DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Sistemas Digitales

Arquitectura 2/2

Primer Cuatrimestre 2025

Sistemas Digitales
DC - UBA

Introducción

Clase anterior

En la clase anterior vimos:

- Definición de arquitecturas.

Clase anterior

En la clase anterior vimos:

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

Clase anterior

En la clase anterior vimos:

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

Clase de hoy

Hoy vamos a ver:

- Acceso a memoria y estructuras.

Clase de hoy

Hoy vamos a ver:

- Acceso a memoria y estructuras.
- Interfaz binaria de aplicación.

Clase de hoy

Hoy vamos a ver:

- Acceso a memoria y estructuras.
- Interfaz binaria de aplicación.
- Uso de la pila.

¿Qué constituye una arquitectura?

¿Qué constituye una arquitectura?

- El conjunto de instrucciones.

¿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.

Repaso

¿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?

Clase anterior

Volvamos a nuestro programa de referencia y al ejemplo de control de ejecución.

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 }
```

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:`

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:`

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

Manejo de estructuras

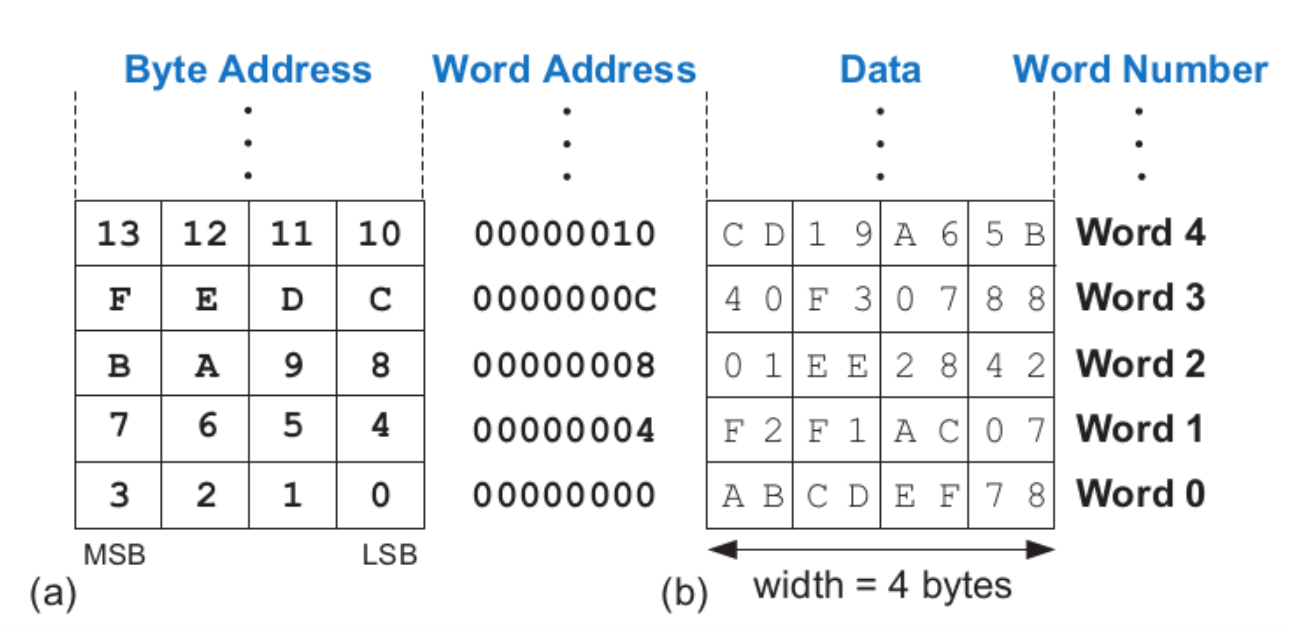
Direcciones de memoria

Recordemos cómo se realiza el acceso a datos en memoria.

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)



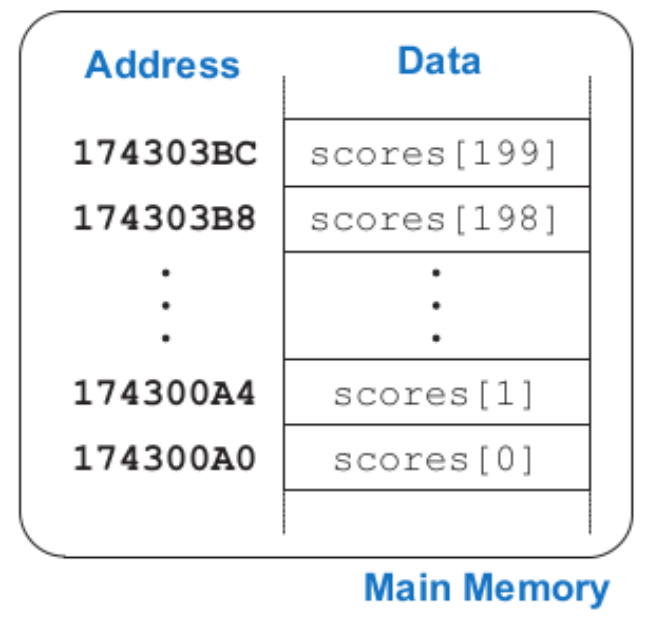
A la izquierda (a), vemos los índices de memoria (`byte address`) representados de derecha a izquierda, donde a la derecha vemos el byte menos significativo (LSB) y a la derecha el byte más significativo de la palabra (MSB). La dirección de palabra (word address) corresponde al índice del byte menos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

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

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

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

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

```
#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:
```

Interfaz binaria de aplicación

Llamadas a función


```
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;
}
```

¿Cómo escribimos funciones en `RISC V` pasando un número arbitrario de parámetros y devolviendo un parámetro de retorno?

Definiendo un contrato


```
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;
}
```

Recordemos que contamos con una memoria principal direccionable y un número acotado de registros y con esto vamos a tener que definir un contrato que indique de qué forma se realizan las llamadas a función.

Interfaz binaria de aplicación

A este contrato que indica de qué forma vamos a realizar las llamadas a función para una arquitectura en particular lo llamamos interfaz binaria de aplicación. Define un conjunto de reglas que tanto quienes programan en ensamblador `RISC V` como el programa de compilación de un lenguaje de alto nivel a ensamblador `RISC V` deben respetar para poder interactuar con otros programas, llamadas a sistema y bibliotecas compartidas.

Ejemplo con argumentos

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

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
```

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

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

```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;
}
```

```
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
```

Uso de la pila y el stack pointer

¿Qué es la pila?

La pila es:

¿Qué es la pila?

La pila es:

- Una región de la memoria definida entre una dirección de memoria alta y la dirección indicada en el registro stack pointer (sp).

¿Qué es la pila?

La pila es:

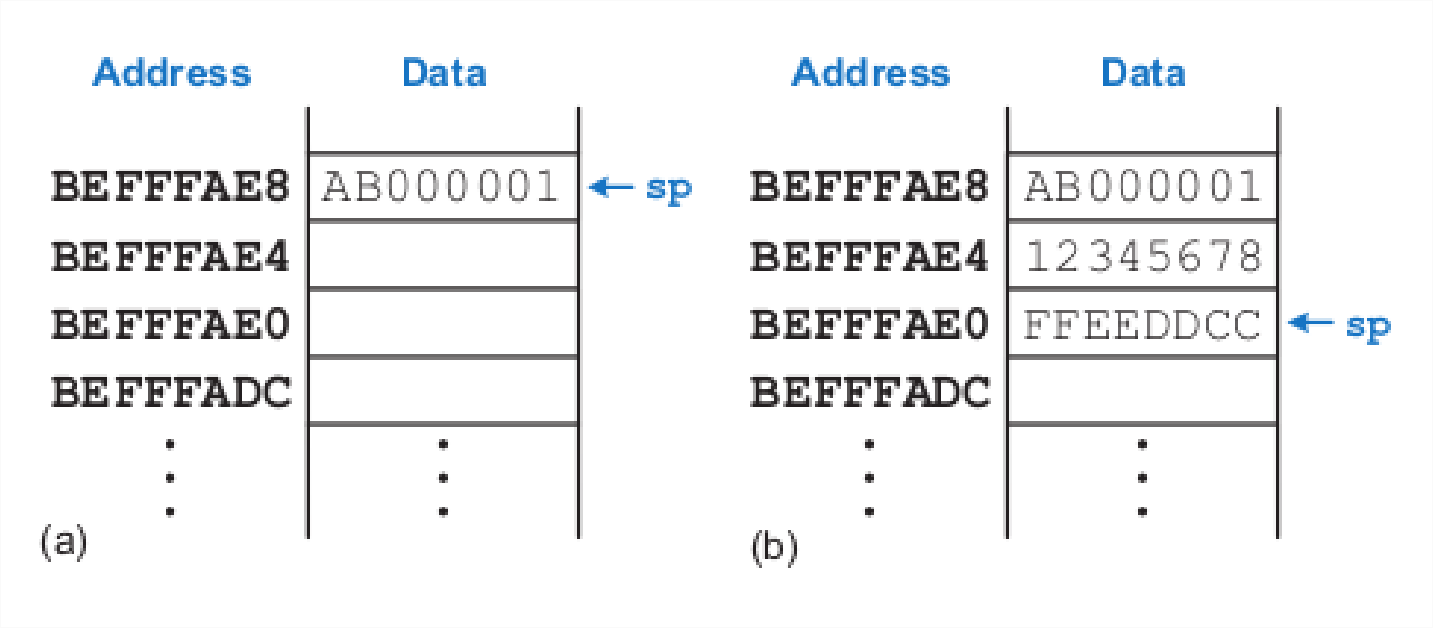
- Una región de la memoria definida entre una dirección de memoria alta y la dirección indicada en el registro stack pointer (sp).
- Un mecanismo para almacenar valores temporarios con una semántica de LIFO (el último elemento almacenado es el primero al que accedemos).

La pila (stack)

La semántica de uso es a través de operación de agregado (`push`) y retiro (`pop`) de un elemento siempre al 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 tope 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 (stack)

Parte de la convención de `RISC V` (interfaz binaria de aplicación) indica que el stack pointer debe siempre estar alineado a 16 bytes, esto significa que su valor debe siempre cumplir con la congruencia

$$sp \,\%16 == 0$$

.

La pila (stack)

Veamos un breve ejemplo del uso de la pila y la alineación del `stack pointer`.

```
# apilando (push) en una llamada a funcion
foo: addi sp, sp, -16 # restamos 16 aunque
    precisemos 8 bytes
sw a0, 4(sp) #guarda a0
sw ra, 0(sp) #guarda ra
# cuerpo de la funcion
# desapilando (pop)
lw a0, 4(sp) #restaura a0
lw ra, 0(sp) #restaura ra
addi sp, sp, 16 # restaura el valor de stack
    pointer
```

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 para llamar funciones

Vamos a presentar una serie de reglas que deberían asegurar que cada función llamadora entrega y cada función llamada recibe a los elementos de memoria del procesador (registros, memoria general y pila) en un estado conocido.

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:

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

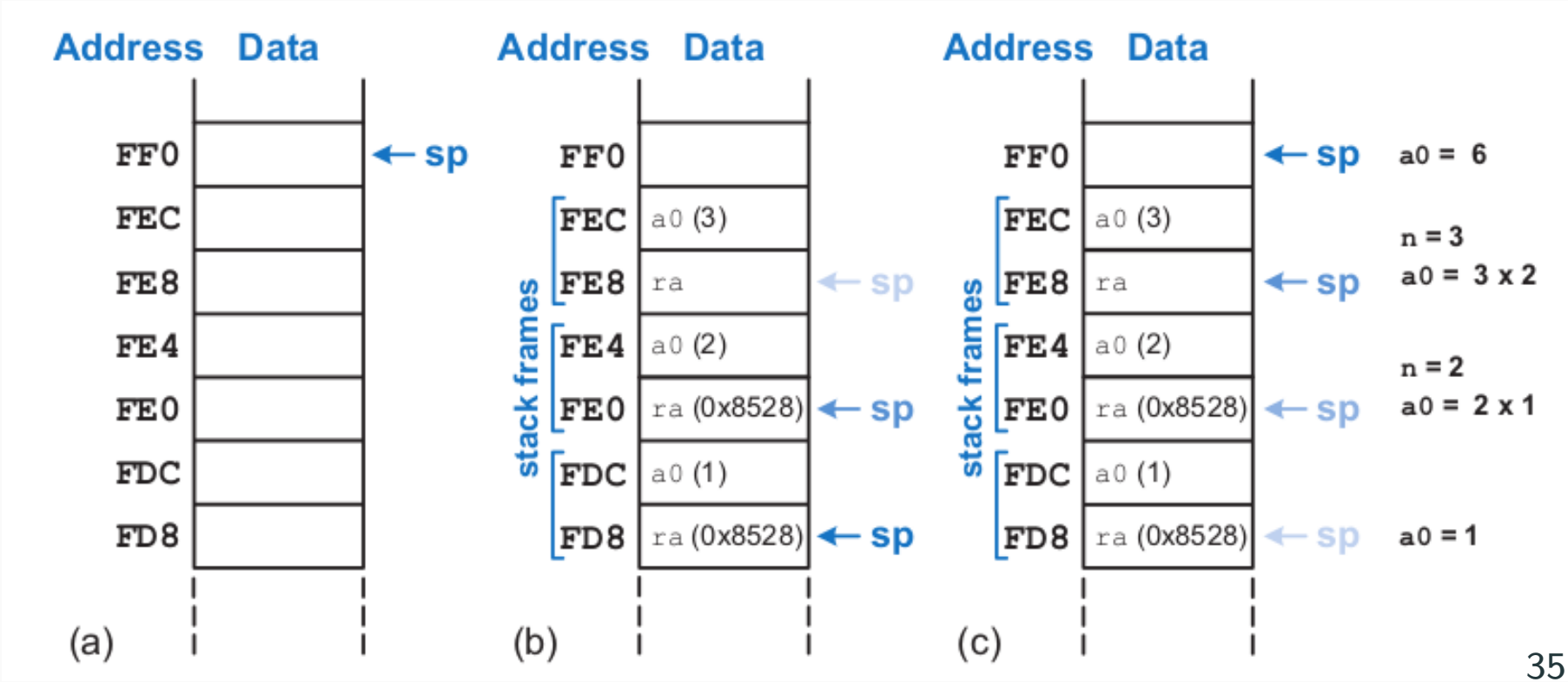
DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

`int factorial(int n){`
`  if(n <= 1){`
`  return 1;`
`  }else{`
`  return`
`  (n*factorial(n-1));`
`  }`
`}`
	`factorial:addi sp, sp, -16`
	`  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, 16`
	`  jr ra`
	`else: addi a0, a0, -1`
	`  jal factorial`
	`  lw t1, 4(sp)`
	`  lw ra, 0(sp)`
	`  addi sp, sp, 16`
	`  mul a0, t1, a0`
	`  jr ra`

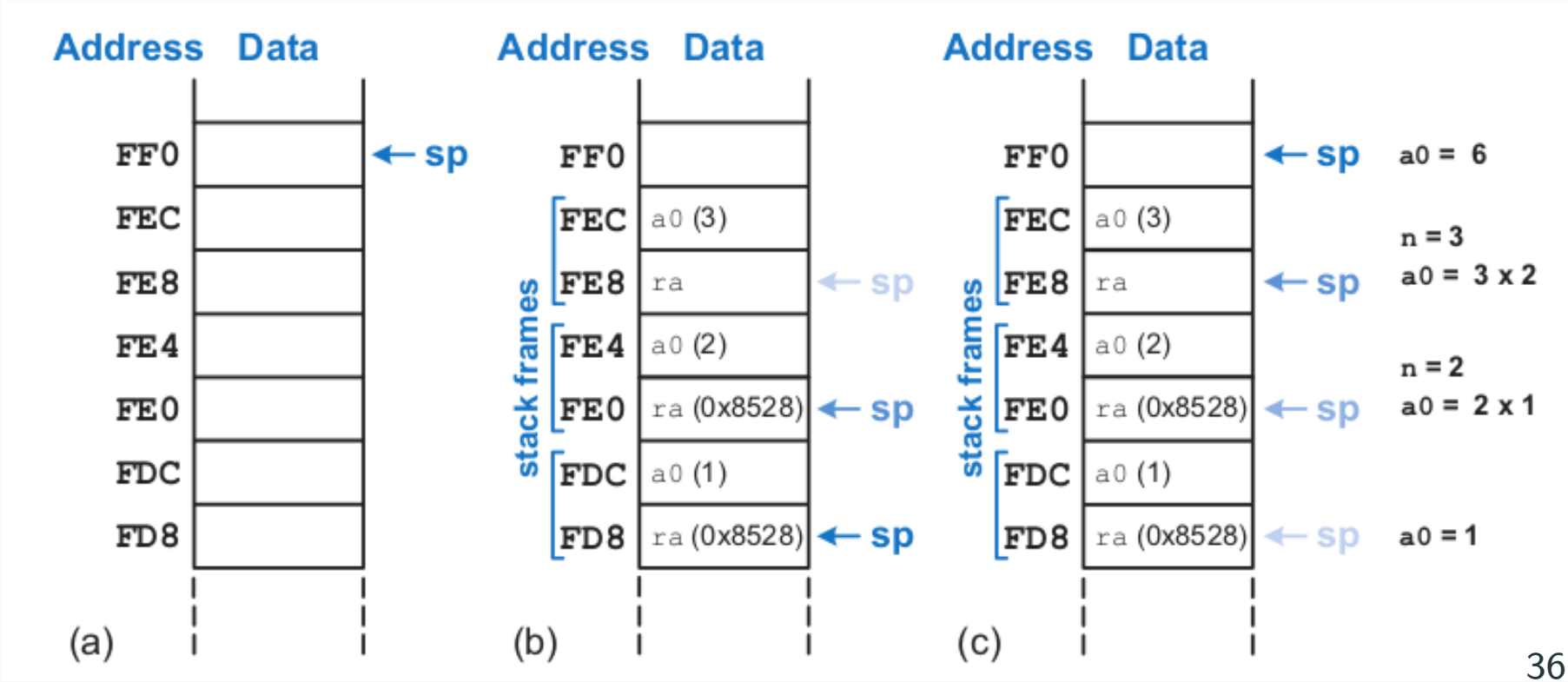
Stack pointer (sp)

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.



Stack pointer (sp)

Aquí la columna a muestra las posiciones altas de la memoria antes de la primer llamada, la columna b muestra el estado luego de tres llamadas recursivas y la columna c indica cómo se actualizan los valores de a0 al ir regresando de cada llamada (`jr ra`).



Pseudoinstrucciones

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`

Interfaz binaria de aplicación en la práctica

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
```

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:

- Acceso a memoria y estructuras.

Clase de hoy

Hoy vimos:

- Acceso a memoria y estructuras.
- Interfaz binaria de aplicación.

Clase de hoy

Hoy vimos:

- Acceso a memoria y estructuras.
- Interfaz binaria de aplicación.
- Uso de la pila.

Fin