómo interpreta el procesador la línea 5? ¿Esto se realiza en una o varias instrucciones de lenguaje máquina?
- ¿Cómo se representan y almacenan las variables `i` y `acumulador`?
- ¿Cómo se representan y almacenan las variables `a` y `largo`?

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Veamos:

- ¿Qué comportamiento tiene este programa?
- ¿Cómo interpreta el procesador la línea 5? ¿Esto se realiza en una o varias instrucciones de lenguaje máquina?
- ¿Cómo se representan y almacenan las variables `i` y `acumulador`?
- ¿Cómo se representan y almacenan las variables `a` y `largo`?
- ¿Cómo se decide cuál es la próxima instrucción a ejecutar?

Programa de ejemplo en C

Al final de la clase vamos a poder responder todas estas preguntas, en el contexto de una arquitectura en particular. A continuación, un pequeño adelanto.

Prgrama de ejemplo en ASM (RISC V)

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

```
1  .section .text
2  .global sumar_arreglo
3  sumar_arreglo:
4  # a0 = int a[], a1 = int largo, t0 = acumulador, t1
       = i
5      li      t0, 0           # acumulador = 0
6      li      t1, 0           # i = 0
7  ciclo:  # Comienzo de ciclo
8      bge     t1, a1, fin     # Si i >= largo, sale del ciclo
9      slli    t2, t1, 2       # Multiplica i por 4 (1 << 2 = 4)
10     add     t2, a0, t2      # Actualiza la dir. de memoria
11     lw      t2, 0(t2)       # De-referencia la dir,
12     add     t0, t0, t2      # Agrega el valor al acumulador
13     addi    t1, t1, 1       # Incrementa el iterador
14     j       ciclo           # Vuelve a comenzar el ciclo
15 fin:
16     mv      a0, t0          # Mueve t0 (acumulador) a a0
17     ret                     # Devuelve valor por a0
```

Lenguaje ensamblador

Lenguajes de alto y bajo nivel

Al programar solemos utilizar **lenguajes de alto nivel**. Estos lenguajes se expresan en un dominio independiente de la arquitectura del procesador donde se vaya a correr el programa.

Lenguajes de alto y bajo nivel

Proveen un nivel de abstracción basado en:

Lenguajes de alto y bajo nivel

Proveen un nivel de abstracción basado en:

- **Variables** que preservan valores (`int a, b = 3;`).

Lenguajes de alto y bajo nivel

Proveen un nivel de abstracción basado en:

- **Variables** que preservan valores (`int a, b = 3;`).
- **Estructuras de control** que permiten modificar la ejecución secuencial del programa (`if`, `switch`, `for`).

Lenguajes de alto y bajo nivel

Proveen un nivel de abstracción basado en:

- **Variables** que preservan valores (`int a, b = 3;`).
- **Estructuras de control** que permiten modificar la ejecución secuencial del programa (`if`, `switch`, `for`).
- Un mecanismo que nos permite **realizar una invocación o llamada a una función** desde cualquier punto del programa, pasando y recibiendo parámetros (`int foo(int bar)`).

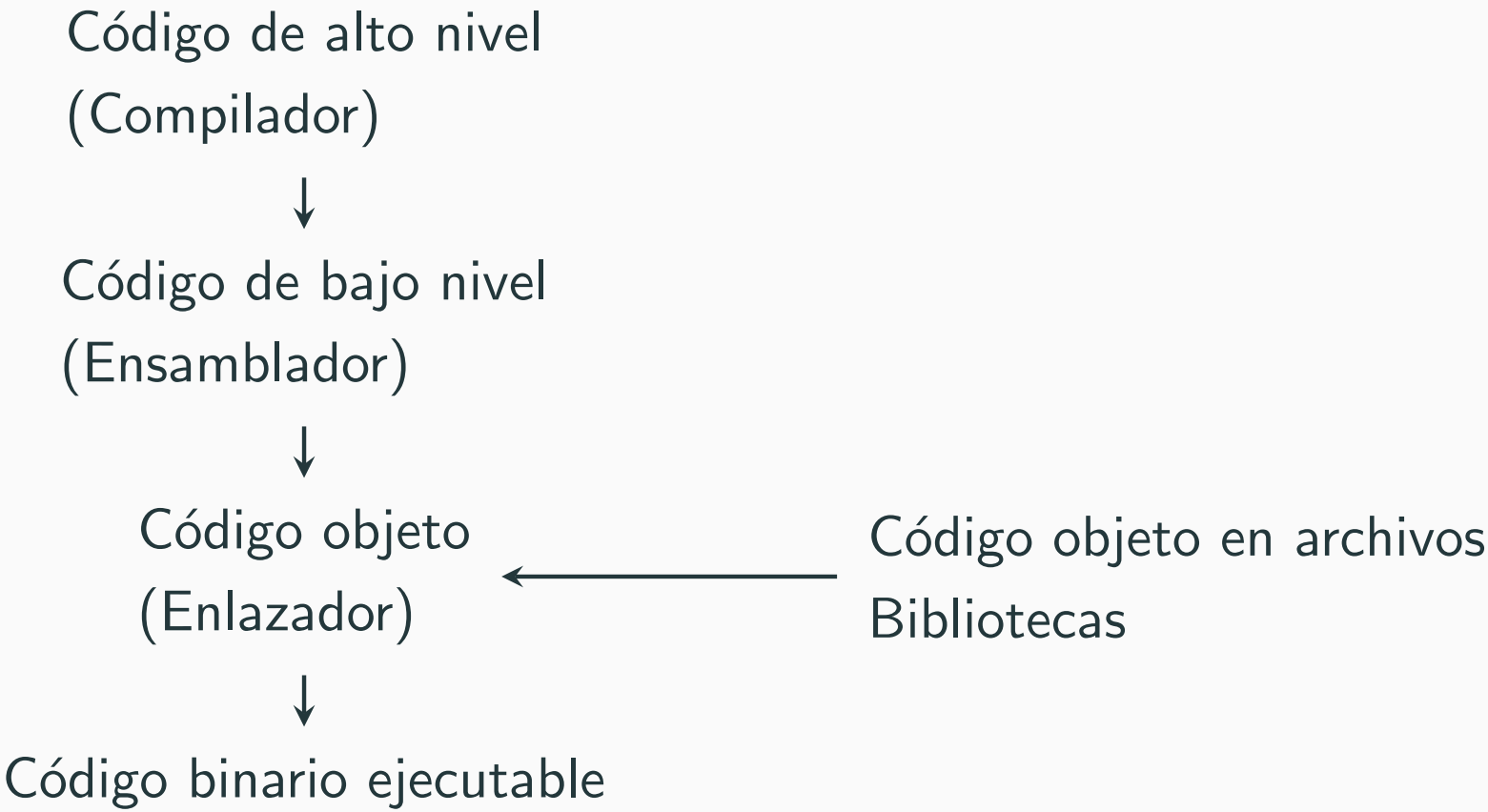
Lenguajes de alto y bajo nivel

Los procesadores pueden ejecutar instrucciones escritas en un lenguaje en particular, que conoce su arquitectura y se expresa estrictamente en términos de sus componentes (instrucciones, registros y memoria). **Este es el lenguaje ensamblador de esta arquitectura RISC V en nuestro caso).**

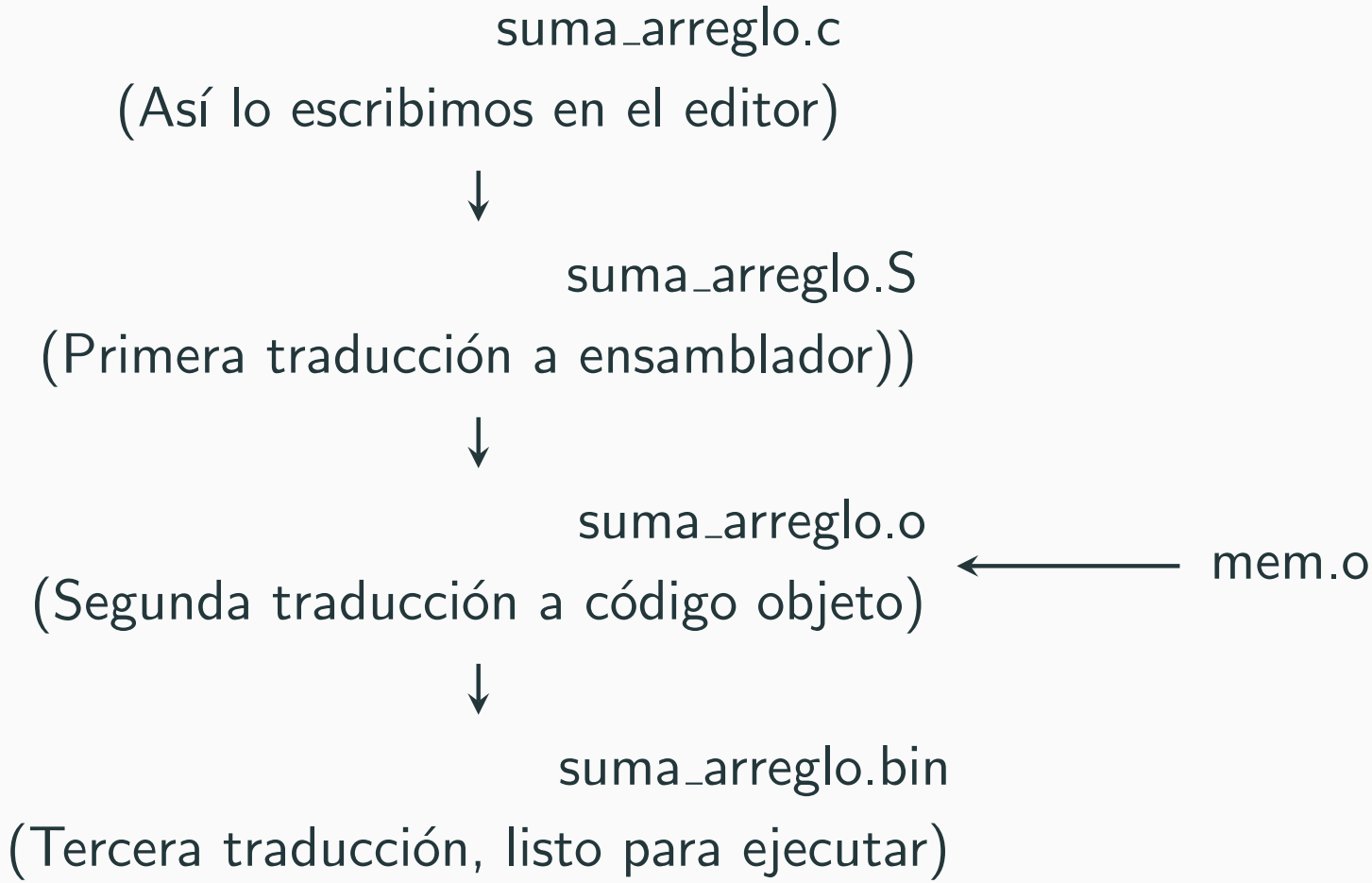
Lenguajes de alto y bajo nivel

Los procesadores implementan una arquitectura y necesitan ser acompañados por programas de **compilado, ensamblado y enlazado** que permiten escribir código en alto nivel y conseguir que éste se traduzca, en una serie de pasos, en código binario ejecutable. En caso contrario solamente podríamos programar en lenguaje ensamblador.

Lenguajes de alto y bajo nivel



Lenguajes de alto y bajo nivel



En nuestro curso vamos a utilizar la arquitectura `RISC V`, que es una arquitectura **abierta, modular, de uso industrial** y que está ganando rápidamente adopción en varios dominios estratégicos.

Instrucciones

Una suma en el lenguaje ensamblador de `RISC V` se escribe de la siguiente manera:

C	RISC V
`a = b + c;`	`add a, b, c`

Instrucciones

Una suma en el lenguaje ensamblador de `RISC V` se escribe de la siguiente manera:

C	RISC V
`a = b + c;`	`add a, b, c`

La primera parte, `add`, recibe el nombre de **mnemónico**, e indica el tipo de operación que queremos realizar, en este caso una suma. Los operandos `b` y `c` son los **operandos de fuente** y **a** el **operando destino** ya que será el que almacene el valor del resultado de la operación.

Instrucciones compuestas

C	RISC V
`// operaciones compuestas` `a = b + c - d;`	`add t, b, c # t = b + c` `sub a, t, d # a = t - d`

Instrucciones compuestas

C	RISC V
`// operaciones compuestas` `a = b + c - d;`	`add t, b, c # t = b + c` `sub a, t, d # a = t - d`

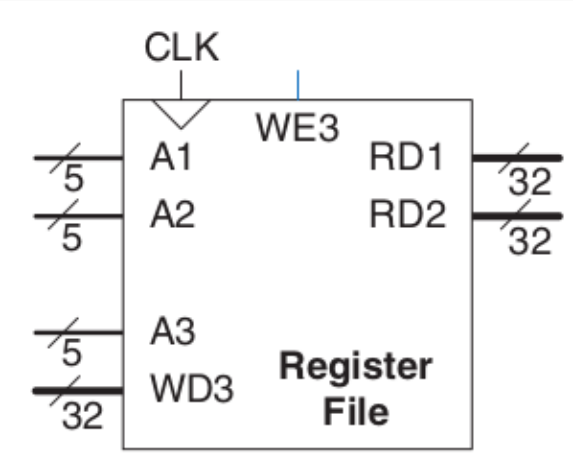
El lenguaje ensamblador no permite la composición de operaciones del modo en que lo hace, por ejemplo, C, por lo que debemos descomponer la operaciones en instrucciones atómicas (una suma y una resta).

Registros

La operaciones lógico aritméticas modifican el estado del procesador según su semántica, dichas modificaciones deben realizarse rápidamente debido a que constistutyen el grueso del cómputo que ocurre en nuestros procesadores. Es por esto que los operandos de fuente y destino suelen ser **registros**.

Register file

`RISC V` cuenta con 32 registros que suelen implementarse como un arreglo de memoria estática de 32 bits con varios puertos. A este arreglo se lo suele referir como banco de registros o archivo de registros (register file).



Nombres de los registros

Los registros pueden nombrarse por su índice, desde `x0` a `x31` o según su uso habitual, que indica el propósito que suele cumplir el registro en el funcionamiento de un programa.

Nombres de los registros

Los registros pueden nombrarse por su índice, desde `x0` a `x31` o según su uso habitual, que indica el propósito que suele cumplir el registro en el funcionamiento de un programa.

- El registro `zero` (`x0`) almacena siempre el valor 0, y no puede ser escrito. Cualquier operación que lo tenga como operando de destino, descarta la escritura del mismo.

Nombres de los registros

Los registros pueden nombrarse por su índice, desde `x0` a `x31` o según su uso habitual, que indica el propósito que suele cumplir el registro en el funcionamiento de un programa.

- El registro `zero` (`x0`) almacena siempre el valor 0, y no puede ser escrito. Cualquier operación que lo tenga como operando de destino, descarta la escritura del mismo.
- Los registros `s0` a `s11` y los `t0` a `t6` se utilizan para almacenar variables.

Nombres de los registros

Los registros pueden nombrarse por su índice, desde `x0` a `x31` o según su uso habitual, que indica el propósito que suele cumplir el registro en el funcionamiento de un programa.

- El registro `zero` (`x0`) almacena siempre el valor 0, y no puede ser escrito. Cualquier operación que lo tenga como operando de destino, descarta la escritura del mismo.
- Los registros `s0` a `s11` y los `t0` a `t6` se utilizan para almacenar variables.
- `ra` y de `a0` a `a7` tienen usos relacionados con las llamadas a función.

Nombres de los registros según su uso

31 … 0	
x0 / zero	Alambrado a cero
x1 / ra	Dirección de retorno
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporal
x6 / t1	Temporal
x7 / t2	Temporal
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Argumento de función, valor de retorno
x11 / a1	Argumento de función, valor de retorno
x12 / a2	Argumento de función
x13 / a3	Argumento de función
x14 / a4	Argumento de función
x15 / a5	Argumento de función
x16 / a6	Argumento de función
x17 / a7	Argumento de función
x18 / s2	Saved register
x19 / s3	Saved register
x20 / s4	Saved register
x21 / s5	Saved register
x22 / s6	Saved register
x23 / s7	Saved register
x24 / s8	Saved register
x25 / s9	Saved register
x26 / s10	Saved register
x27 / s11	Saved register
x28 / t3	Temporal
x29 / t4	Temporal
x30 / t5	Temporal
x31 / t6	Temporal

32

31 … 0
pc

32

Registros y variables

En el lenguaje ensamblador no nos referimos a un conjunto de variables no acotadas y cuyo nombre podemos definir según convenga para la interpretación del programa, sino que contamos con un conjunto fijo de 32 elementos con los que operar. Por eso, cuando traducimos un programa de un lenguaje de alto nivel a ensamblador debemos decidir **en qué registros almacenar los valores de nuestras variables.**

Instrucciones compuestas sobre registros

C	RISC V

`// operaciones compuestas`
`a = b + c - d;`
	`# s0 = a, s1 = b`
	`# s2 = c, s3 = d, t0 = t`
	`add t0, s1, s2  # t = b + c`
	`sub s0, t0, s3  # a = t - d`

Instrucciones compuestas sobre registros

C	RISC V

`// operaciones compuestas`
`a = b + c - d;`
	`# s0 = a, s1 = b`
	`# s2 = c, s3 = d, t0 = t`
	`add t0, s1, s2  # t = b + c`
	`sub s0, t0, s3  # a = t - d`

Volvemos al ejemplo anterior utilizando los nombres reales de los registros sobre los que podemos operar.

Valores inmediatos

Las instrucciones de lenguaje ensamblador pueden tener valores constantes como operandos, suelen llamarse **valores inmediatos** ya que se encuentran disponibles en la misma instrucción (no hace falta recuperar su valor a partir de un registro o desde la memoria). El valor puede escribirse en decimal, hexadecimal (prefijo `0x`) o binario (prefijo `0b`). **Los valores inmediatos son de 12 bits y se extiende su signo a 32 bits antes de operar**.

Operando con constantes

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

`a = a + 4;`
`b = a -12;`
	`#s0=a , s1=b`
	`addi s0 , s0 , 4 # a = a + 4`
	`addi s1 , s0 , -12 # b = a - 12`

Operando con constantes

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

`a = a + 4;`
`b = a -12;`
	`#s0=a , s1=b`
	`addi s0 , s0 , 4 # a = a + 4`
	`addi s1 , s0 , -12 # b = a - 12`

Podemos definir constantes positivas y negativas como operandos utilizando la operación `addi` (`add inmediate`).

Asignando constantes a registros

C	RISC V

`i = 0;`
`x = 2032;`
`y = -78;`
	`#s4=i, s5=x, s6=y`
	`addi s4, zero, 0 # i = 0`
	`addi s5, zero, 2032 # i = 0`
	`addi s6, zero, -78 # i = 0`

Asignando constantes a registros

C	RISC V

`i = 0;`
`x = 2032;`
`y = -78;`
	`#s4=i, s5=x, s6=y`
	`addi s4, zero, 0 # i = 0`
	`addi s5, zero, 2032 # i = 0`
	`addi s6, zero, -78 # i = 0`

Podemos definir constantes positivas y negativas como operandos.

Valores inmediatos de 32 bits

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V
`int a = 0xABCDE123;`	`lui s2, 0xABCDE` *#s2=0xABCDE000* `addi s2, s2, 0x123` *#s2=0xABCDE123*

Valores inmediatos de 32 bits

C	RISC V
`int a = 0xABCDE123;`	`lui s2, 0xABCDE #s2=0xABCDE000` `addi s2, s2, 0x123 #s2=0xABCDE123`

Como los valores inmediatos son de 12 bits y se los extiende respetando el signo a 32 bits cuando realizamos una operación, cargar una constante de 32 bits requiere que hagamos dos operaciones. Primero cargamos los veinte bits más altos con la instrucción `lui`(load upper inmediate) y luego los 12 bits más bajos con un `addi` como veníamos haciendo.

Valores inmediatos de 32 bits

C	RISC V
`int a = 0xFEEDA987;`	`lui s2, 0xFEEDB #s2=0xFEEDB000` `addi s2, s2, -1657 #s2=0xFEEDA987`

Valores inmediatos de 32 bits

C	RISC V
`int a = 0xFEEDA987;`	`lui s2, 0xFEEDB #s2=0xFEEDB000` `addi s2, s2, -1657 #s2=0xFEEDA987`

Si la parte baja se expresa como un número negativo (bit más alto en 1), al extender el signo va a cargar con unos la parte alta. Por eso tenemos que tener esto en cuenta. La parte alta con todos unos equivale a un menos uno en complemento a dos, por lo cual, para compensar el efecto de la extensión del signo en la suma, se incrementa en uno la parte alta que vamos a cargar. En el ejemplo hacemos `lui s2, 0xFEEDB` en lugar de `lui s2, 0xFEEDA`.

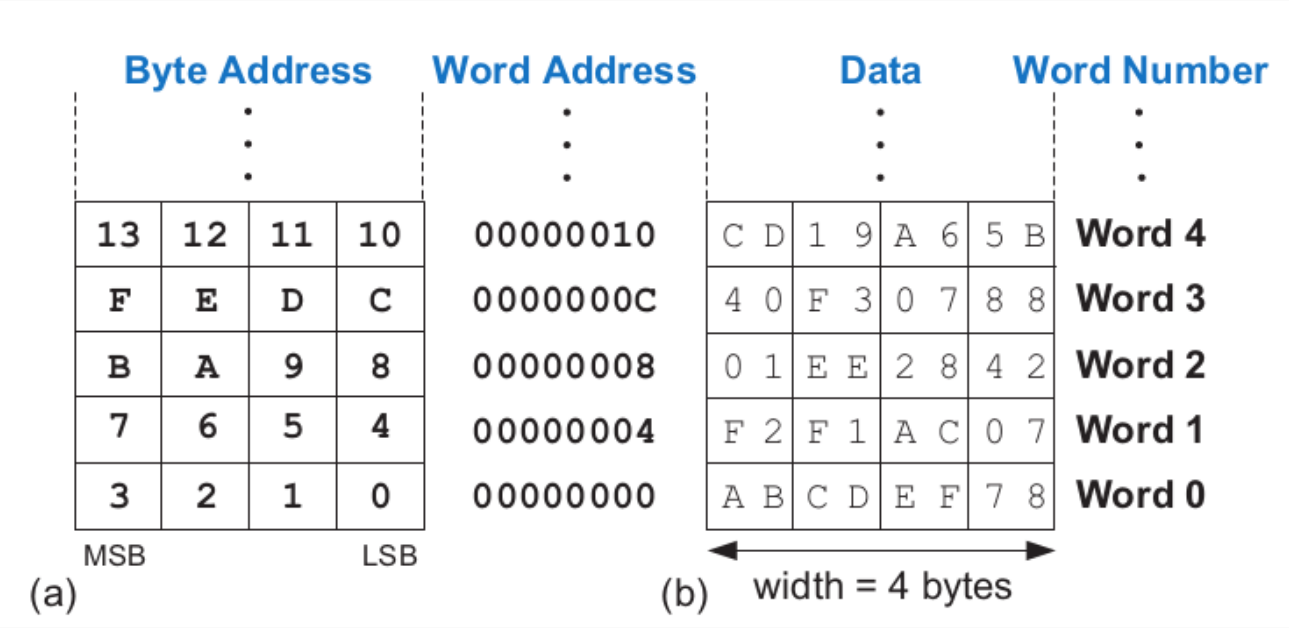
Memoria

El tipo de operando que resta presentar es el de **memoria**. La memoria se estructura y accede como si fuera un arreglo de elementos de 32 bits (4 bytes). El acceso a memoria es significativamente más lento que el acceso a registros pero nos permite acceder a mucha más información que si tuviésemos que operar solamente con registros.

Direcciones de memoria

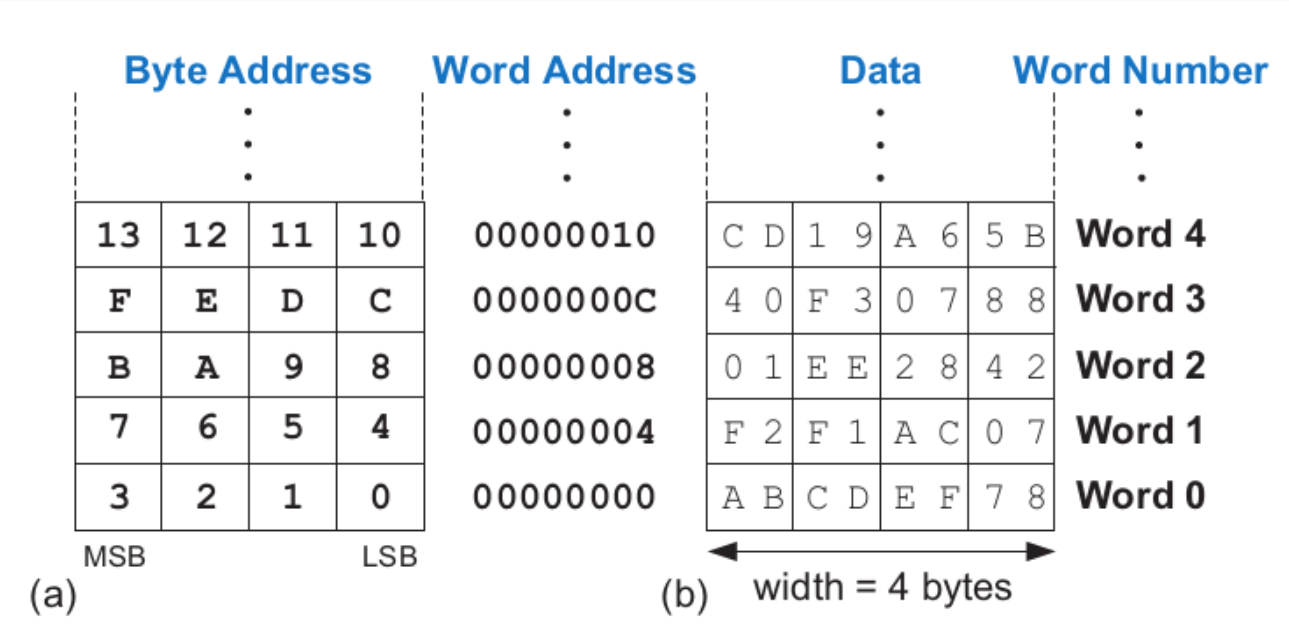
`RISC V` permite acceder a la memoria con índices (direcciones) de 32 bits, o sea 4.294.967.296 índices posibles. **Pero cabe notar que el índice apunta a un byte en particular**, o sea, a uno de los cuatro bytes de la palabra, de modo que entre una palabra de 32 bits y otra, los índices avanzan en cuatro unidades. Podemos indicar que la lectura o escritura se hará en base a un byte en particular.

Accesos a memoria (direcciones)



A la izquierda (a), vemos los índices de memoria (`byte address`) representados de derecha a izquierda, donde a la derecha vemos el byte menos significativo (LSB) y a la derecha el byte más significativo de la palabra (MSB). La dirección de palabra (word address) corresponde al índice del byte menos significativo de ésta.

Accesos a memoria (datos)



A la derecha (b) vemos los datos ordenados según palabras de 32 bits (4 bytes) y el número de palabra (word number). La relación entre número de palabra y dirección de palabra es:

$$\text{word address} * 4 = \text{word number}$$

Leyendo y escribiendo datos

Para operar con la memoria utilizamos las instrucciones `lw` (load word) para leer una palabra de memoria en un registro y `sw` (store word) para escribir una palabra desde un registro a la memoria. Las direcciones se definen como:

$$\text{dirección} = \text{base} + \text{desplazamiento}$$

Donde la base será el valor de un registro y el desplazamiento una constante con signo de 12 bits.

Leyendo un arreglo

C	RISC V
`int a = mem[2];`	`#s7 = a, s3 = mem` `lw, s7, 8(s3)`

Leyendo un arreglo

C	RISC V
`int a = mem[2];`	`#s7 = a, s3 = mem` `lw, s7, 8(s3)`

Si suponemos que los datos del arreglo `mem` son palabras de 4 bytes, y que la posición de memoria en la que comienza el arreglo está almacenada en `s3`, la forma de leer el tercer dato del arreglo (recordemos que el primer dato se encuentra en `mem[0]`) es indicando `s3` como la base y 8 como el desplazamiento, ya que la memoria se accede con índices que apuntan de a byte y cada dato tiene 4 bytes (4 * 2 = 8).

Escribiendo en un arreglo

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

`mem[5] = 33;`	`#s3 = mem`
	`addi t3, zero, 33`
	`sw, t3, 20(s3)`

Escribiendo en un arreglo

C	RISC V

`mem[5] = 33;`	`#s3 = mem`
	`addi t3, zero, 33`
	`sw, t3, 20(s3)`

Si suponemos que los datos del arreglo `mem` son palabras de 4 bytes, y que la posición de memoria en la que comienza el arreglo está almacenada en `s3`, la forma de escribir el quinto dato del arreglo (recordemos que el primer dato se encuentra en `mem[0]`) es indicando `s3` como la base y 20 como el desplazamiento ya que la memoria se accede con índices que apuntan de a byte y cada dato tiene 4 bytes (4 * 5 = 20).

Repaso de lenguaje ensamblador

Hasta este punto se presentó lo siguiente:

Hasta este punto se presentó lo siguiente:

- Definición de **arquitectura**.

Repaso de lenguaje ensamblador

Hasta este punto se presentó lo siguiente:

- Definición de **arquitectura**.
- Definición de lenguajes de alto y bajo nivel.

Repaso de lenguaje ensamblador

Hasta este punto se presentó lo siguiente:

- Definición de **arquitectura**.
- Definición de lenguajes de alto y bajo nivel.
- Lenguaje ensamblador de RISCV.

Repaso de lenguaje ensamblador

Hasta este punto se presentó lo siguiente:

- Definición de **arquitectura**.
- Definición de lenguajes de alto y bajo nivel.
- Lenguaje ensamblador de RISCV.
- Operaciones, operandos, uso de registros, constantes y memoria.

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

¿Qué podemos entender de la traducción que presentamos antes?

Programa de ejemplo en ASM (RISC V)


```
1  .section .text
2  .global sumar_arreglo
3  sumar_arreglo:
4  # a0 = int a[], a1 = int largo, t0 = acumulador, t1
       = i
5      li      t0, 0          # acumulador = 0
6      li      t1, 0          # i = 0
7  ciclo:  # Comienzo de ciclo
8      bge     t1, a1, fin    # Si i >= largo, sale del ciclo
9      slli    t2, t1, 2      # Multiplica i por 4 (1 << 2 = 4)
10     add     t2, a0, t2     # Actualiza la dir. de memoria
11     lw      t2, 0(t2)      # De-referencia la dir,
12     add     t0, t0, t2     # Agrega el valor al acumulador
13     addi    t1, t1, 1      # Incrementa el iterador
14     j       ciclo          # Vuelve a comenzar el ciclo
15 fin:
16     mv      a0, t0         # Mueve t0 (acumulador) a a0
17     ret                    # Devuelve valor por a0
```

Intervalo

Programando con RISCV

Programas almacenados en memoria

Uno de los principios fundamentales de los procesadores es el de **programa almacenado en memoria**, eso significa que las instrucciones que describen el comportamiento de un programa se almacenan (siguiendo un formato particular) en la memoria del procesador, la misma que se accede en las operaciones de lectura y escritura (`sw`,`lw`).

Tamaño de la instrucción

Cada instrucción ocupa 32 bits (una palabra), por lo cual sus direcciones se incrementan en múltiplos de 4, recordemos que la arquitectura RISC V permite acceder a la memoria con direcciones que refieren al byte menos significativo a partir del cual leer o escribir la palabra.

Escribiendo en un arreglo

Dirección	Instrucción almacenada
0x538	`addi s1, s2, 3`
0x53C	`lw t2, 8(s1)`
0x540	`sw s3, 3(t6)`

Escribiendo en un arreglo

Dirección	Instrucción almacenada
0x538	`addi s1, s2, 3`
0x53C	`lw t2, 8(s1)`
0x540	`sw s3, 3(t6)`

El procesador ejecuta el programa almacenando la **posición de memoria de la instrucción que se está ejecutando en un registro de 32 bits conocido como el program counter (PC)**. Va a cargar el contenido de la instrucción de memoria (fetch), ejecutarla (execute) y luego incrementar el PC en 4 posiciones para repetir el ciclo. Al comenzar este programa se carga la instrucción de la posición `0x538`, se la ejecuta, se incrementa el `PC` a `0x53C` y se vuelve a repetir el ciclo.

Program counter

En la sección de control de ejecución condicional veremos la importancia que tiene el valor del program counter.

Instrucciones lógicas

El set de instrucciones de RISC V cuenta con instrucciones lógicas como la conjunción (`and`), disyunción (`or`) y la disyunción excluyente (`xor`).

Instrucciones lógicas

En el diagrama vemos los valores de los registros `s1` y `s2`, representados en formato binario, y luego los resultados de aplicar las operaciones lógicas con distintos operandos de destino utilizando los anteriores como fuente.

	Source registers			
s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
`and s3, s1, s2`	**s3**	0100 0110	1010 0001	0000 0000	0000 0000
`or s4, s1, s2`	**s4**	1111 1111	1111 1111	1111 0001	1011 0111
`xor s5, s1, s2`	**s5**	1011 1001	0101 1110	1111 0001	1011 0111

Instrucciones lógicas

Algunos usos típicos de las isntrucciones lógicas son:

Instrucciones lógicas

Algunos usos típicos de las isntrucciones lógicas son:

- `or`: Combinar dos registros que sólo tienen asignada la parte alta y baja respectivamente, un `or` entre `0xFEED0000` y `0x0000F0CA` resulta en `0xFEEDF0CA`.

Instrucciones lógicas

Algunos usos típicos de las isntrucciones lógicas son:

- `or`: Combinar dos registros que sólo tienen asignada la parte alta y baja respectivamente, un `or` entre `0xFEED0000` y `0x0000F0CA` resulta en `0xFEEDF0CA`.
- `and`: Nos permite limpiar partes de un registro, si quisiéramos preservar solamente la parte baja de `0xBABAC0C0` podemos hacer un `and` con `0x0000FFFF` consiguiendo `0x0000C0C0`.

Instrucciones lógicas

Algunos usos típicos de las isntrucciones lógicas son:

- `or`: Combinar dos registros que sólo tienen asignada la parte alta y baja respectivamente, un `or` entre `0xFEED0000` y `0x0000F0CA` resulta en `0xFEEDF0CA`.
- `and`: Nos permite limpiar partes de un registro, si quisiéramos preservar solamente la parte baja de `0xBABAC0C0` podemos hacer un `and` con `0x0000FFFF` consiguiendo `0x0000C0C0`.
- `xor`: Conseguir la negación lógica al aplicar la operación a -1, recordemos que -1 se codifica con todos 1, por lo que `xori s1, s2, -1` va a aplicar un `xor` entre `s2` y -1 que se codifica como `0xFFF` en 12 bits y se extiende a `0xFFFFFFFF` al ejecutar, consiguiendo un `xor` contra todos unos, que efectivamente niega el valor.

Instrucciones de desplazamiento

Las instrucciones de desplazamiento permiten desplazar un valor a izquierda o derecha en una cantidad definida por el segundo operando fuente, si este segundo operando se trata de un inmediato, lo codifica en 5 bits (complemento a dos extendiendo el signo a 32 bits).

Instrucciones de desplazamiento

Hay tres operaciones posibles:

Instrucciones de desplazamiento

Hay tres operaciones posibles:

- `sll` (shift left logical): desplaza a izquierda el valor tantas veces como especifique el segundo operando fuente, completando con ceros a derecha.

Instrucciones de desplazamiento

Hay tres operaciones posibles:

- `sll` (shift left logical): desplaza a izquierda el valor tantas veces como especifique el segundo operando fuente, completando con ceros a derecha.
- `srl` (shift right logical): desplaza a derecha el valor tantas veces como especifique el segundo operando fuente, completando con ceros a izquierda.

Instrucciones de desplazamiento

Hay tres operaciones posibles:

- `sll` (shift left logical): desplaza a izquierda el valor tantas veces como especifique el segundo operando fuente, completando con ceros a derecha.
- `srl` (shift right logical): desplaza a derecha el valor tantas veces como especifique el segundo operando fuente, completando con ceros a izquierda.
- `sra` (shift right arithmetic): desplaza a derecha el valor tantas veces como especifique el segundo operando fuente, completando con el valor del bit más significativo a izquierda (preserva signo).

Instrucciones de desplazamiento

Hay tres operaciones posibles:

- `sll` (shift left logical): desplaza a izquierda el valor tantas veces como especifique el segundo operando fuente, completando con ceros a derecha.
- `srl` (shift right logical): desplaza a derecha el valor tantas veces como especifique el segundo operando fuente, completando con ceros a izquierda.
- `sra` (shift right arithmetic): desplaza a derecha el valor tantas veces como especifique el segundo operando fuente, completando con el valor del bit más significativo a izquierda (preserva signo).

Existen versiones donde el segundo operando fuente es un inmediato (`slli, srli, srai`).

Instrucciones de desplazamiento

En el diagrama vemos los valores del registro s5, representado en formato binario, y luego los resultados de aplicar las operaciones de desplazamiento.

		Source register			
	s5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly code		Result			
slli t0, s5, 7	t0	1000 1110	0000 1000	0111 0011	1000 0000
srli s1, s5, 17	s1	0000 0000	0000 0000	0111 1111	1000 1110
srai t2, s5, 3	t2	1111 1111	1110 0011	1000 0010	0001 1100

Consiguiendo un byte en particular

Utilizando desplazamientos y máscaras podemos acceder a un byte en particular dentro de una palabra, si tenemos el valor `0xABCDEF00` en el registro `s1` y queremos conseguir el segundo byte (desde el menos significativo) y almacenarlo en `s2` podemos hacer lo siguiente:

```
1 srli t0, s1, 8
2 andi s2, t0, 0xFF
```

La primera instrucción desplaza el valor un byte a la derecha y la segunda preserva solamente el byte menos significativo, que luego almacena en `s2`.

Control del flujo de ejecución

Para poder ejecutar programas que no tengan un flujo secuencial (donde todas las instrucciones se suceden en orden), necesitamos poder saltear instrucciones en nuestro programa o volver a una instrucción anterior, como suele suceder en los lenguajes de alto nivel con las estructuras de `if`, `while`, `for`, `case`. El mecanismo para conseguir esto en el lenguaje ensamblador de `RISC V` es modificar el valor del registro `PC` (program counter) de modo que la próxima instrucción no sea la siguiente en la memoria sino la que se defina en una instrucción específica.

Instrucciones de control del flujo de ejecución

Las instrucciones de control de flujo van a comparar el valor de los dos primeros operandos, y en función del resultado van reemplazar el valor del `PC` con el del tercer operando. Las instrucciones son:

Instrucciones de control del flujo de ejecución

Las instrucciones de control de flujo van a comparar el valor de los dos primeros operandos, y en función del resultado van reemplazar el valor del `PC` con el del tercer operando. Las instrucciones son:

- `beq`(branch if equal): que reemplaza el valor del `PC` si los dos primeros operandos son iguales.

Instrucciones de control del flujo de ejecución

Las instrucciones de control de flujo van a comparar el valor de los dos primeros operandos, y en función del resultado van reemplazar el valor del `PC` con el del tercer operando. Las instrucciones son:

- `beq`(branch if equal): que reemplaza el valor del `PC` si los dos primeros operandos son iguales.
- `bne`(branch if not equal): que reemplaza el valor del `PC` si los dos primeros operandos son distintos.

Instrucciones de control del flujo de ejecución

Las instrucciones de control de flujo van a comparar el valor de los dos primeros operandos, y en función del resultado van reemplazar el valor del `PC` con el del tercer operando. Las instrucciones son:

- `beq`(branch if equal): que reemplaza el valor del `PC` si los dos primeros operandos son iguales.
- `bne`(branch if not equal): que reemplaza el valor del `PC` si los dos primeros operandos son distintos.
- `blt`(branch if less than): que reemplaza el valor del `PC` si el primer operando es menor que el segundo.

Instrucciones de control del flujo de ejecución

Las instrucciones de control de flujo van a comparar el valor de los dos primeros operandos, y en función del resultado van reemplazar el valor del `PC` con el del tercer operando. Las instrucciones son:

- `beq`(branch if equal): que reemplaza el valor del `PC` si los dos primeros operandos son iguales.
- `bne`(branch if not equal): que reemplaza el valor del `PC` si los dos primeros operandos son distintos.
- `blt`(branch if less than): que reemplaza el valor del `PC` si el primer operando es menor que el segundo.
- `bge`(branch if greater than or equal): que reemplaza el valor del `PC` si el primer operando es mayor o igual que el segundo.

Instrucciones de control sobre operandos sin signo

Existen variantes que interpretan a los operandos como enteros sin signo a la hora de realizar las comparaciones. Sus mnemónicos son `bltu, bgeu`.

Ejemplo y etiquetas


```
1  addi s0, zero, 4
2  addi s1, zero, 1
3  slli s1, s1, 2
4  beq s0, s1, target
5  addi s1, s1, 1
6  sub s1, s1, s0
7  target:
8  add s1, s1, s0
```

Ejemplo y etiquetas


```
1  addi s0, zero, 4
2  addi s1, zero, 1
3  slli s1, s1, 2
4  beq s0, s1, target
5  addi s1, s1, 1
6  sub s1, s1, s0
7  target:
8  add s1, s1, s0
```

Este ejemplo carga un 4 en `s0` y un 1 en `s1` (`addi`), luego desplaza a `s1` dos posiciones a la izquierda (`slli`), lo cual equivale a multiplicar por 4 y compara sin ambos registros son iguales (`beq`). El último operando es de tipo etiqueta. Las etiquetas se definen como `nombre:` donde `nombre` es la referencia que podemos usar en otras instrucciones y será interpretada como la dirección de memoria donde se almacena la instrucción inmediatamente siguiente a su definición.

Ejemplo y etiquetas


```
1  addi s0, zero, 4
2  addi s1, zero, 1
3  slli s1, s1, 2
4  beq s0, s1, target
5  addi s1, s1, 1
6  sub s1, s1, s0
7 target:
8  add s1, s1, s0 #dir: 0xB400
```

Ejemplo y etiquetas


```
1  addi s0, zero, 4
2  addi s1, zero, 1
3  slli s1, s1, 2
4  beq s0, s1, target
5  addi s1, s1, 1
6  sub s1, s1, s0
7 target:
8  add s1, s1, s0 #dir: 0xB400
```

Si la instrucción `add s1, s1, s0` se encuentra almacenada en la dirección `0xB400`, al evaluar la condición en `beq s0, s1, target` y determinar que los valores de los operandos son iguales, el `PC` será actualizado con el valor `0xB400` y la próxima instrucción a ejecutar será `add s1, s1, s0` en lugar de `addi s1, s1, 1`.

Saltos incondicionales

En los casos anteriores el valor del PC se actualizaba solamente cuando se cumplía una condición luego de comparar el valor de dos operandos. Para realizar una actualización (salto) incondicional del valor del PC se utilizan las instrucciones:

Saltos incondicionales

En los casos anteriores el valor del `PC` se actualizaba solamente cuando se cumplía una condición luego de comparar el valor de dos operandos. Para realizar una actualización (salto) incondicional del valor del `PC` se utilizan las instrucciones:

- `j` (jump): que simplemente actualiza el valor del `PC` con el del operando provisto (inmediato de 20 bits extendidos en signo a 32).

Saltos incondicionales

En los casos anteriores el valor del `PC` se actualizaba solamente cuando se cumplía una condición luego de comparar el valor de dos operandos. Para realizar una actualización (salto) incondicional del valor del `PC` se utilizan las instrucciones:

- `j` (jump): que simplemente actualiza el valor del `PC` con el del operando provisto (inmediato de 20 bits extendidos en signo a 32).
- `jal` (jump and link): que almacena el valor actual del `PC` en el registro indicado en el primer operando y actualiza el valor del `PC` con el del segundo operando (inmediato de 20 bits extendidos en signo a 32).

Saltos incondicionales


```
1  j target
2  srai s1, s1, 2
3  addi s1, s1, 1
4  sub s1, s1, s0
5 target:
6  add s1, s1, s0
```

Saltos incondicionales


```
1  j target
2  srai s1, s1, 2
3  addi s1, s1, 1
4  sub s1, s1, s0
5 target:
6  add s1, s1, s0
```

En este ejemplo la segunda, tercera y cuarta instrucción no se ejecutan, ya que el salto incondicional de la primera instrucción contínua la ejecución en `add s1, s1, s0`.

Escribiendo un loop

C	RISC V

`// calcula el valor de x`
`// tal que 2 a la x es 128`
`int pow = 1;`
`int x = 0;`
 	
`while(pow != 128){`
`  pow = pow * 2;`
`  x = x + 1;`
`}`
	`#s0=pow, s1=x`
	`addi s0, zero, 1`
	`add s1, zero, zero`
	`#t0=128`
	`addi t0, zero, 128`
	`while:`
	`beq s0, t0, fin`
	`slli s0, s0, 1 #pow=pow*2`
	`addi s1, s1, 1 #x+=1`
	`j while`
	`fin:`

Escribiendo un loop

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

`// calcula el valor de x`
`// tal que 2 a la x es 128`
`int pow = 1;`
`int x = 0;`
 	
`while(pow != 128){`
`  pow = pow * 2;`
`  x = x + 1;`
`}`
	`#s0=pow, s1=x`
	`addi s0, zero, 1`
	`add s1, zero, zero`
	`#t0=128`
	`addi t0, zero, 128`
	`while:`
	`beq s0, t0, fin`
	`slli s0, s0, 1 #pow=pow*2`
	`addi s1, s1, 1 #x+=1`
	`j while`
	`fin:`

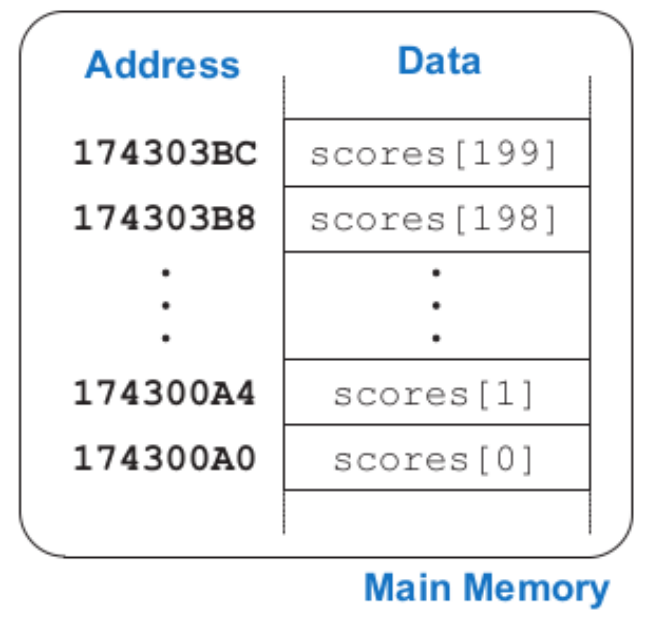
Esta traducción indica como podemos implementar un ciclo `while` con un salto condicional y uno incondicional.

Estructura de los arreglos

Los arreglos son estructuras que ubican elementos del mismo tamaño y tipo de forma consecutiva en la memoria del procesador. En un lenguaje de alto nivel, la forma de acceder a un elemento es a partir de una dirección base y la posición en el arreglo, a la que llamamos su índice. La forma de acceder en lenguaje ensamblador es calculando el desplazamiento desde la dirección del comienzo del arreglo hasta la dirección en la que se encuentra el elemento.

Acceso a un elemento del arreglo

En este ejemplo el arreglo `scores` contiene 200 elementos de 32 bits y comienza en la dirección `0x174300A0`. La forma de acceder al i-ésimo elemento es cargando el dato que se encuentra en `base + tamaño * índice`, en este caso, si queremos acceder al elemento 199 sería `0x174300A0 + 4 * 198 = 0x174303B8`.



Acceso a un elemento del arreglo

En el siguiente ejemplo se incrementa el valor de cada elemento del arreglo en 10.

Incrementando los valores de un arreglo

C

```c
int i;
int scores[200];

for(i = 0; i < 200; i = i + 1){
    scores[i] = scores[i] + 10;
}
```

RISC V

```
#s0=dir. scores, s1=i
 addi s1, zero, 0
 addi t2, zero, 200
for:
 bge s1, t2, fin
 slli t0, s1, 2
 add t0, t0, s0
 lw t1, 0(t0)
 addi t1, t1, 10
 sw t1, 0(t0)
 addi s1, s1, 1
 j for
fin:
```

Llamadas a función, argumentos

En un lenguaje de alto nivel los programas se dividen en funciones que pueden llamarse unas o otras. Para implementar esta funcionalidad se debe decidir de qué manera una función puede identificar a otra y cómo se enviarán los parámetros de entrada y de salida. Los parámetros de entrada serán llamados **argumentos** y los de salida **valor de retorno**.

Llamadas a función, jal

En `RISC V` la **función llamadora** puede utilizar los registros `a0` hasta `a7` para enviar argumentos y luego la **función llamada** utiliza `a0` para copiar el valor de retorno. A la hora de invocar la ejecución de una función la **función llamadora** debe almacenar el `PC` en `ra`. Esto se consigue utilizando la instrucción `jal ra, foo`, donde `foo` es la **función llamada**.

Llamadas a función, preservando estado

La **función llamada** no debe interferir con el estado de la **función llamadora**, debido a esto debe respetar los valores de los registros guardados (`s0` a `s11`) y el registro de la dirección de retorno (`ra`), que indica cómo retornar la ejecución a la **función llamadora**.
También debe mantenerse invariante la porción de memoria (stack) correspondiente a **función llamadora**.

Ejemplo de llamada

C	RISC V

`int main(){`
`  simple();`
`  ...`
`}`
 	
`void simple(){`
`  return;`
`}`
	`0x00000300 main: jal ra, simple`
	`0x00000304 ...`
	`... ...`
	`0x0000051c simple: jr ra`

Ejemplo de llamada

C	RISC V

`int main(){`
`  simple();`
`  ...`
`}`
 	
`void simple(){`
`  return;`
`}`
	`0x00000300 main: jal ra, simple`
	`0x00000304 ...`
	`... ...`
	`0x0000051c simple: jr ra`

Un ejemplo de llamada a `simple` y un retorno con un salto incondicional al registro de la direccón de retorno `jr ra`.

Ejemplo con argumentos

A continuación presentamos un ejemplo que involucra argumentos.

Ejemplo con argumentos

C

```c
int main(){
 int y;
 ...
 y = dif_sumas(2,3,4,5);
 ...
}
int dif_sumas(int f, int g,
 int h, int i){
 int resultado;
 resultado = (f+g)-(h+i);
 return resultado;
}
```

RISC V

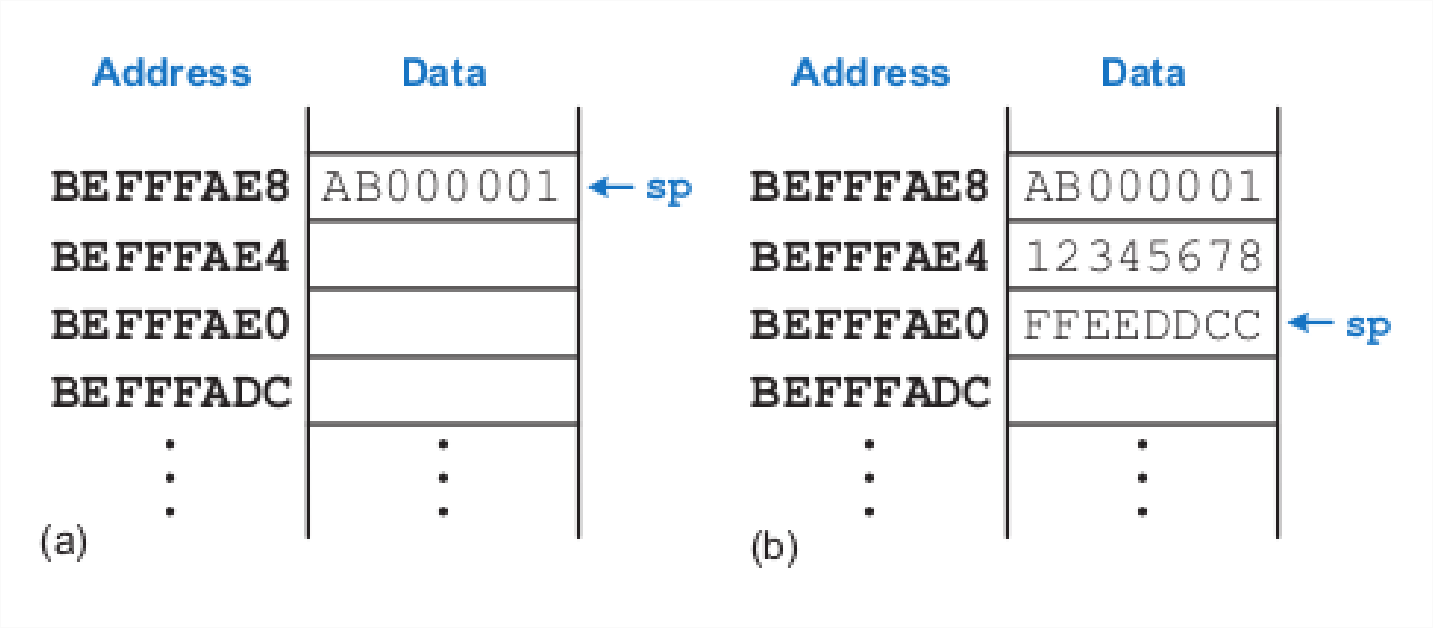
```
main:#s7=y
 addi a0, zero, 2
 addi a1, zero, 3
 addi a2, zero, 4
 addi a3, zero, 5
 jal dif_sumas
 add s7, a0, zero
dif_sums:#s3=result
 add t0, a0, a1
 add t1, a2, a3
 sub s3, t0, t1
 add a0, s3, zero
 jr ra
```

La pila (stack)

La pila es una parte de la memoria que se utiliza para almacenar información temporaria, se utiliza con un esquema LIFO (el último elemento ingresado es el primero en retirarse), como una pila de valores. La semántica de uso es a través de operación de agregado (`push`) y retiro (`pop`) de un elemento (tope) de la pila. La pila suele comenzar en las direcciones altas de la memoria y va tomando (con cada push) las direcciones inmediatamente más bajas. Por eso se suele decir que la pila crece hacia abajo.

Stack pointer (sp)

Al igual que en muchas otras arquitecturas, `RISC V` propone el uso de uno de sus registros, `sp` (stack pointer), para indicar la dirección de topo de pila. En este ejemplo vemos como se actualiza la pila (y el stack pointer) luego de agegar dos palabras de 32 bits (`0x12345678` y `0xFFEEDDCC`) cambiando el `sp` de `0xBEFFFAE8` a `0xBEFFFAE0` (`sp` apunta al último elemento cargado).



La pila y las llamadas a función

Habíamos dicho que al llamar a una función había un acuerdo entre la **función llamadora** (la que inicia la llamada) y la **función llamada** (la que la recibe), donde se preservaba parte del estado del procesador entre el llamado y el retorno.

Reglas de llamada

Preserved (*callee*-saved)	Nonpreserved (*caller*-saved)
Saved registers: `s0-s11`	Temporary registers: `t0-t6`
Return address: `ra`	Argument registers: `a0-a7`
Stack pointer: `sp`	
Stack above the stack pointer	Stack below the stack pointer

Estado del procesador entre llamadas

Reglas de preservación de estado:

Estado del procesador entre llamadas

Reglas de preservación de estado:

- **Regla para la llamadora:** Antes de llamar debe guardar los valores de los registros temporarios que necesite utilizar al retornar (`t0-t6, a0-a7`).

Estado del procesador entre llamadas

Reglas de preservación de estado:

- **Regla para la llamadora:** Antes de llamar debe guardar los valores de los registros temporarios que necesite utilizar al retornar (`t0-t6, a0-a7`).
- **Regla para la llamada:** Si va a utilizar los registros permanentes (`s0-s11, ra`) debe guardarlos al comenzar y restaurarlos antes de retornar.

Estado del procesador entre llamadas

Reglas de preservación de estado:

- **Regla para la llamadora:** Antes de llamar debe guardar los valores de los registros temporarios que necesite utilizar al retornar (`t0-t6, a0-a7`).
- **Regla para la llamada:** Si va a utilizar los registros permanentes (`s0-s11, ra`) debe guardarlos al comenzar y restaurarlos antes de retornar.

Para esto podemos utilizar la pila.

Ejemplo recursivo

C

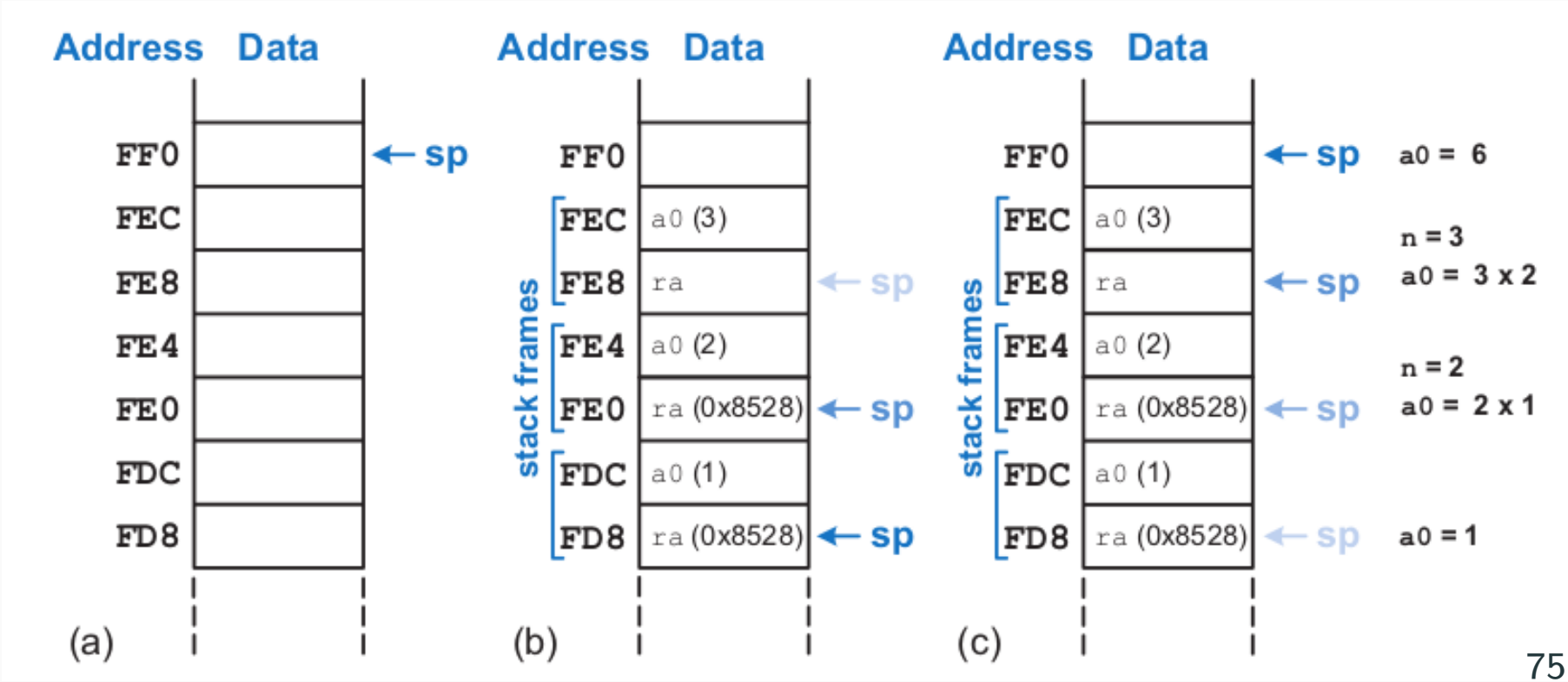
```c
int factorial(int n){
 if(n <= 1){
  return 1;
 }else{
  return
   (n*factorial(n-1));
 }
}
```

RISC V

```
factorial:addi sp, sp, -8
 sw a0, 4(sp) #guarda a0
 sw ra, 0(sp) #guarda ra
 addi t0, zero, 1
 bgt a0, t0, else
 addi a0, zero, 1
 addi sp, sp, 8
 jr ra
else: addi a0, a0, -1
 jal factorial
 lw t1, 4(sp)
 lw ra, 0(sp)
 addi sp, sp, 8
 mul a0, t1, a0
 jr ra
```

Stack pointer (sp)

Podemos ver como cada llamada recursiva utiliza una porción de la pila para preservar su estado y así cumplir con las reglas antes mencionadas, al espacio de la pila utilizado por la llamada en cuestión lo llamamos **marco de pila o stack frame**.



Pseudoinstrucciones

Algunas de las instrucciones empleadas en el lenguaje ensamblador no son verdaderamente instrucciones, en el sentido de que el procesador no sabe interpretarlas, sino que es el compilador el que se encarga de traducir una de estas así llamadas **pseudointstrucción** en una instrucción propiamente dicha. El uso de las pseudoinstrucciones se debe a que encapsulan operaciones comunes y convenientes pero que no justifican su inclusión en el set de instrucciones de la arquitectura si queremos mantenerlo acotado.

Ejemplos de pseudoinstrucciones

`j label`	`jal zero, label`
`jr ra`	`jalr zero, ra, 0`
`mv t5, s3`	`addi t5, s3, 0`
`not s7, t2`	`xori s7, t2, -1`
`nop`	`addi zero, zero, 0`
`li s8, 0x7EF`	`addi s8, zero, 0x7EF`
`li s8, 0x56789DEF`	`lui s8, 0x5678A` `addi s8, s8, 0xDEF`
`bgt s1, t3, L3`	`blt t3, s1, L3`
`bgez t2, L7`	`bge t2, zero, L7`
`call L1`	`jal L1`
`call L5`	`auipc ra,` $imm_{31:12}$ `jalr ra, ra,` $imm_{11:0}$
`ret`	`jalr zero, ra, 0`

Recomendación a la hora de programar

No intenten memorizar los nombres de todas las instrucciones y su semántica, tengan la documentación mientras escriben o hacen seguimiento de sus programas de lenguaje ensamblador:

- Hoja con lista de registros e instrucciones.
- Reglas de llamada a función.
- Estructura de la memoria.

Recomendación a la hora de programar

Vuelvan a revisar el material de lectura (manuales, clases y apuntes) tantas veces como haga falta. **Hacer repetidas lecturas de la documentación es parte de la práctica de la ingeniería.**

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

¿Qué podemos entender de la traducción que presentamos antes?

Programa de ejemplo en ASM (RISC V)


```
1  .section .text
2  .global sumar_arreglo
3  sumar_arreglo:
4  # a0 = int a[], a1 = int largo, t0 = acumulador, t1
       = i
5      li      t0, 0           # acumulador = 0
6      li      t1, 0           # i = 0
7  ciclo:  # Comienzo de ciclo
8      bge     t1, a1, fin     # Si i >= largo, sale del ciclo
9      slli    t2, t1, 2       # Multiplica i por 4 (1 << 2 = 4)
10     add     t2, a0, t2      # Actualiza la dir. de memoria
11     lw      t2, 0(t2)       # De-referencia la dir,
12     add     t0, t0, t2      # Agrega el valor al acumulador
13     addi    t1, t1, 1       # Incrementa el iterador
14     j       ciclo           # Vuelve a comenzar el ciclo
15 fin:
16     mv      a0, t0          # Mueve t0 (acumulador) a a0
17     ret                     # Devuelve valor por a0
```

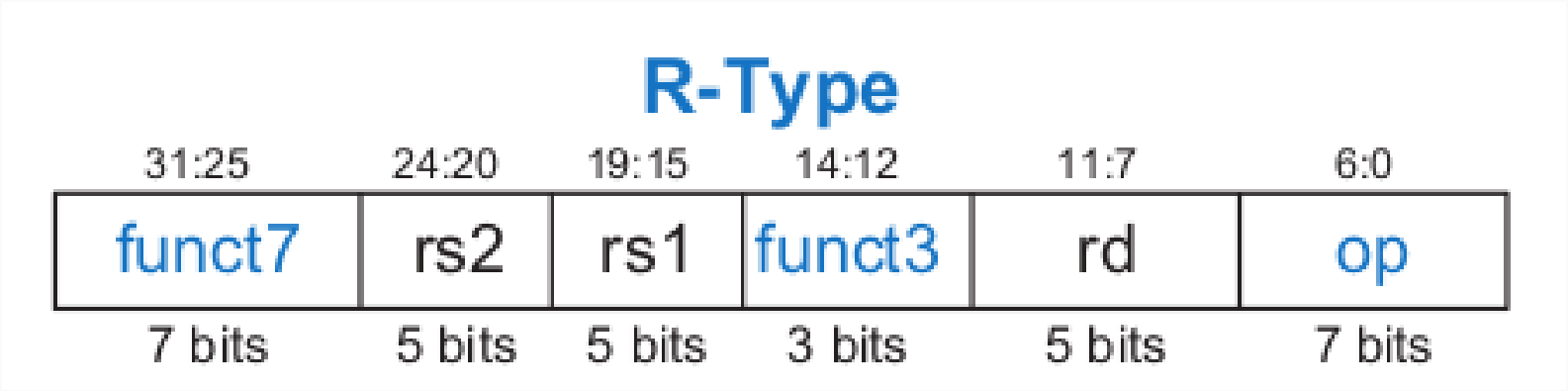
Intervalo

Lenguaje de máquina

Lenguaje de máquina

El lenguaje ensamblador es un lenguaje de bajo nivel pero los programas escritos en este lenguaje no pueden ser ejecutados por el procesador, es por eso que el código fuente debe ser ensamblado para producir el archivo binario cuyos contenidos pueden ser cargados en memoria y ejecutados.

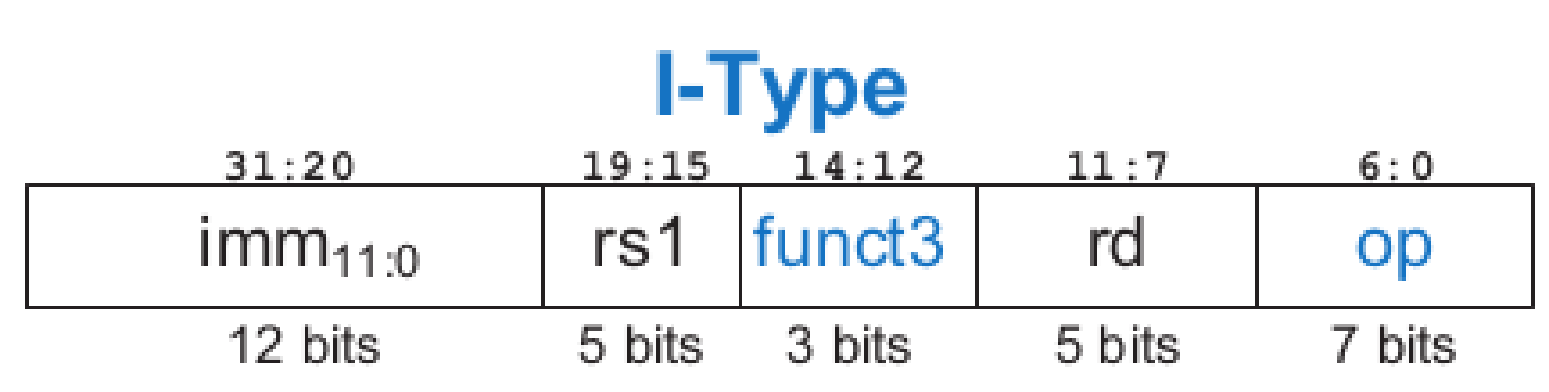
Las instrucciones de tipo `R` utilizan dos registros como operandos fuente (`rs1, rs2`) y uno como operando destino `rd`. El campo `op` junto con `funct7` y `funct3` determinan el tipo de instrucción codificada.



Instrucciones R, ejemplos

Assembly	Field Values					
	funct7	rs2	rs1	funct3	rd	op
`add s2, s3, s4` `add x18,x19,x20`	0	20	19	0	18	51
`sub t0, t1, t2` `sub x5, x6, x7`	32	7	6	0	5	51
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Las instrucciones de tipo `I` utilizan un registros como operando fuente (`rs1`), un inmediato de 12 bits (`imm`) y uno como operando destino `rd`. El campo `op` junto con `funct3` determinan el tipo de instrucción codificada.

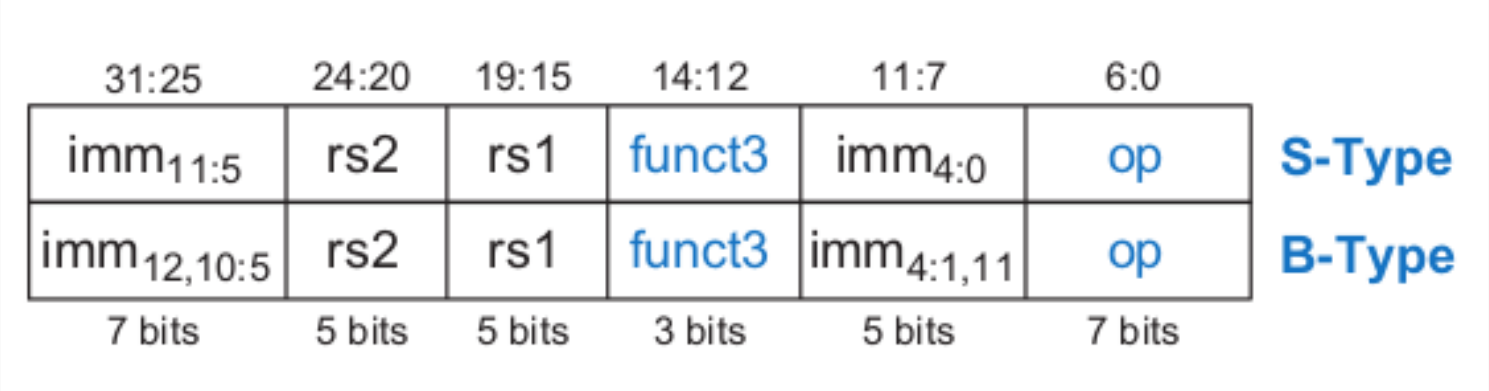


Instrucciones I, ejemplos

Assembly	$imm_{11:0}$	rs1	funct3	rd	op
`addi s0, s1, 12` `addi x8, x9, 12`	12	9	0	8	19
`addi s2, t1, -14` `addi x18,x6, -14`	-14	6	0	18	19
`lw t2, -6(s3)` `lw x7, -6(x19)`	-6	19	2	7	3
`lb s4, 0x1F(s4)` `lb x20,0x1F(x20)`	0x1F	20	0	20	3
`slli s2, s7, 5` `slli x18, x23, 5`	5	23	1	18	19
`srai t1, t2, 29` `srai x6, x7, 29`	(upper 7 bits = 32) 29	7	5	6	19
	12 bits	5 bits	3 bits	5 bits	7 bits

Instrucciones S/B

Las instrucciones de carga (`S`) y de saltos condicionales (`B`) se codifican como se indica a continuación. Ambos formatos codifican un inmediato en la instrucción, en el caso de las isntrucciones de carga es de 12 bits, en los saltos condicionales es de 13 bits y expresa el desplazamiento en complemento a 2 al que se debe saltar en relación al valor actual del `PC`. Este desplazamiento (`offset`) siempre se desplaza una posición a izquierda antes de sumarlo al `PC` ya que se encuentra siempre en posiciones pares.



Instrucciones S, ejemplos

Assembly	Field Values					
	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
`sw t2, -6(s3)` `sw x7, -6(x19)`	1111 111	7	19	2	11010	35
`sh s4, 23(t0)` `sh x20,23(x5)`	0000 000	20	5	1	10111	35
`sb t5, 0x2D(zero)` `sb x30,0x2D(x0)`	0000 001	30	0	0	01101	35
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Instrucciones B, ejemplos


```
#Address   # RISC-V Assembly
0x70       beq  s0, t5, L1   1
0x74       add  s1, s2, s3   2
0x78       sub  s5, s6, s7   3
0x7C       lw   t0, 0(s1)    4
0x80  L1:  addi s1, s1, -15
```

`L1` is 4 instructions (i.e., **16 bytes**) past `beq`

$imm_{12:0}$ = 16	0	0	0	0	0	0	0	0	1	0	0	0	0
bit number	12	11	10	9	8	7	6	5	4	3	2	1	0

Assembly

```
beq s0, t5,  L1
beq x8, x30, 16
```

Field Values

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0000 000	30	8	0	1000 0	99
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

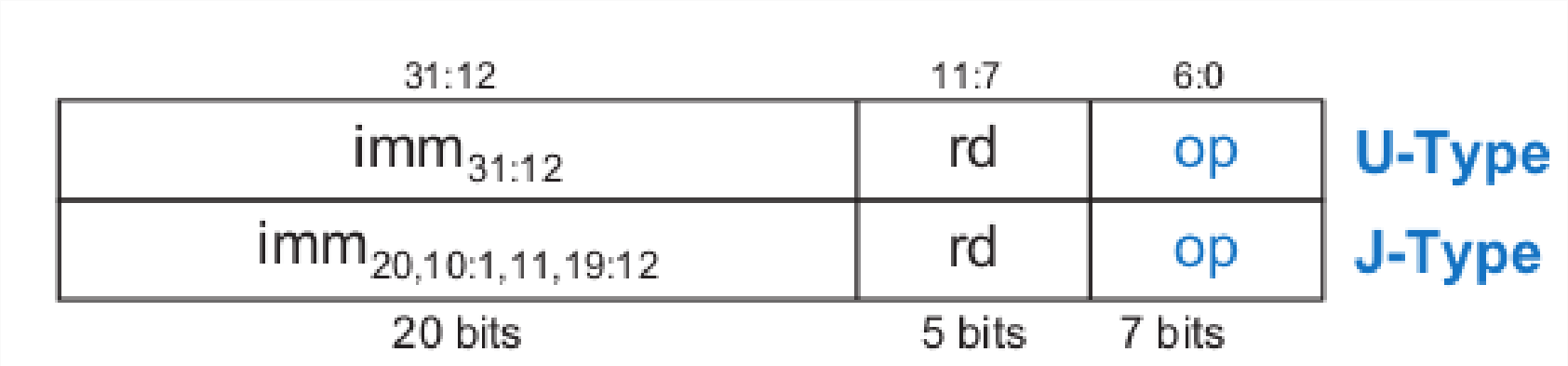
Machine Code

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0000 000	11110	01000	000	1000 0	110 0011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

(0x01E40863)

Instrucciones U/J

Las instrucciones de inmediato superior (`U`) y de saltos incondicionales (`J`) se codifican como se indica a continuación. Ambos formatos codifican un inmediato en la instrucción, en el caso de las instrucciones de inmediato superior es de 20 bits, en los saltos incondicionales es de 21 bits y expresa el valor de los 21 bits más altos de la dirección a la que se debe saltar en relación al valor actual del `PC`. Este desplazamiento (`offset`) siempre se desplaza una posición a izquierda antes de sumarlo al `PC` ya que se encuentra siempre en posiciones pares.



Instrucciones U, ejemplos

Assembly	Field Values		
	$imm_{31:12}$	rd	op
`lui s5, 0x8CDEF` `lui x21,0x8CDEF`	0x8CDEF	21	55
	20 bits	5 bits	7 bits

Instrucciones J, ejemplos


```
# Address          RISC-V Assembly
0x0000540C         jal ra, func1
0x00005410         add s1, s2, s3
...                ...

0x000ABC04  func1: add s4, s5, s8
...                ...
```

`func1` is **0xA67F8 bytes** past `jal`

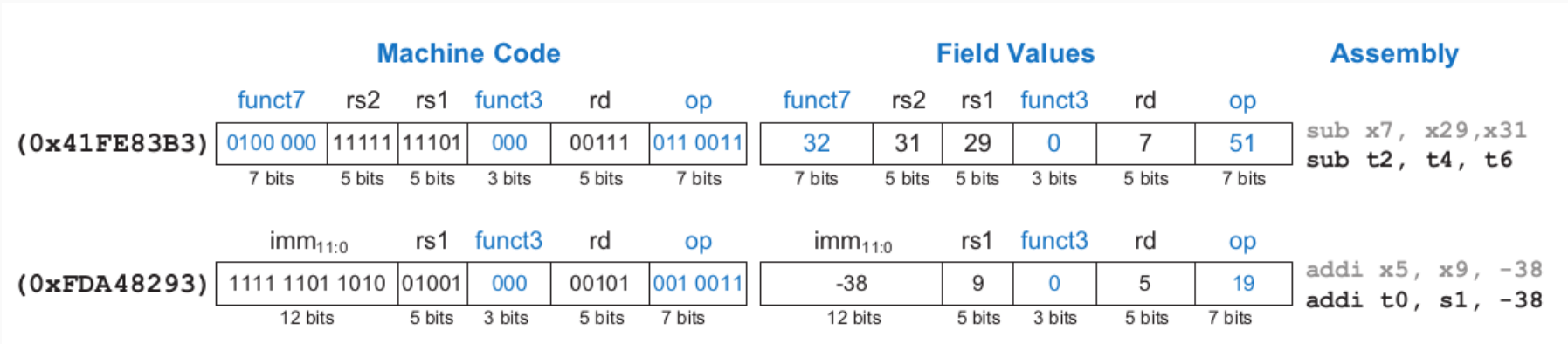
imm = 0xA67F8	0	1	0	1	0	0	1	1	0	0	1	1	1	1	1	1	1	0	0	0
bit number	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Assembly	Field Values			Machine Code			
	$imm_{20,10:1,11,19:12}$	rd	op	$imm_{20,10:1,11,19:12}$	rd	op	
`jal ra, func1` `jal x1, 0xA67F8`	0111 1111 1000 1010 0110	1	111	0111 1111 1000 1010 0110	00001	110 1111	(0x7F8A60EF)
	20 bits	5 bits	7 bits	20 bits	5 bits	7 bits	

Decodificación

Es importante comprender el formato con el que se codifican las instrucciones al traducirlas al lenguaje máquina para poder realizar tanto la codificación como la decodificación de las mismas en caso de ser necesario.

Ejemplo de decodificación

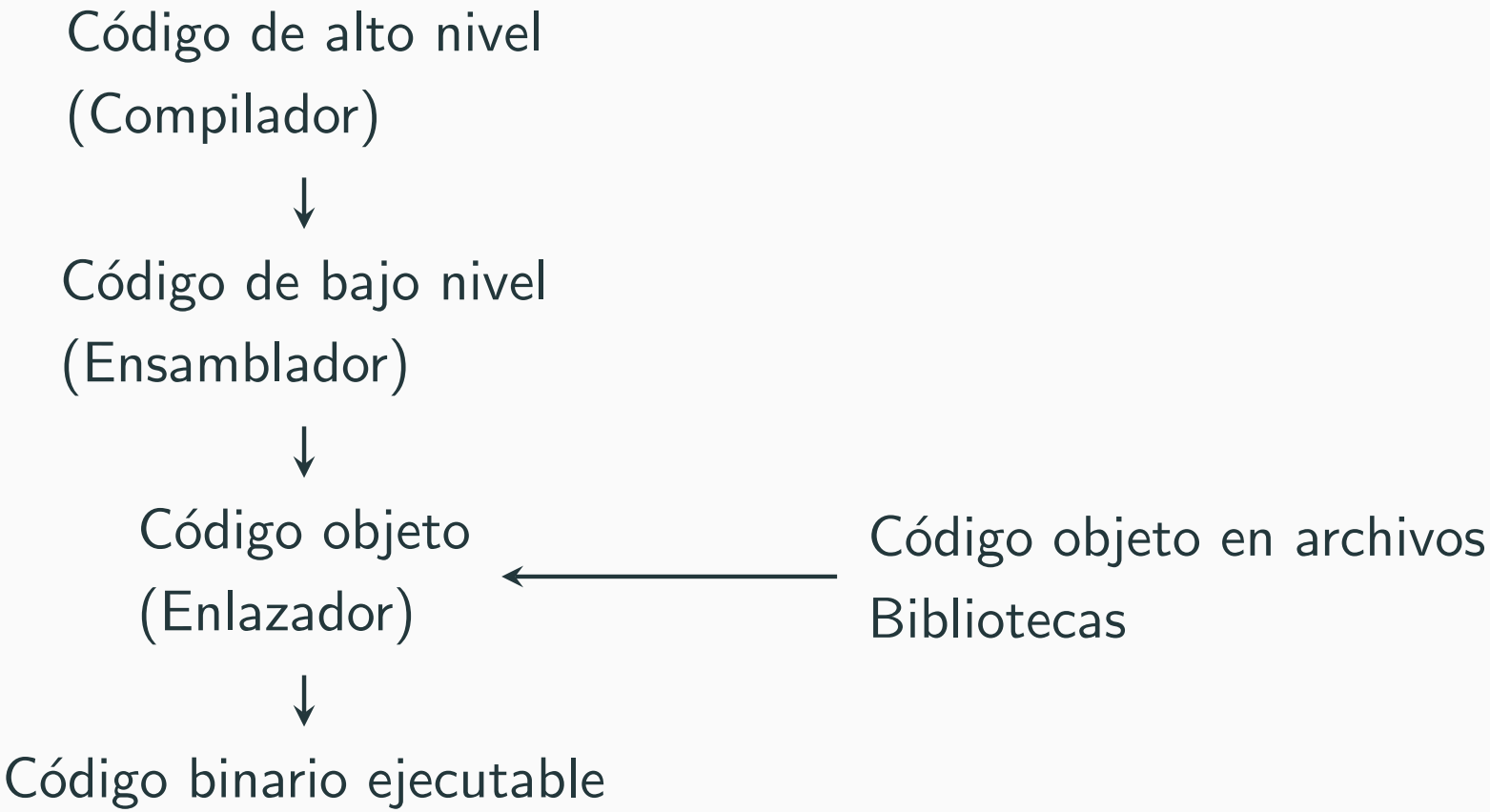


Compilación, ensamblado y ejecución

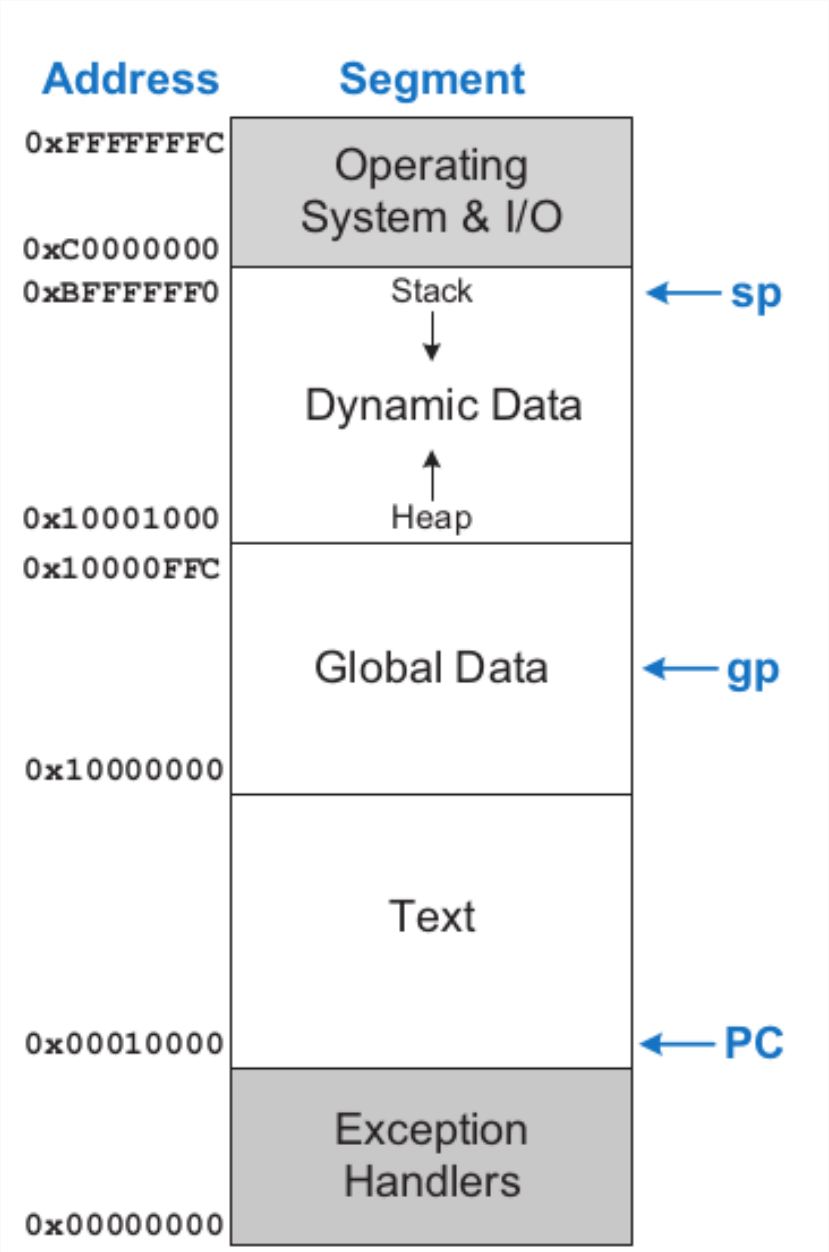
Cadena de compilación

Habíamos presentado anteriormente el esquema de traducciones que nos permite llegar de código de alto nivel a un formato binario que pueda cargarse en la memoria principal para poder ejecutar, vamos a repasarlo y a presentar el mapa de memoria.

Repaso de la cadena de compilación



El mapa de memoria



El mapa de memoria

El mapa de memoria divide a la memoria principal según su uso:

El mapa de memoria

El mapa de memoria divide a la memoria principal según su uso:

- La región más alta se reserva para comunicacíon de **entrada y salida**.

El mapa de memoria

El mapa de memoria divide a la memoria principal según su uso:

- La región más alta se reserva para comunicacíon de **entrada y salida**.
- Luego se encuentra la región de **datos dinámicos** donde en las direcciones altas vamos a encontrar la pila (`stack`)y en las direcciones bajas el `heap` que es la estructura que permite a un programa hacer un pedido explícito de memoria (`malloc`, `free`, sin usar el `stack`).

El mapa de memoria

El mapa de memoria divide a la memoria principal según su uso:

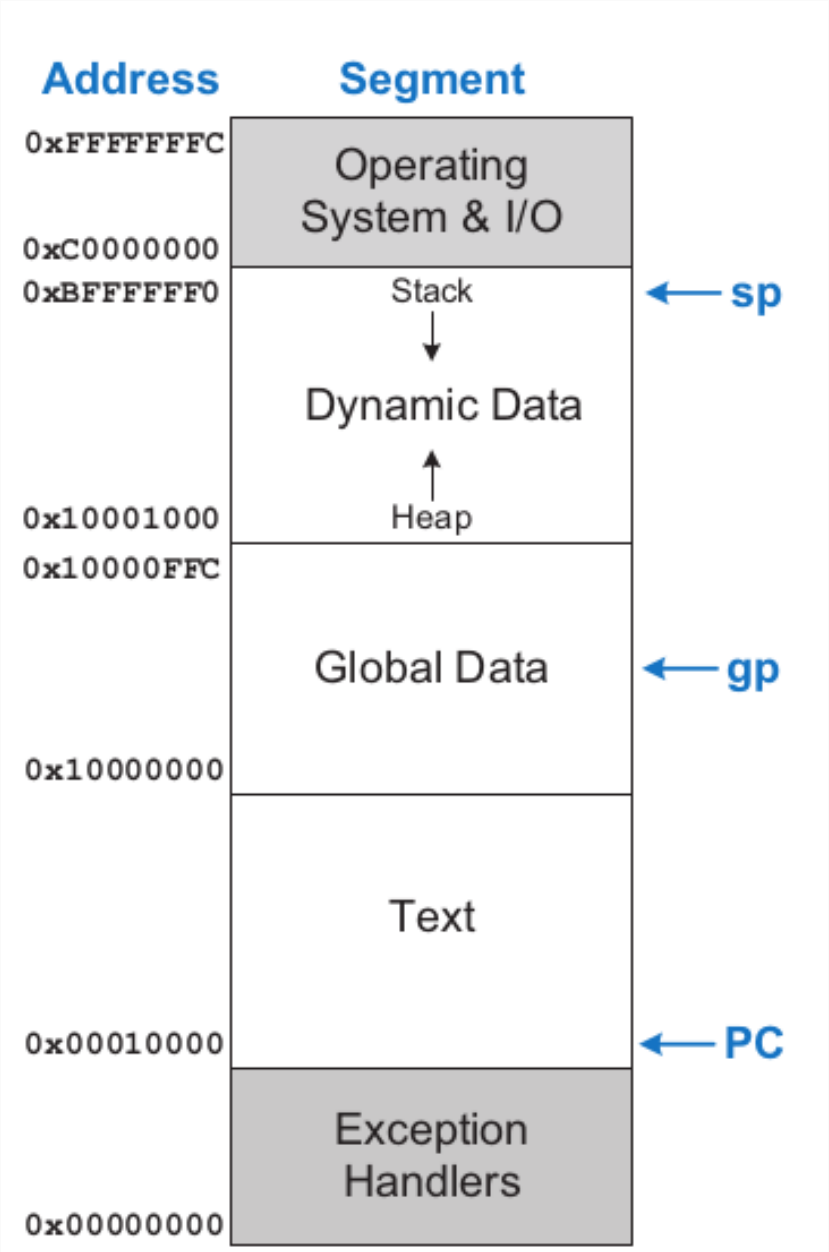
- La región más alta se reserva para comunicacíon de **entrada y salida**.
- Luego se encuentra la región de **datos dinámicos** donde en las direcciones altas vamos a encontrar la pila (`stack`)y en las direcciones bajas el `heap` que es la estructura que permite a un programa hacer un pedido explícito de memoria (`malloc`, `free`, sin usar el `stack`).
- Luego se encuentran los **datos globales**(`.global`), donde se almacenan variables y constantes globales.

El mapa de memoria

El mapa de memoria divide a la memoria principal según su uso:

- La región más alta se reserva para comunicacíon de **entrada y salida**.
- Luego se encuentra la región de **datos dinámicos** donde en las direcciones altas vamos a encontrar la pila (`stack`)y en las direcciones bajas el `heap` que es la estructura que permite a un programa hacer un pedido explícito de memoria (`malloc`, `free`, sin usar el `stack`).
- Luego se encuentran los **datos globales**(`.global`), donde se almacenan variables y constantes globales.
- Y luego el **texto**(`.text`), que es donde se encuentra el contenido binario de nuestro programa.

El mapa de memoria



Directivas de ensamblado

Existen algunas directivas, que no son realmente instrucciones, sino indicaciones para que el progama ensamblador puede reservar memoria, definir constantes y ubicar el programa y los datos según las secciones definidas en el mapa de memoria, a continuación presentamos algunas.

Directivas de ensamblado

Assembler Directive	Description
`.text`	Text section
`.data`	Global data section
`.bss`	Global data initialized to 0
`.section .foo`	Section named `.foo`
`.align N`	Align next data/instruction on 2^N-byte boundary
`.balign N`	Align next data/instruction on N-byte boundary
`.globl sym`	Label `sym` is global
`.string "str"`	Store string "`str`" in memory
`.word w1, w2, ..., wN`	Store N 32-bit values in successive memory words
`.byte b1, b2, ..., bN`	Store N 8-bit values in successive memory bytes
`.space N`	Reserve N bytes to store variable
`.equ name, constant`	Define symbol `name` with value `constant`
`.end`	End of assembly code

Sección de datos y de texto

Veamos por ejemplo cómo se inicializan los datos en la sección de `.data` que va a ubicar la información en lo que el mapa se muestra como `Global Data`, arriba del código (`.text`), mostramos:

Sección de datos y de texto

Veamos por ejemplo cómo se inicializan los datos en la sección de `.data` que va a ubicar la información en lo que el mapa se muestra como `Global Data`, arriba del código (`.text`), mostramos:

- Una constante `largo` de 32 bits (una palabra o `word`).

Sección de datos y de texto

Veamos por ejemplo cómo se inicializan los datos en la sección de `.data` que va a ubicar la información en lo que el mapa se muestra como `Global Data`, arriba del código (`.text`), mostramos:

- Una constante `largo` de 32 bits (una palabra o `word`).
- Una constante `caracter` de 8 bits (un `byte`).

Sección de datos y de texto

Veamos por ejemplo cómo se inicializan los datos en la sección de `.data` que va a ubicar la información en lo que el mapa se muestra como `Global Data`, arriba del código (`.text`), mostramos:

- Una constante `largo` de 32 bits (una palabra o `word`).
- Una constante `caracter` de 8 bits (un `byte`).
- Un arreglo `arreglo` de palabras de 32 bits.

Inicializando datos


```
1       .section .data
2       # A partir de este punto comienzan los datos
3       largo: .word 0x4
4       caracter: .byte 10
5       arreglo: .word 0xc, 0x34d, 0x1, 0x0
6       .setion .text
7       # A partir de este punto comienzan las
            instrucciones
```

Al igual que con los saltos en el programa, las etiquetas que declaran constantes van a indicar la posición de memoria desde donde debe cargarse el dato.

Revisión del programa de ejemplo

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

¿Qué podemos entender de la traducción que presentamos antes?

Programa de ejemplo en ASM (RISC V)


```
1  .section .text
2  .global sumar_arreglo
3  sumar_arreglo:
4  # a0 = int a[], a1 = int largo, t0 = acumulador, t1
       = i
5      li      t0, 0           # acumulador = 0
6      li      t1, 0           # i = 0
7  ciclo:  # Comienzo de ciclo
8      bge     t1, a1, fin     # Si i >= largo, sale del ciclo
9      slli    t2, t1, 2       # Multiplica i por 4 (1 << 2 = 4)
10     add     t2, a0, t2      # Actualiza la dir. de memoria
11     lw      t2, 0(t2)       # De-referencia la dir,
12     add     t0, t0, t2      # Agrega el valor al acumulador
13     addi    t1, t1, 1       # Incrementa el iterador
14     j       ciclo           # Vuelve a comenzar el ciclo
15 fin:
16     mv      a0, t0          # Mueve t0 (acumulador) a a0
17     ret                     # Devuelve valor por a0
```

Cierre

Clase de hoy

Hoy vimos:

- Definición de **arquitecturas**.

Clase de hoy

Hoy vimos:

- Definición de **arquitecturas**.
- El lenguaje ensamblador de `RISC V`.

Clase de hoy

Hoy vimos:

- Definición de **arquitecturas**.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.

Clase de hoy

Hoy vimos:

- Definición de **arquitecturas**.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.
- Compilación, ensamblado, vinculación y ejecución.

Clase de hoy

Hoy vimos:

- Definición de **arquitecturas**.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.
- Compilación, ensamblado, vinculación y ejecución.
- Programa de ejemplo.

Fin