

Universidad Tecnológica Nacional
Facultad Regional Tucumán
Ingeniería en Sistemas de Información
ARQUITECTURA DE COMPUTADORES

TRABAJO PRÁCTICO N° 5

Capítulo 4:

**La Arquitectura de Programación.
ARC, una computadora RISC.**

Capítulo 5:

**Los Lenguajes y la Máquina.
5.2: El Proceso de Ensamblado.**

Arquitectura RISC, explicación básica.

CISC y RISC

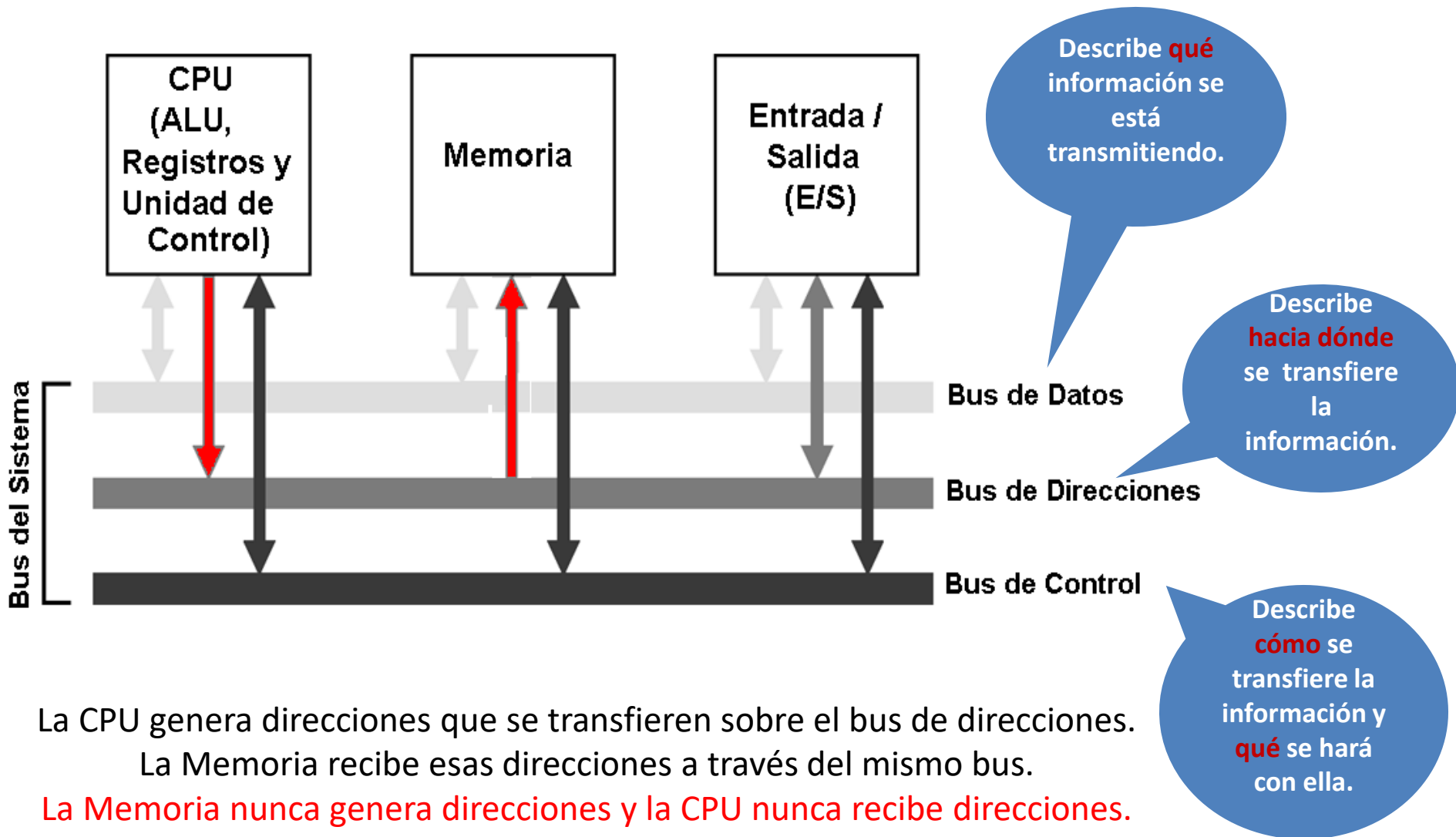
CISC	RISC
Instrucciones multiciclo	Instrucciones de único ciclo
Carga/almacenamiento incorporadas en otras instrucciones	Carga/almacenamiento son instrucciones separadas
Arquitectura memoria-memoria	Arquitectura registro-registro
Instrucciones largas, Código con menos líneas	Instrucciones cortas, Código con más líneas
Utiliza memoria de microprograma	Implementa las instrucciones directamente en hardware
Se enfatiza la versatilidad del repertorio de instrucciones	Se añaden instrucciones nuevas sólo si son de uso frecuente y no reducen el rendimiento de las más importantes
Reduce la dificultad de implementar compiladores	Compiladores complejos
	Elimina microcódigo y la decodificación de instrucciones complejas

El Modelo del Bus del Sistema

**¿Recuerda los componentes del
Modelo del Bus de Sistema?**

El Modelo de Bus del Sistema

Es un refinamiento del modelo de von Neumann.





escribe un

C++

```
.begin
.org 2048
progl: ld    [x], %r1
      ld    [y], %r2
      addcc %r1, %r2, %r3
      st    %r3, [z]
      jmp1  %r15 + 4, %r0
x:     15
y:     9
z:     0
.end
```

Programa en Lenguaje de Alto Nivel

Momento de
Compilado

Programa en Lenguaje Simbólico (assembler)

Momento de
Ensamblado

Programa en Lenguaje de Máquina (código máquina)

Momento de Enlace

Programa Ejecutable

Almacenamiento

Disco Duro



Momento de la
Carga



Momento de la
Ejecución



Salidas del
Programa



Los resultados
obtenidos en la
CPU vuelven a
Memoria para su
almacenamiento

**DEPENDENCIA
DE LA CPU**

0 y 1 es lo
único que
entiende la
computadora

Permite
relacionarse
con otros
programas en
código
máquina.

El Sistema Operativo
carga el programa en
lenguaje máquina desde
el disco a memoria

Desde memoria se
carga cada instrucción
a la CPU una por vez,
junto con los datos
necesarios para
ejecutar la instrucción

Algunas Definiciones (Repaso)

- Instrucción: orden a ejecutar por la CPU.
- Lenguaje ensamblador, assembler o simbólico: se escribe con letras, símbolos, dígitos.
- Lenguaje de máquina o Código Objeto: es lo que entiende la máquina, 0 y 1.

```
.begin
.org 2048
progl: ld    [x], %r1
      ld    [y], %r2
      addcc %r1, %r2, %r3
      st    %r3, [z]
      jmp1  %r15 + 4, %r0
x:     15
y:     9
z:     0
.end
```

1100	0010	0000	0000	0010	1000	0001	0100
1100	0100	0000	0000	0010	1000	0001	1000
1000	0110	1000	0000	0100	0000	0000	0010
1100	0110	0010	0000	0010	1000	0001	1100
1000	0001	1100	0011	1110	0000	0000	0100
0000	0000	0000	0000	0000	0000	0000	1111
0000	0000	0000	0000	0000	0000	0000	1001
0000	0000	0000	0000	0000	0000	0000	0000

Cada instrucción escrita en assembler le corresponde un único código máquina.

Algunas definiciones (Repaso)

- Dirección: es un puntero a una posición de memoria, la cual contiene un dato o una instrucción. Es el número que identifica en forma unívoca cada palabra.

Las direcciones comienzan en cero, y la última es una unidad menos que el tamaño de memoria.

EJEMPLO:

El ancho del bus de direcciones da el tamaño de memoria, un ancho de 32 bits surge una memoria que tiene $2^{32} = 4 \text{ GB}$, entonces:

Primera dirección = 0

Última dirección = $2^{32} - 1$

ARQUITECTURA

ARC

ARQUITECTURA ARC

- ARC (A RISC Computer): es un subconjunto del modelo de arquitectura basado en el procesador SPARC, desarrollado por Sun Microsystems.
- RISC: significa computadora con un conjunto reducido de instrucciones.

ARQUITECTURA ARC

Tipos de palabras:

- Palabra datos o instrucciones: tienen 32 bits cada una.
- Palabra de dirección de memoria: también tiene 32 bits, puede especificar una dirección de una memoria de $2^{32} = 4 \text{ GB}$.

Tamaños Comunes en los formatos de Palabras de Datos

Bit	0
Nibble (4 bits)	0110
Byte (8 bits)	10110000
16 bit (media palabra)	11001001 01000110
⇒ 32 bit (palabra)	10110100 00110101 10011001 01011000
64 bit (palabra doble)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101
128 bit (palabra cuádruple)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101 00001011 10100110 11110010 11100110 10100100 01000100 10100101 01010001

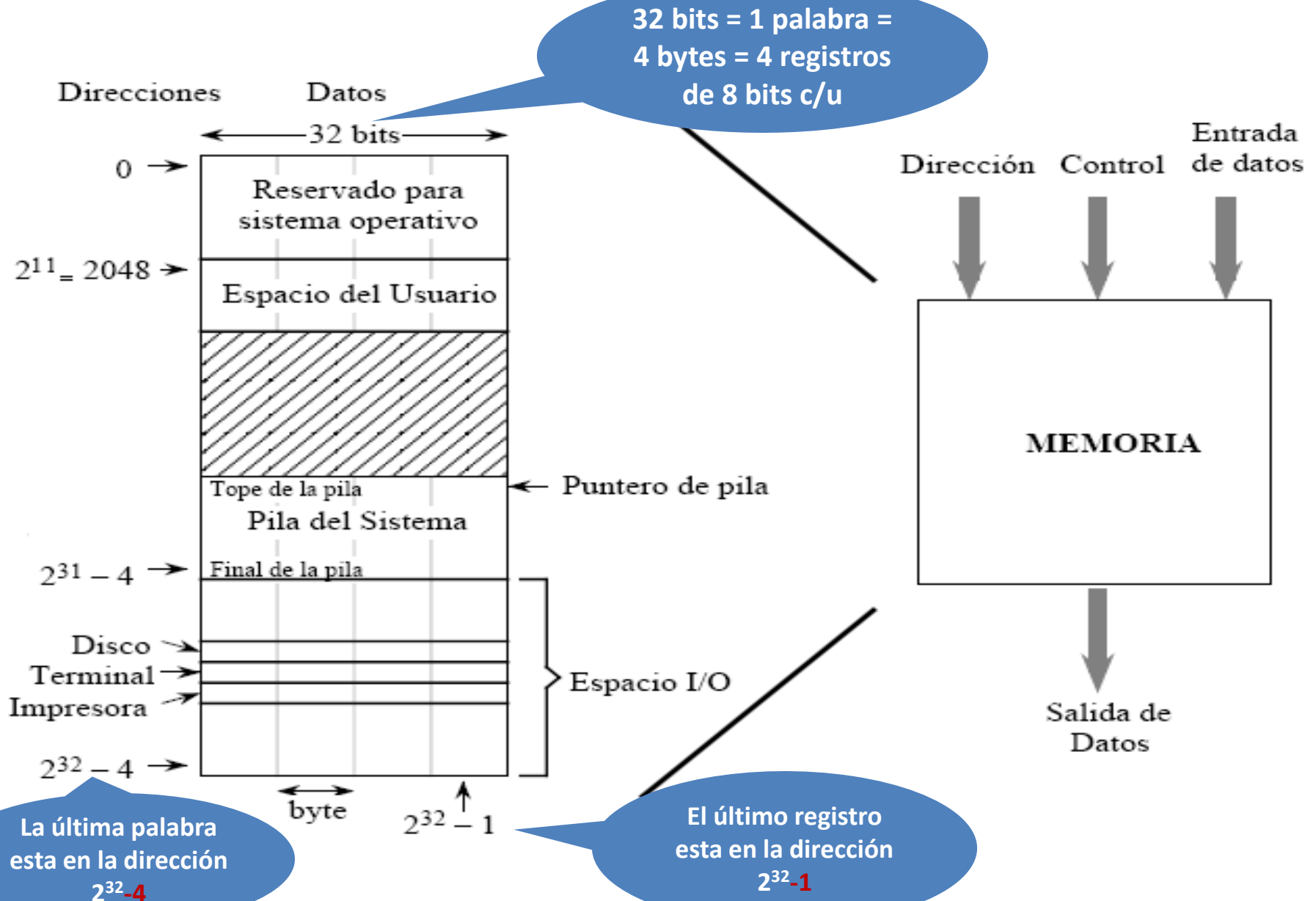
- El de 32 bits es el que se usa en la Arquitectura ARC.
- **Equivale a 4 bytes ($4 \times 8 = 32$), que son 4 registros.**
- El dato más pequeño que se puede hacer referencia en la memoria es el byte (8 bits).

Memoria en la Arquitectura ARC

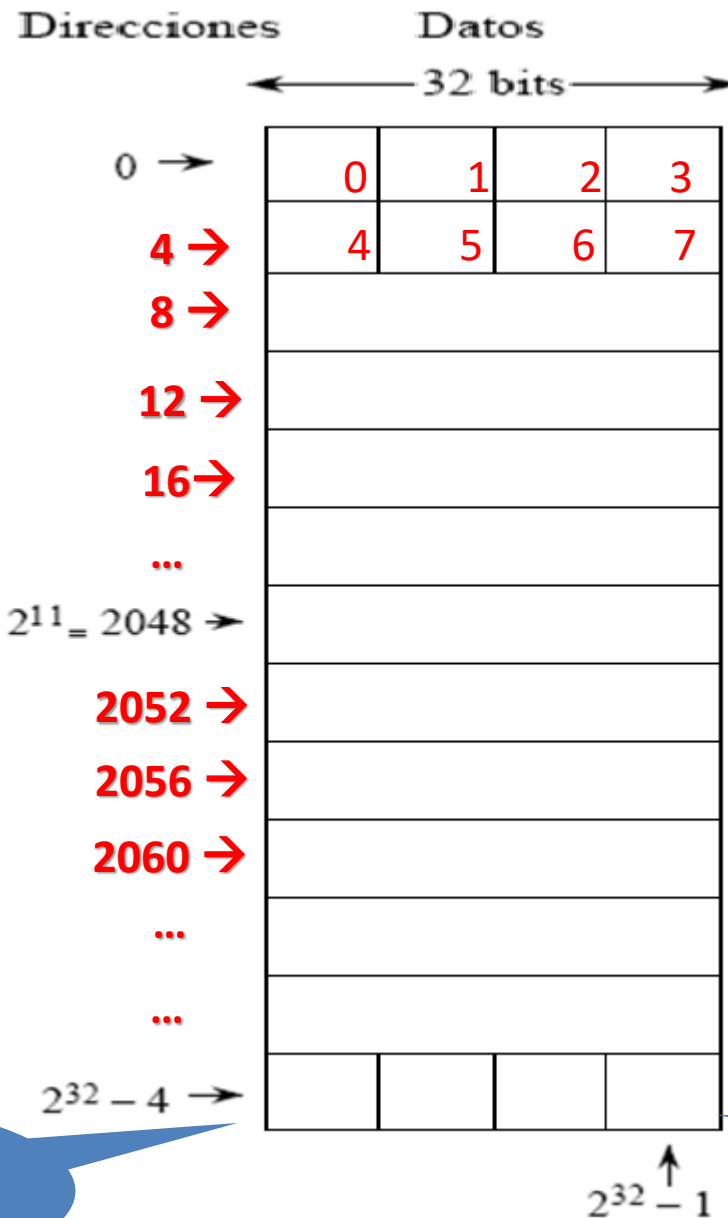
- La Memoria es de 4 GB (2^{32}).

Consiste en un conjunto de registros numerados (direccionados) en forma consecutiva, cada uno almacena normalmente un byte (8 bits).

Mapa de Memoria de la Arquitectura ARC



Mapa de Memoria de la Arquitectura ARC



32 bits = 1 palabra =
4 bytes = 4 registros
de 8 bits c/u

Cada registro tiene su dirección

**En la memoria de
ARC las direcciones
avanzan de a 4.**

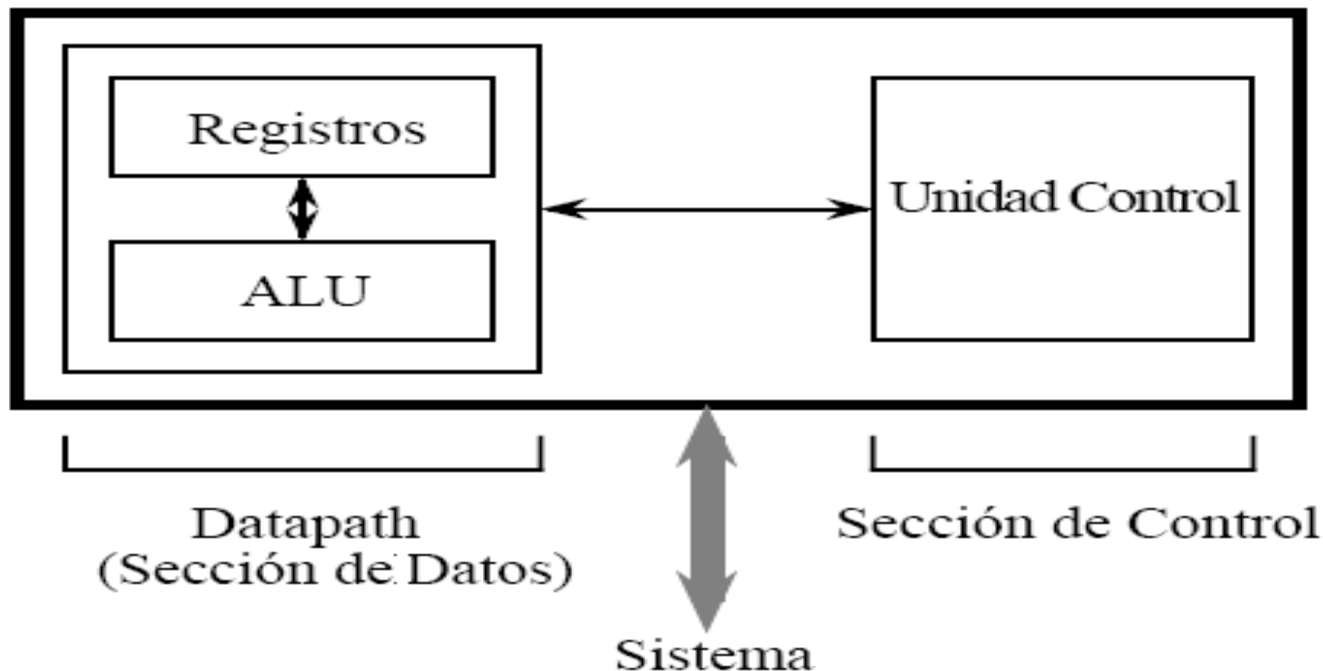
La última palabra
esta en la dirección
 $2^{32}-4$

El último registro
esta en la dirección
 $2^{32}-1$

■ Vista Abstracta de una CPU

La CPU consiste en:

- Una sección de datos, llamada también “trayecto de datos” o "datapath", que contiene registros y una ALU.
- Una sección de control, que interpreta las instrucciones y realiza las transferencias entre registros.



- Los pasos que la Unidad de Control lleva a cabo en la ejecución de un programa son:
 1. Buscar de la memoria la siguiente instrucción a ejecutar.
 2. Decodificar el código de operación.
 3. Buscar los operandos de la memoria principal, si los hubiera.
 4. Ejecutar la instrucción y almacenar los resultados.
 5. Volver al paso 1.

**Esto se conoce como:
Ciclo de Búsqueda y Ejecución.**

Trayecto de Datos (datapath)

Esta formado por:

- Conjunto de registros: son memorias pequeñas y rápidas, separadas de la memoria principal, **están en la CPU**. Se las usa para el almacenamiento temporal durante las operaciones de cálculo. Cada registro posee una dirección ordenada secuencialmente a partir de cero. Éstas direcciones son más chicas que las de la memoria principal.

Por ejemplo:

Un conjunto de registros que contenga 32 registros usará una palabra de dirección de solo 5 bits ($2^5=32$).

00000 = registro 0

a

11111 = registro 31

- ALU: implementa una variedad de operaciones, de uno y dos operandos.

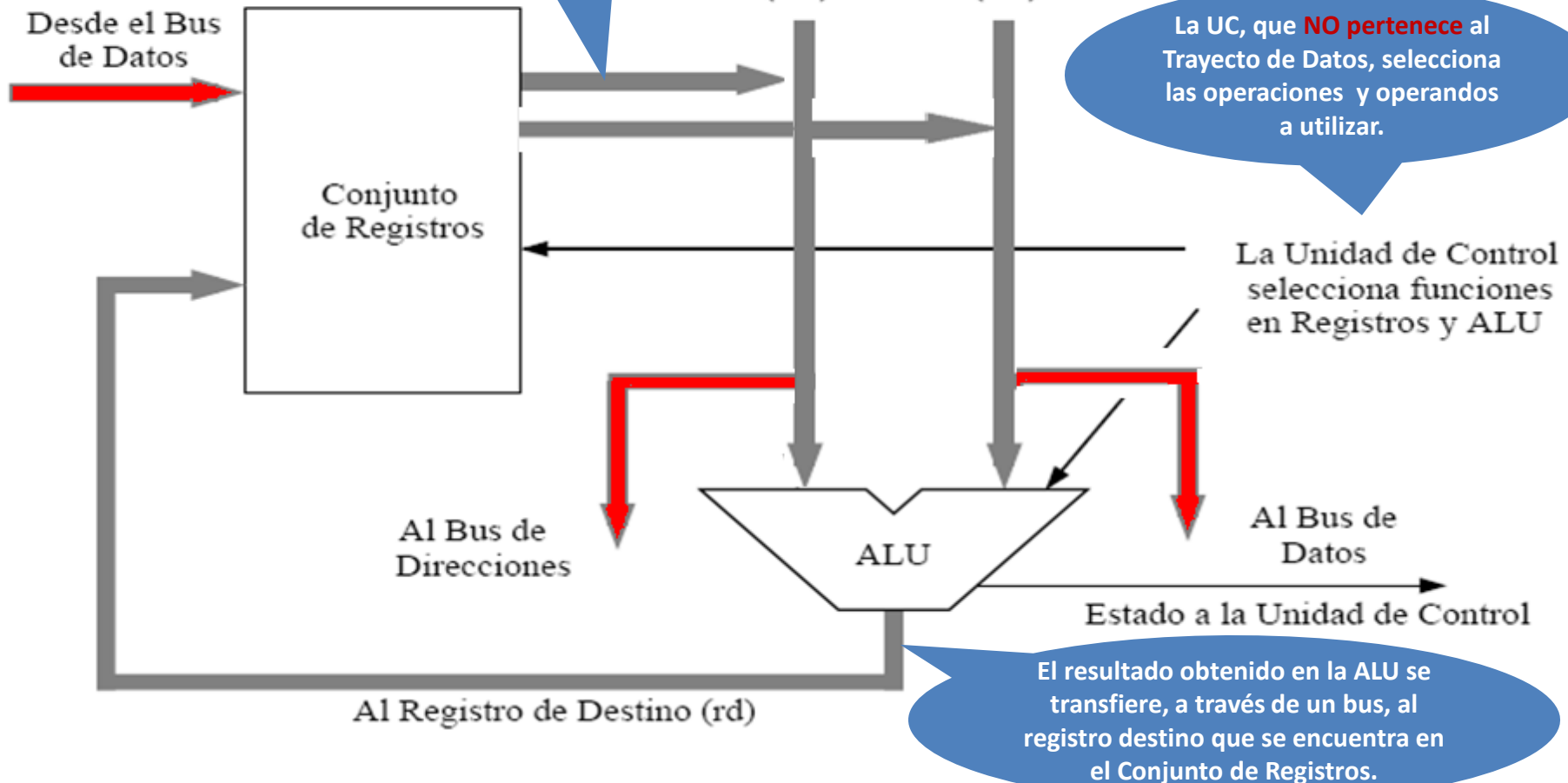
Por ejemplo:

Suma aritmética, Operaciones Lógicas (And, Or, Not).

■ Trayecto de Datos (datapath)

➡ Buses del Sistema, que conectan el Trayecto de Datos con Memoria y E/S.

➡ Buses adicionales, internos al Trayecto de Datos, que conectan el Conjunto de Registros con la ALU.



Registros de la CPU de la Máquina ARC

- Cada registro tiene 32 bits.
- Los registros se especifican con el símbolo %.
- Hay 32 registros visibles para el usuario, del 0 al 31, de uso general, salvo los de usos específicos:
 - El registro 0 (%r0) siempre contendrá el valor cero.
 - El registro 14 (%r14) es el puntero de pila.
 - El registro 15 (%r15) es el registro de enlace.

Registros ARC de uso general

Register 00	%r0 [= 0]	Register 11	%r11	Register 22	%r22
Register 01	%r1	Register 12	%r12	Register 23	%r23
Register 02	%r2	Register 13	%r13	Register 24	%r24
Register 03	%r3	Register 14	%r14 [%sp]	Register 25	%r25
Register 04	%r4	Register 15	%r15 [link]	Register 26	%r26
Register 05	%r5	Register 16	%r16	Register 27	%r27
Register 06	%r6	Register 17	%r17	Register 28	%r28
Register 07	%r7	Register 18	%r18	Register 29	%r29
Register 08	%r8	Register 19	%r19	Register 30	%r30
Register 09	%r9	Register 20	%r20	Register 31	%r31
Register 10	%r10	Register 21	%r21		

← 32 bits →

Registros ARC

- El registro PC es el registro número 32.
- Existen 4 registros temporales.
- El registro IR es el número 37.

En total hay 38 registros en el Conjunto de Registros del Trayecto de Datos.

Registros ARC

- El registro **PSR** (Registro de Estado del Procesador) *no se encuentra en el Trayecto de Datos, está en la Sección de Control*. Contiene información acerca del *estado del procesador*. Posee códigos de condición que especifican si una operación aritmética dio :
 - Resultado negativo (n). n = 1 el resultado fue negativo
n = 0 el resultado fue positivo
 - Resultado cero (z).
 - Si produjo un desborde (v), o sea que el resultado de la operación aritmética es demasiado grande para la ALU.
 - Si produjo un arrastre (acarreo) a la salida de la ALU (c).






Los números negativos se los expresa en complemento a 2.

Conjunto de Instrucciones

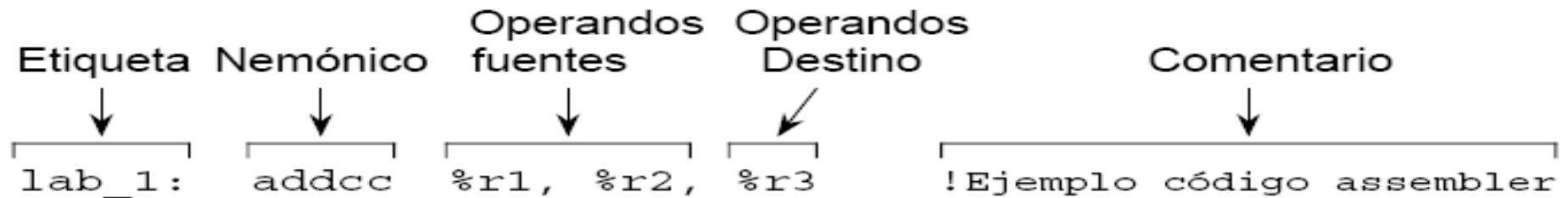
- Es la colección de instrucciones que un procesador puede ejecutar.
- Difiere de un procesador a otro, en el tamaño de las instrucciones, el tipo de operaciones que permiten, el tipo de operandos que puede ejecutar y los resultados que pueden entregar.
- El tamaño de una instrucción en ARC es de 32 bits, o sea una palabra.
- En ARC solo dos instrucciones pueden acceder a Memoria, ld (load=carga) y st (store=descarga).

El Conjunto de Instrucciones de ARC

	Nemonico	Significado
Memoria	ld	Cargar registro desde la memoria
	st	Almacenar un registro en la memoria
Lógicas	sethi	Cargar los 22 bits mas significativos de un registro
	andcc	Operación lógica AND bit a Bit 
	orcc	Operación lógica OR bit a bit 
	orncc	Operación lógica NOR bit a bit 
Aritmeticas	srl	Desplazar a derecha (lógico) agrega ceros a la izq.
	addcc	Sumar 
Control	call	Salto ó Llamado a subrutina
	jmp1	Salto y Enlace (retorno de subrutina)
	be	Bifurcación o Salto por igual
	bneg	Bifurcación o Salto por negativo
	bcs	Bifurcación o Salto por acarreo
	bvs	Bifurcación o Salto por desborde u "overflow"
	ba	Bifurcación o Salto incondicional

Las instrucciones aritméticas y lógicas terminadas con el sufijo “**cc**” especifican que luego de realizadas las operaciones se deben actualizar los códigos de condición del registro **PSR**.

Formato de Lenguaje Ensamblador ARC



- El lenguaje hace distinción entre mayúsculas y minúsculas.
- Los campos de etiqueta y comentario son optativos.
- El campo etiqueta usa caracteres alfabéticos, numéricos (siempre y cuando no sea el primer dígito), los símbolos guión bajo (_), signo monetario (\$), punto (.) y los **dos puntos (:)** que indica el **final de la etiqueta**.
- El campo comentario va precedido del símbolo **!**
- Los operandos se separan con comas (,) y su uso dependerá de cada instrucción.
- Los operandos origen 1 y destino siempre deben ser un registro. El operando origen 2 puede ser una constante.

Formatos de las instrucciones ARC en Lenguaje Máquina (código objeto)

- El formato de instrucción definirá como el programa ensamblador (que traduce programas en lenguaje assembler a códigos binarios), distribuye los diferentes campos de una instrucción y la forma en que los interpreta la Unidad de Control.
- Cada instrucción tiene 32 bits.
- Los dos bits más significativos forman el campo **OP**, que corresponde al código de operación, a partir de éste se identificará el formato.

■ Formatos de las instrucciones ARC

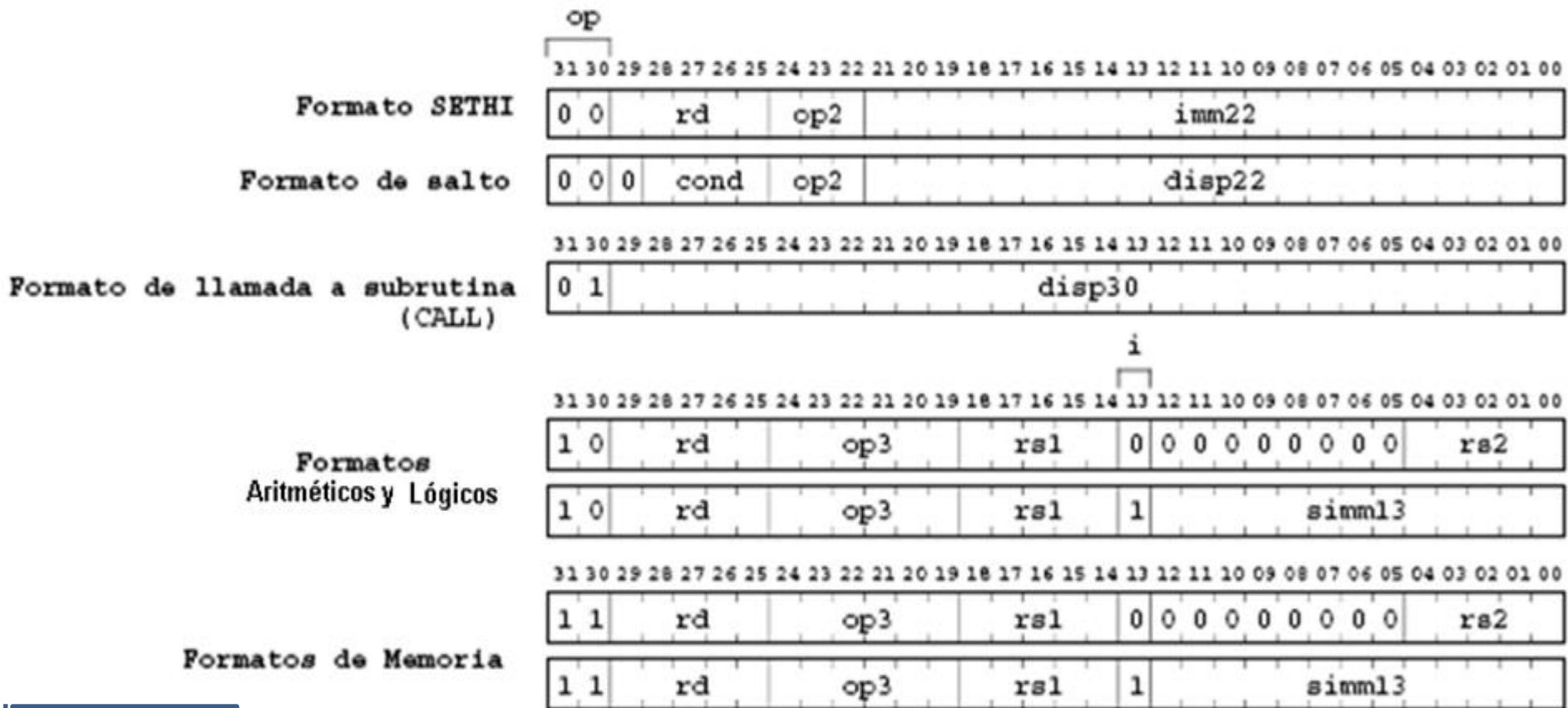


Figura 4.10,
página 114,
capítulo 4
del libro del
autor
Murdocca.

op	Format
00	SETHI/salto
01	CALL
10	Aritm y Lógicos
11	Memoria

op2	Inst.
010	branch
100	sethi

op3 (op=10)	
010000	addec
010001	andec
010010	orcc
010110	orncc
100110	srl
111000	jmp1

op3 (op=11)	
000000	ld
000100	st

cond	salto
0001	be
0101	bcs
0110	bneg
0111	bvs
1000	ba

Ejemplos de Instrucciones en lenguaje assembler y código objeto

INSTRUCCIONES ARITMÉTICAS Y LÓGICAS

Las instrucciones que terminan en “**cc**” modifican los códigos de condición del registro **PSR**.

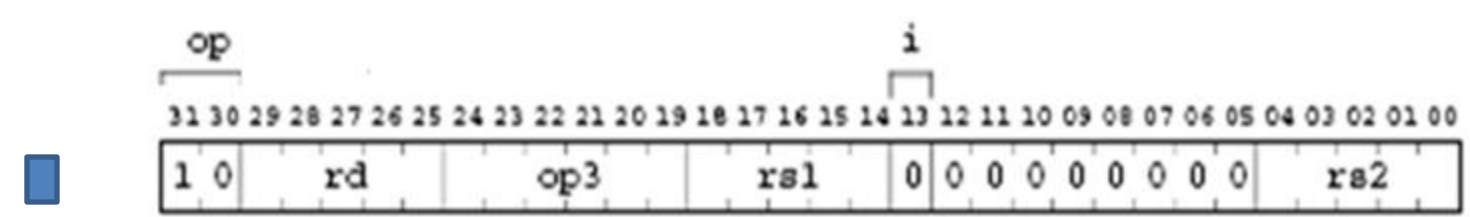
• Instrucciones aritméticas:

Suma el contenido del registro 1 con el contenido del registro 2 , el resultado lo almacena en el registro 3.

Lenguaje assembler:

```
lab_1:  addcc  %r1, %r2, %r3      ! %r3 ← %r1 + %r2
```

Formato de la instrucción:



Código objeto:

Op (2 bits)	rd Reg. destino (5 bits)	op3 (6 bits)	rs1 Reg. origen 1 (5 bits)	i	rs2 Reg. origen 2 (5 bits)
10	00011	010000	00000	1	000010
Arit.	%r3	addcc	%r1		%r2

En los campos rd, rs1 y rs2 van los números de los registros escritos con 5 dígitos binarios.

Identifica la instrucción, según la tabla op3 (con op = 10)

NOTA: Las etiquetas y comentarios no se traducen a código objeto.

• Instrucciones aritméticas:

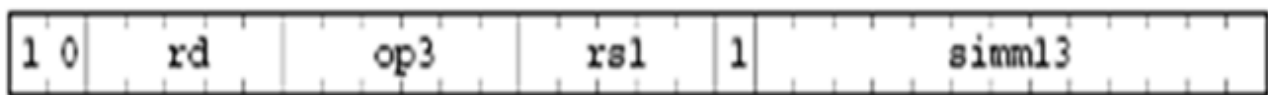
El valor de la constante esta en decimal, 12₍₁₀₎. Si se lo quisiera expresar en hexadecimal se debe poner adelante "0x" o debe terminar en "H".

Al contenido del registro 1 le suma el valor 12, el resultado lo almacena en el registro 3.

Lenguaje assembler:

```
lab_2:    addcc %r1, 12, %r3    ! %r3 ← %r1 + Constant
```

Formato de la instrucción:



Código objeto:

Op (2 bits)	rd Reg. destino (5 bits)	op3 (6 bits)	rs1 Reg. origen 1 (5 bits)	i	Simm13 (13 bits)
1 0	0 0 0 1 1	0 1 0 0 0 0	0 0 0 0 1	1	0 0 0 0 0 0 0 0 0 0 1 1 0 0
Arit.	%r3	addcc	%r1		Constante = 12

Cuando el segundo registro origen es una constante entonces **i = 1**, y en el campo simm13 se escribe el valor de la constante usando 13 dígitos binarios.

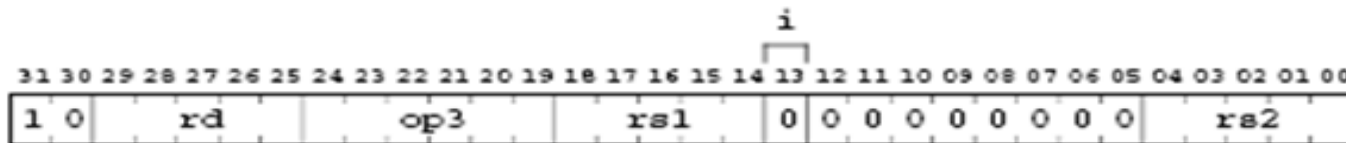
- Instrucciones lógicas:

Realiza el AND, bit por bit, entre los contenido de los %r1 y %r2 almacena el resultado en el %r3.

Lenguaje assembler:

```
lab_6: andcc %r1 , %r2 , % r3
```

Formato de la instrucción:



Código objeto:

Op (2 bits)	rd Reg. destino (5 bits)	op3 (6 bits)	rs1 Reg. origen 1 (5 bits)	i		rs2 Reg. origen 2 (5 bits)
1 0	0 0 0 1 1	0 1 0 0 0 1	0 0 0 0 1	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 0
Log.	%r3	andcc	%r1			%r2

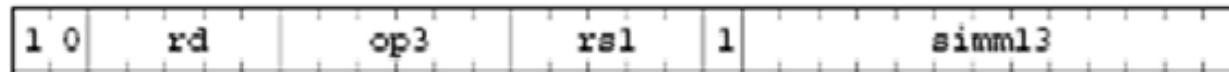
- Instrucciones lógicas:

Realiza el OR, bit por bit, entre el contenido del %r1 y la constante de valor 1, almacena el resultado en el %r1.

Lenguaje assembler:

```
lab_7: orcc %r1 , 1 , %r1
```

Formato de la instrucción:



Código objeto:

Op (2 bits)	rd Reg. destino (5 bits)	op3 (6 bits)	rs1 Reg. origen 1 (5 bits)	i	Simm13 (13 bits)
1 0	0 0 0 0 1	0 1 0 0 1 0	0 0 0 0 1	1	0 0 0 0 0 0 0 0 0 0 0 0 1
Log.	%r1	orcc	%r1		1

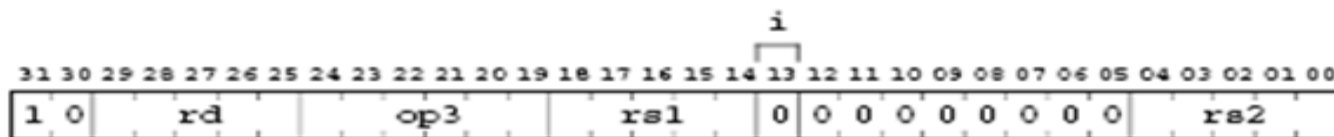
- Instrucciones lógicas:

Realiza el NOR, bit por bit, entre los contenidos de los %r1 y %r0, almacena el resultado en %r1.
Hace el complemento.

Lenguaje assembler:

```
lab_8:  orncc %r1 , %r0 , %r1
```

Formato de la instrucción:



Código objeto:

Op (2 bits)	rd Reg. destino (5 bits)	op3 (6 bits)	rs1 Reg. origen 1 (5 bits)	i		rs2 Reg. origen 2 (5 bits)
1 0	0 0 0 0 1	0 1 0 1 1 0	0 0 0 0 1	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0
Log.	%r1	orncc	%r1			%r0

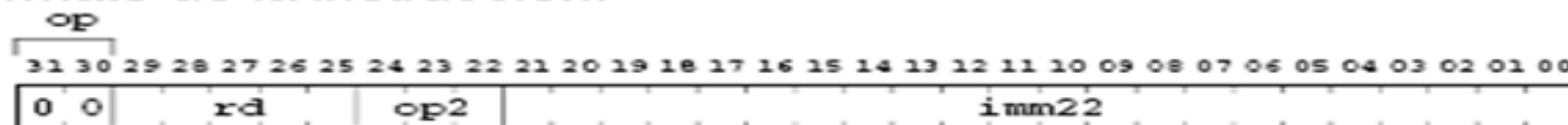
- Instrucciones lógicas:

Carga el valor hexadecimal 304F15 en los 22 bits más significativos del %r1, colocando en 0 los restantes 10 bits.

Lenguaje assembler:

```
lab_5: sethi 0x304F15 , %r1
```

Formato de la instrucción:



Código objeto:

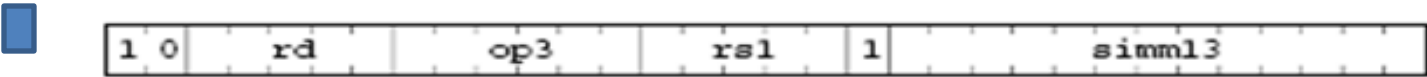
Op (2 bits)	rd Reg. destino (5 bits)	Op2 (3 bits)	imm22 (22 bits)																											
00	00001	100	1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0	0	0	1	0	1	0	1				
Sethi/ salto	%r1	sethi	Valor a cargar = 304F15(H)																											
			3	0				4	F				1	5																

- Instrucciones lógicas:
- Desplaza el contenido del %r1 tres posiciones a la derecha, almacena el resultado en %r2. Las posiciones que quedaron vacías se completan con ceros.

Lenguaje assembler:

```
lab_9:  srl      %r1 , 3 , %r2
```

Formato de la instrucción:



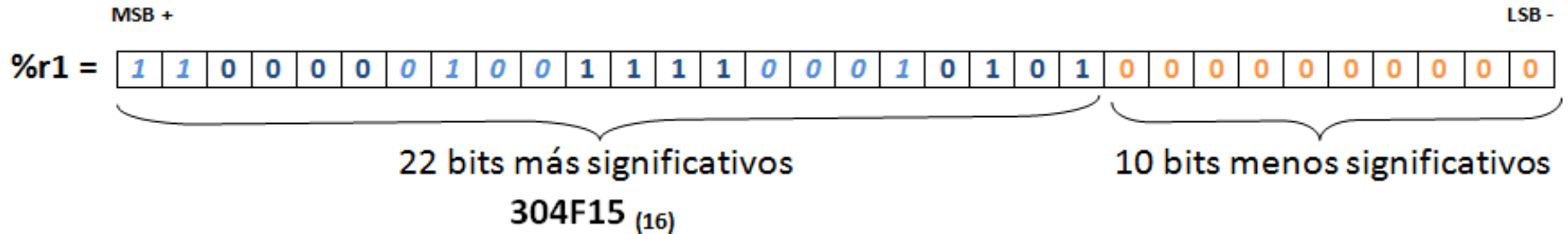
Código objeto:

Op (2 bits)	rd Reg. destino (5 bits)	op3 (6 bits)	rs1 Reg. origen 1 (5 bits)	i	Simm13 (13 bits)
1 0	0 0 0 1 0	1 0 0 1 1 0	0 0 0 0 1	1	0 0 0 0 0 0 0 0 0 0 0 1 1
Log.	%r2	srl	%r1		3

Ejemplos de aplicación

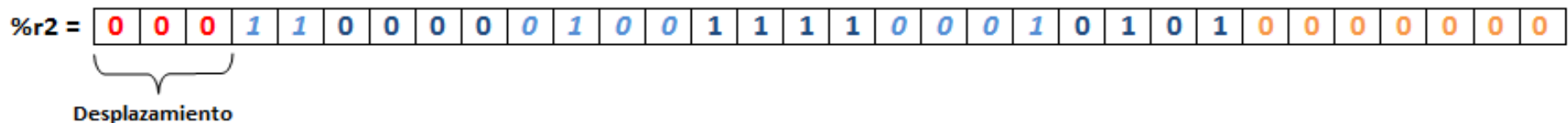
```
lab_5: sethi    0x304F15 , % r1
```

Carga el valor hexadecimal
304F15 en los 22 bits más
significativos del %r1,
colocando en 0 los restantes
10 bits.



```
lab_9:  srl    %r1, 3, %r2
```

Desplaza el contenido del %r1 tres posiciones a la derecha, almacena el resultado en %r2. Las posiciones que quedaron vacías se completan con ceros.



EJERCICIOS

Ejercicio

Escriba una instrucción en assembler que realice la **suma aritmética** usando dos registros fuentes, y el registro 5 como destino.

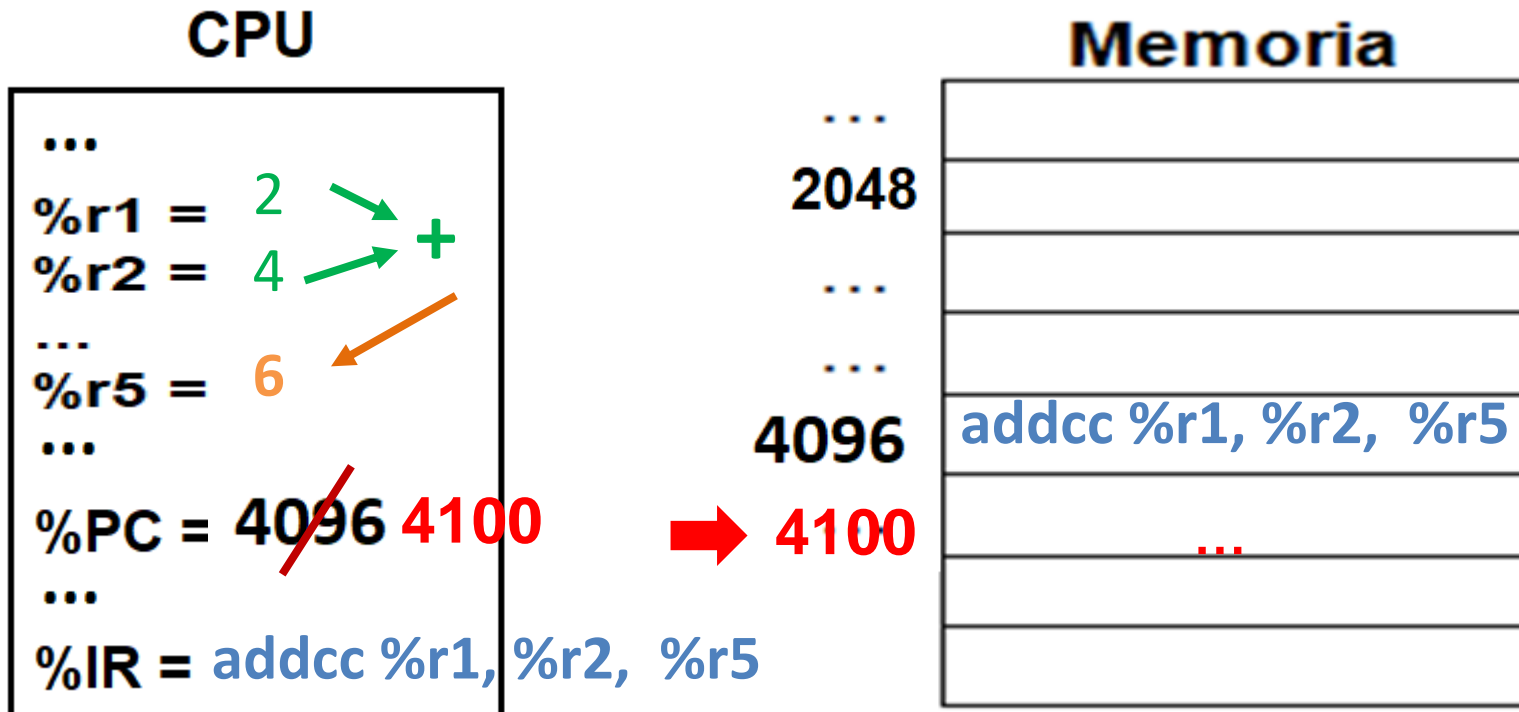
addcc %r1, %r2, %r5

Ésta es una posibilidad, podemos variar los registros a sumar.

Ejercicio

Escriba una instrucción en assembler que realice la **suma aritmética** usando dos registros fuentes, y el registro 5 como destino. **Dibuje el mapa de memoria y registros de la CPU involucrados.**

Este es un ejemplo, podemos variar los valores no indicados.



Ejercicio

Escriba una instrucción en assembler que realice la **suma aritmética** usando un registro fuente y una constante, y el registro 10 como destino.

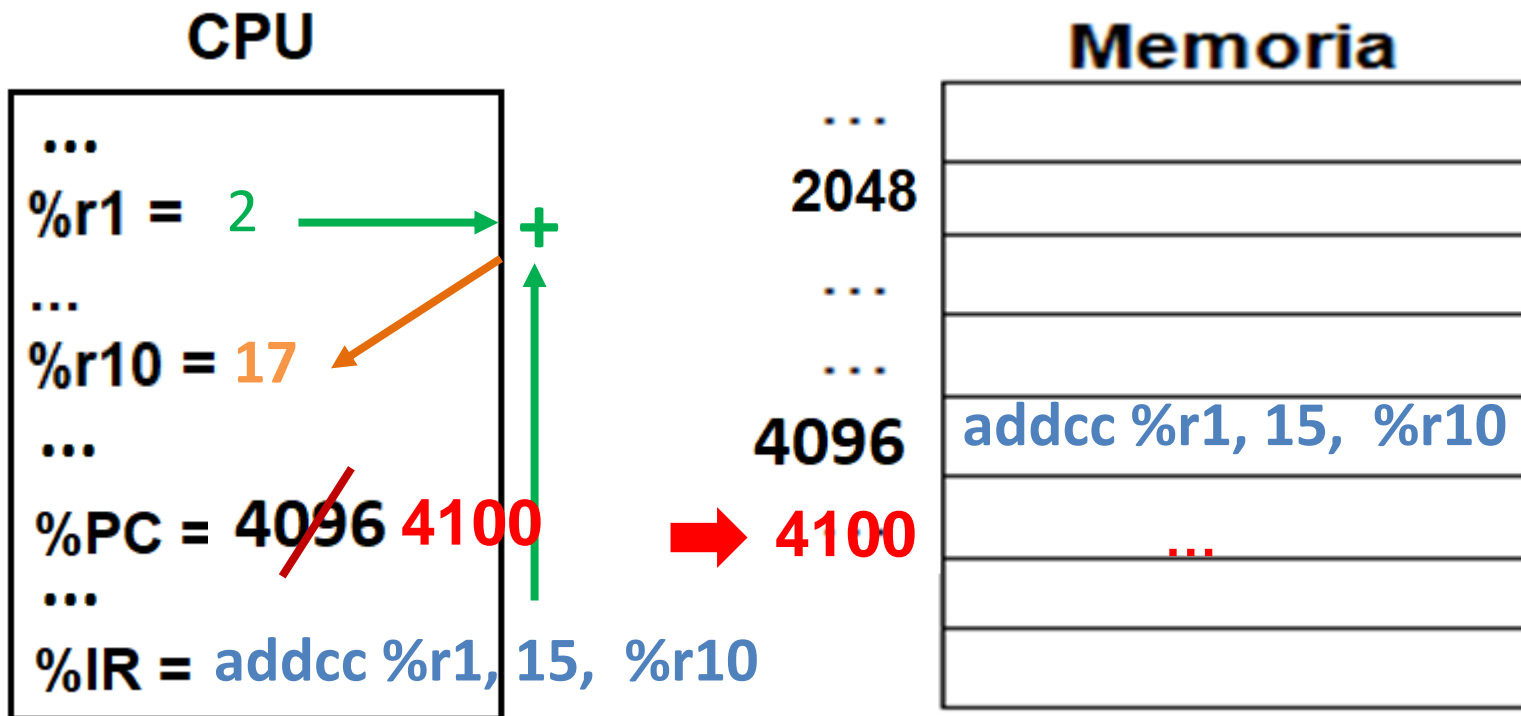
addcc %r1, 15, %r10

Ésta es una posibilidad, podemos variar el registro y la constante a sumar.

Ejercicio

Escriba una instrucción en assembler que realice la **suma aritmética** usando un registro fuente y una constante, y el registro 10 como destino. **Dibuje el mapa de memoria y registros de la CPU involucrados.**

Este es un ejemplo, podemos variar los valores no indicados.



Ejercicio

Convierta a código máquina cada una de las instrucciones escritas en assembler.

addcc %r12, %r20, %r5

10 00101 010000 01100 0 00000000 10100

addcc %r7, 15, %r10

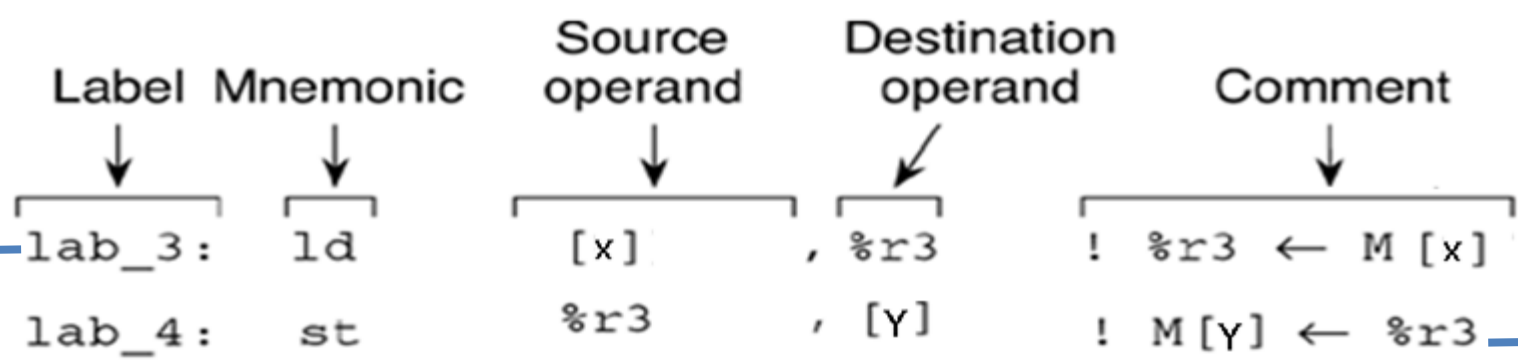
10 01010 010000 00111 1 0000000001111

INSTRUCCIONES DE MEMORIA

- **Instrucciones de memoria:**
 - La referencia al **contenido de una posición de memoria** se indica con corchetes [].
 - Los registros siempre se mencionan en función de su contenido, nunca en términos de una dirección.
 - Si no se nombra el operando origen 1 se considera que es el %r0.

- Instrucciones de memoria:

EJEMPLOS: usando etiquetas para hacer referencia al contenido de una posición de memoria [].



Al contenido de la posición X de Memoria se lo copia en el registro 3 del Trayecto de Datos.

Copia el contenido del registro 3 en la posición Y de Memoria.

ld saca (lee) de la Memoria.

st guarda (escribe) en Memoria.

- Instrucciones de memoria:

EJEMPLOS: usando registros y constantes, que sumados, hacen referencia al contenido de una posición de memoria [].

Label	Mnemonic	Source operand	Destination operand	Comment
lab_3:	ld	[%r1],	%r3	! %r3 ← M[%r1]
lab_4:	ld	[%r1+%r2],	%r3	! %r3 ← M[%r1+%r2]
lab_5:	ld	[%r1-122],	%r3	! %r3 ← M[%r1-122]
lab_6:	st	%r3,	[%r1]	! M[%r1] ← %r3
lab_7:	st	%r3,	[%r1+%r2]	! M[%r1+%r2] ← %r3
lab_8:	st	%r3,	[%r1-122]	! M[%r1-122] ← %r3

Se suman los contenidos de los %r1 y %r2, el resultado será una posición de memoria.

Lo que se encuentre en esa posición de memoria se guardará en el %r3 de la CPU.

- **Instrucciones de memoria → *código máquina*:**

- ☐ El formato de éstas instrucciones tiene tres operandos, dos de origen (rs1 y rs2) y uno de destino (rd).

- ☐ Sin embargo, las instrucciones de memoria solo necesitan dos operandos, uno para la dirección y otro para el dato.

- ☐ El campo operando restante se lo utilizará también para la dirección:

- Cuando $i = 1$ los operandos contenidos en los campos rs1 y simm13 se sumarán para obtener la dirección.

- Cuando $i = 0$ los operandos contenidos en los campos rs1 y rs2 se sumarán para obtener la dirección.

• Instrucciones de memoria:

En los ejemplos se utiliza el %r0 para rs1, con lo cual solo se especifica el operando origen restante.

En éste ejemplo supondremos que la dirección **X** es 2064.

Lenguaje assembler:

```
lab_3:  ld      [x]      , %r3      ! %r3 ← M [x]
```

Formato de la instrucción:



Código objeto:

Op (2 bits)	rd reg. destino (5 bits)					op3 (6 bits)						rs1 reg. origen 1 (5 bits)					i	Simm13 (13 bits)												
11	00011	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	1	0100000000010000														
Mem	%r3					ld												Dirección x = 2064												
																	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	

En simm13 va la dirección, en éste caso X esta en la dirección 2064, escrita en binario usando 13 bits. No es el contenido de la dirección.

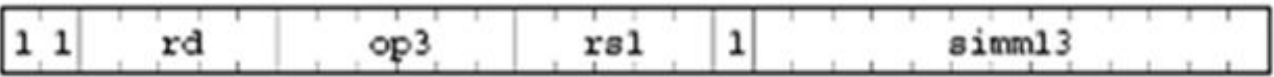
• Instrucciones de memoria:

En éste ejemplo
supondremos que la
dirección Y es 2065.

Lenguaje assembler:

```
lab_4:  st    %r3    , [Y]    ! M[Y] ← %r3
```

Formato de la instrucción:



Código objeto:

Op (2 bits)		rd Reg. destino (5 bits)					op3 (6 bits)						rs1 Reg. origen 1 (5 bits)					i	Simm13 (13 bits)																																																																
1	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1																																																			
Me m		%r3					st												Dirección y = 2065																																																																
																			<table><tr><td>4</td><td>2</td><td>1</td><td>5</td><td>2</td><td>1</td><td>6</td><td>3</td><td>1</td><td>8</td><td>4</td><td>2</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>5</td><td>2</td><td>4</td><td>2</td><td>6</td><td></td><td></td><td></td><td></td></tr><tr><td>9</td><td>4</td><td>2</td><td>2</td><td>6</td><td>8</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>6</td><td>8</td><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>													4	2	1	5	2	1	6	3	1	8	4	2	1	0	0	0	1	5	2	4	2	6					9	4	2	2	6	8								6	8	4										
4	2	1	5	2	1	6	3	1	8	4	2	1																																																																							
0	0	0	1	5	2	4	2	6																																																																											
9	4	2	2	6	8																																																																														
6	8	4																																																																																	

En el único caso en que el campo rd (registro destino) identificará un registro origen es en la instrucción st.

EJERCICIOS

Ejercicio

Escriba una instrucción en assembler que *almacene* en la dirección 4096 de memoria el contenido del registro 24.

Distintas versiones:

Y está en la posición 4096

st %r24, [Y]

st %r24, [%r0, Y]

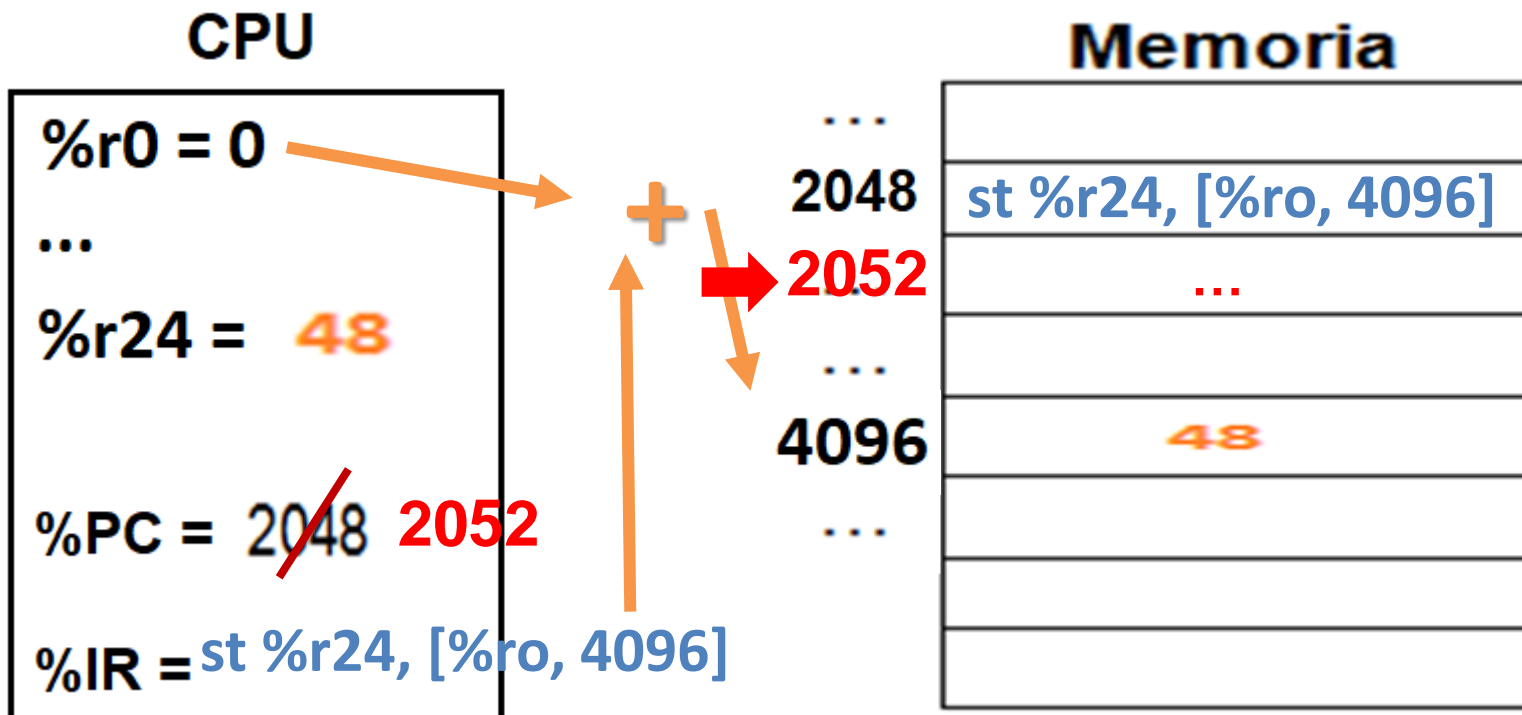
st %r24, [4096]

st %r24, [%r0, 4096]

Ejercicio

Escriba una instrucción en assembler que *almacene* en la dirección 4096 de memoria el contenido del registro 24. **Dibuje el mapa de memoria y registros de la CPU involucrados.**

Este es un ejemplo, podemos variar los valores no indicados.



Ejercicio

Escriba una instrucción en assembler que *lea* el contenido de la dirección 4096 de memoria, **use dos registros fuentes**, y el registro 24 como destino.

ld [%r1, %r2], %r24

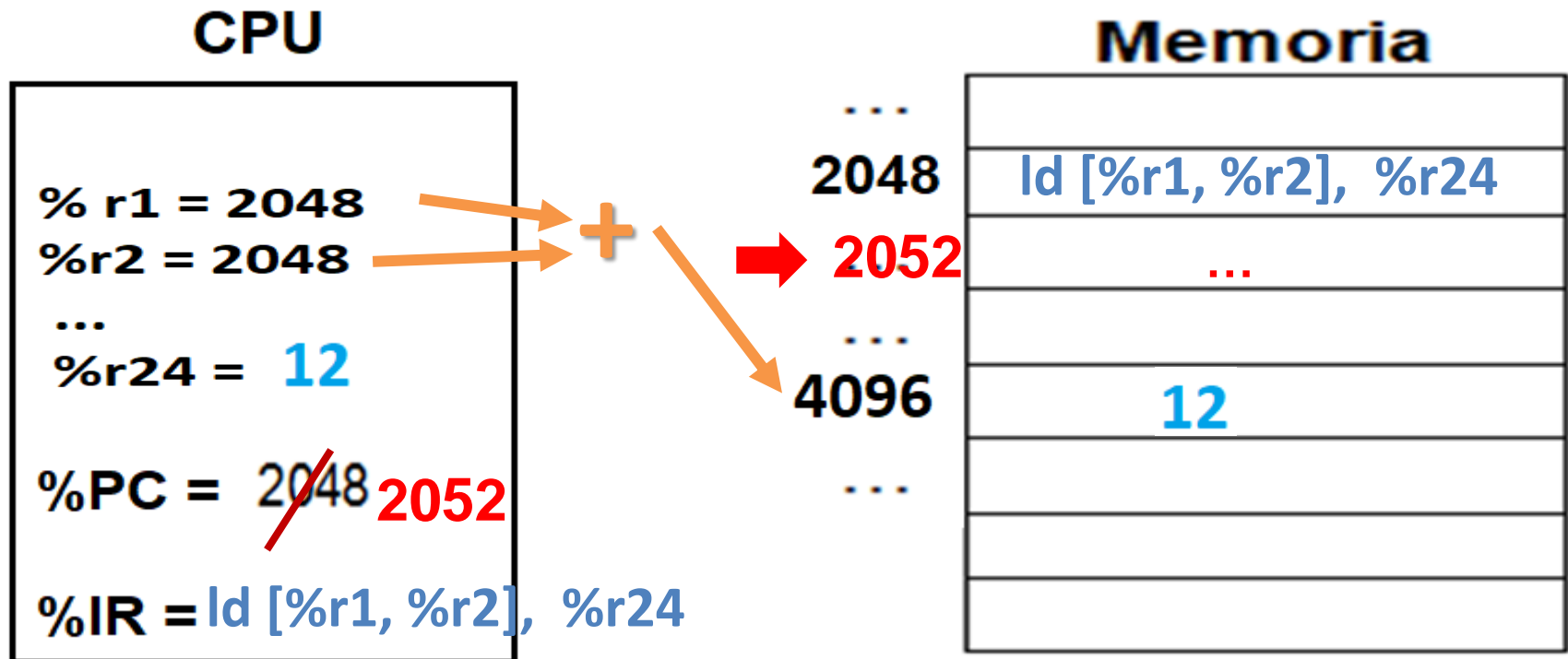
Donde %r1=2048 y %r2=2048,
sumados da 4096.

Ésta es una posibilidad, podemos variar los registros y sus contenidos a sumar.

Ejercicio

Escriba una instrucción en assembler que *lea* el contenido de la dirección 4096 de memoria, use dos registros fuentes, y el registro 24 como destino. Dibuje el mapa de memoria y registros de la CPU involucrados.

Este es un ejemplo, podemos variar los valores no indicados.



Ejercicio

Convierta a código máquina cada una de las instrucciones escritas en assembler.

ld [%r1, %r2], %r24

11 11000 000000 00001 0 00000000 00010

st %r24, [Y] ! Y está en la posición 4096

11 11000 000100 00000 1 10000000000000

INSTRUCCIONES DE CONTROL

En éstas instrucciones, la referencia a la **dirección de una posición de memoria** se indica **sin corchetes []**, se usan **etiquetas o cantidad de palabras que saltan** para llegar a la dirección deseada.

Algunas definiciones

- Una Subrutina, llamada también función o procedimiento, es una secuencia de instrucciones a la que se invoca como si se tratara de una única instrucción.

Cuando el programa llama a la subrutina se transfiere el control a ésta, la que ejecuta la secuencia de instrucciones requerida, tras lo cual vuelve a la posición inmediatamente siguiente a la que generó el llamado.

Mas adelante ampliaremos éste tema.

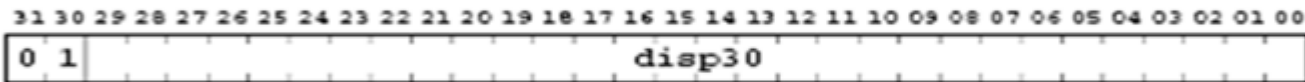
• Instrucciones de control:

Es una operación de llamado y enlace. Invoca una subrutina, que comienza en la posición de memoria sub_r, y almacena la dirección de la instrucción actual, que esta en **PC**, en el **%r15**.

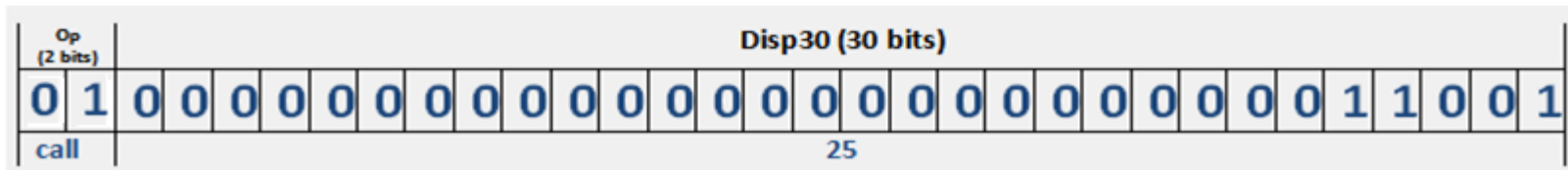
Lenguaje assembler:

```
lab_10:  call sub_r
```

Formato de la instrucción:



Código objeto:



Se considera que sub_r se encuentra ubicada 25 **palabras** (25 x 4 = 100 bytes) después de la instrucción call.

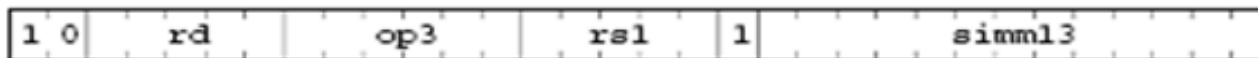
• Instrucciones de control:

Retorno de subrutina. Debe volver a la dirección siguiente a **call**, que se la calcula como el valor de **PC**, que esta en **%r15**, **más 4**, el resultado se dice que se descarta en **%r0**, *pero internamente la arquitectura guarda el destino en el PC, que es el %r32, pero como no se puede escribir 32 con 5 bits se dice que se descarta en el %r0.*

Lenguaje assembler:

```
lab_11:  jmpl %r15 + 4 , %r0
```

■ Formato de la instrucción:



Código objeto:

Op (2 bits)	rd Reg. destino (5 bits)	op3 (6 bits)	rs1 Reg. origen 1 (5 bits)	i	Simm13 (13 bits)
10	00000	111000	01111	1	00000000000100
	%r0	jmpl	%r15		4

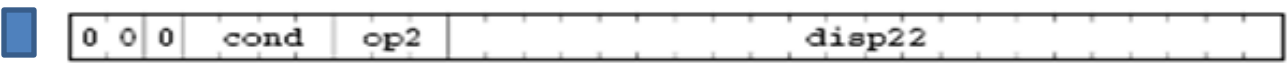
• Instrucciones de control (de Salto condicional):

Salta a label si el código de condición **z = 1**, que indica que todos los bits del registro que contiene el resultado son ceros. Si z=0 sigue con la instrucción siguiente a be en el programa.

Lenguaje assembler:

```
lab_12: be label
```

Formato de la instrucción:



Código objeto:

Op (2 bits)		Cond. (4 bits)	Op2 (3 bits)	Disp22 (22 bits)
0 0	0	0 0 0 1	0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
Salto		be	Branch o salto	5

En el formato de salto el bit 29 siempre vale 0.

Se considera que label se encuentra en una dirección de memoria 5 **palabras** (5 x 4 = 20 bytes) mayor que la de la instrucción be.

• Instrucciones de control (de Salto condicional):

Lenguaje assembler:

```
lab_13: bneg label
```

Formato de la instrucción:

Salta a label si el código de condición **n=1**, que indica que el primer bit más significativo (el del signo) del registro que contiene el resultado es 1, o sea es un número negativo. Si n=0 sigue con la instrucción siguiente a bneg en el programa.



Código objeto:

Op (2 bits)			Cond. (4 bits)				Op2 (3 bits)			Disp22 (22 bits)																		
0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
Salto			bneg				Branch o salto			5																		

Se considera que label se encuentra en una dirección de memoria 5 palabras (5 x 4 = 20 bytes) mayor que la de la instrucción bneg.

- **Instrucciones de control (de Salto condicional):**

Lenguaje assembler:

lab_14: bcs label

Formato de la instrucción:



Código objeto:

Op (2 bits)	Cond. (4 bits)	Op2 (3 bits)	Disp22 (22 bits)
0 0 0	0 1 0 1	0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
Salto	bcs	Branch o salto	5

Se considera que label se encuentra en una dirección de memoria 5 **palabras** ($5 \times 4 = 20$ bytes) mayor que la de la instrucción bcs.

Salta a label si el código de condición **c =1**, que indica que ocurrió un arrastre o acarreo al realizar los cálculos en la ALU. Si c=0 sigue con la instrucción siguiente a bcs en el programa.

- **Instrucciones de control (de Salto condicional):**

Salta a label si el código de condición **v=1**, que indica que ocurrió un desborde al realizar los cálculos en la ALU. Si v=0 sigue con la instrucción siguiente a bvs en el programa.

Lenguaje assembler:

lab_15: bvs label

Formato de la instrucción:



Código objeto:

Op (2 bits)			Cond. (4 bits)				Op2 (3 bits)			Disp22 (22 bits)																			
0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
Salto			bvs				Branch o salto			5																			

Se considera que label se encuentra en una dirección de memoria 5 palabras (5 x 4 = 20 bytes) mayor que la de la instrucción bvs.

- Instrucciones de control (de Salto *incondicional*):

Lenguaje assembler:

lab_16: ba label

Salta a label,
independientemente de los
valores que adopten los códigos
de condición.

Formato de la instrucción:



Código objeto:

Op (2 bits)		Cond. (4 bits)	Op2 (3 bits)	Disp22 (22 bits)
0 0	0	1 0 0 0	0 1 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1
Salto		ba	Branch o salto	Complemento a 2 de 5

Se considera que label se encuentra en una dirección de memoria 5 **palabras** (5 x 4 = 20 bytes) **menor** que la de la instrucción ba.

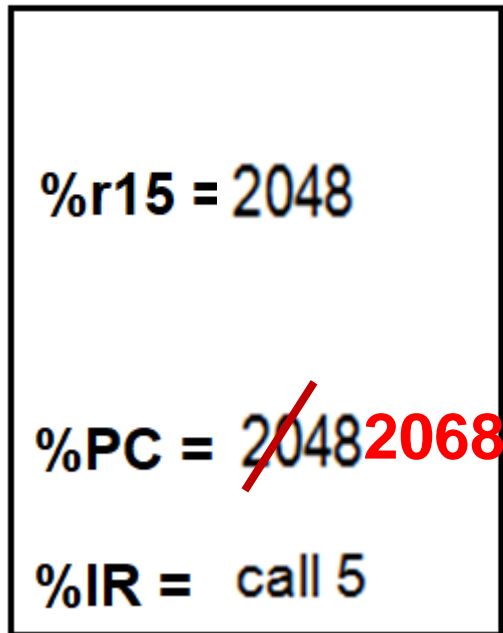
EJERCICIOS

Ejercicio

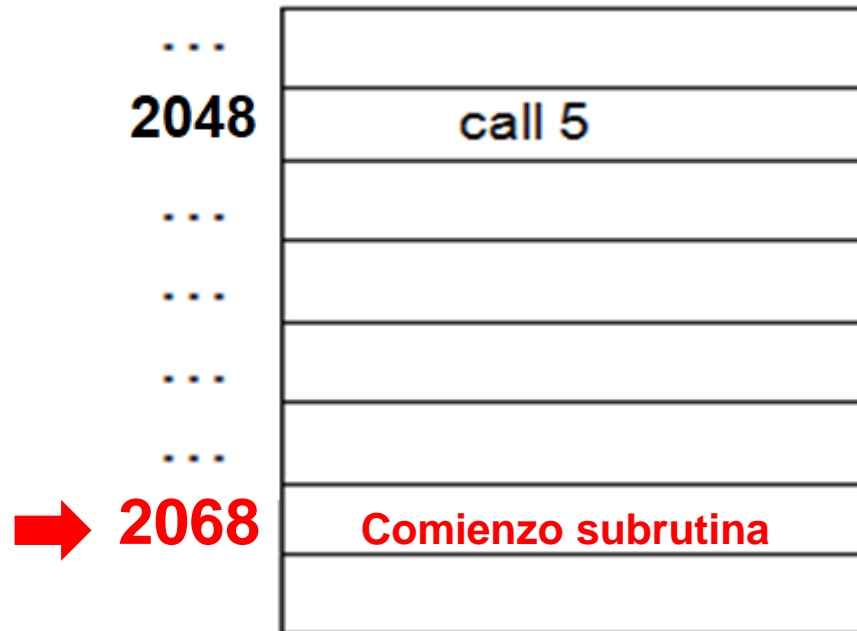
Escriba una instrucción de Llamado a Subrutina, **usando un valor**, no etiqueta, para la cantidad de palabras a saltar. Dibuje el mapa de memoria.

Esto es un ejemplo, podemos variar la cantidad de palabras a saltar y la dirección de la instrucción .

CPU



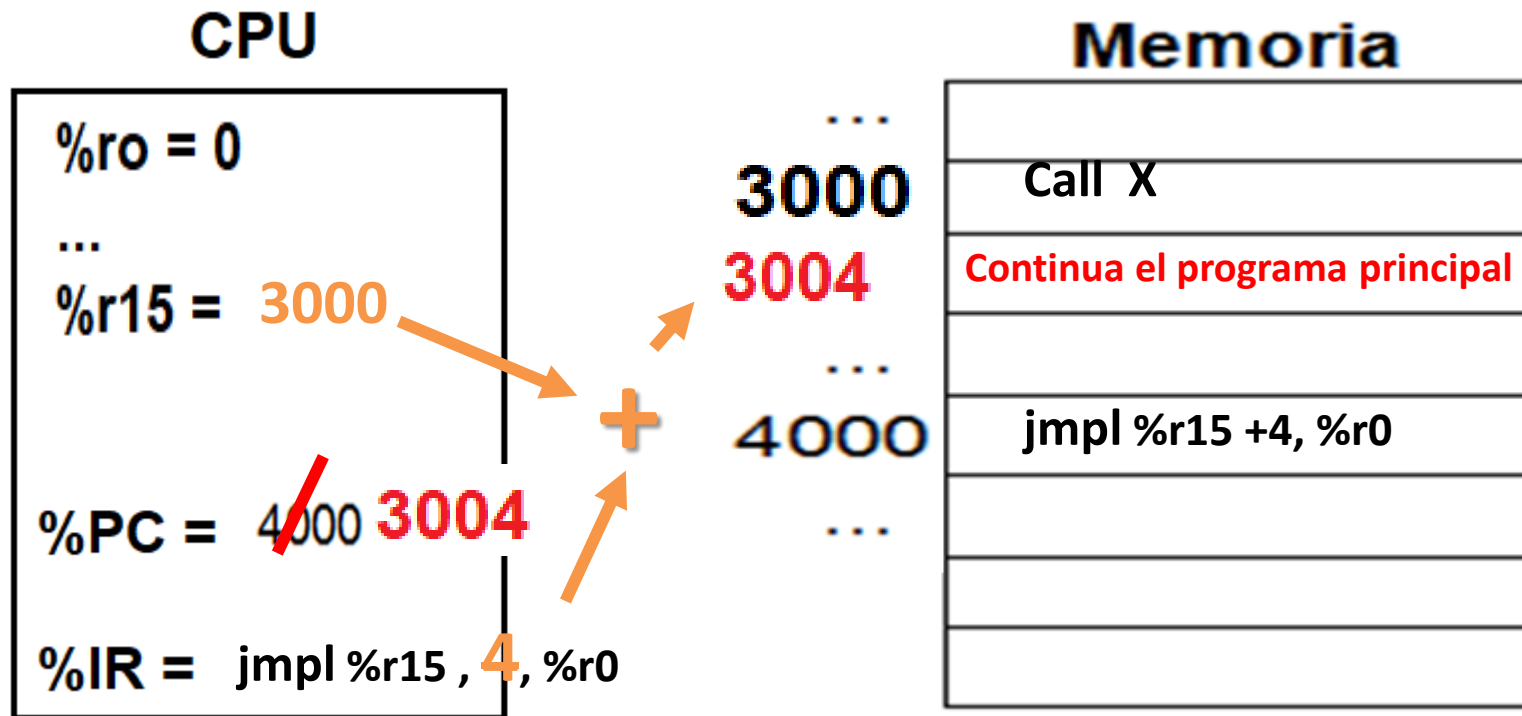
Memoria



$2048 + (4 \text{ registros por palabras} \times 5 \text{ palabras a saltar}) = 2048 + 20 = 2068$ dirección destino del salto, donde comienza la subrutina.

Ejercicio

Luego de ejecutar la instrucción **jmp $\%r15 + 4, \%r0$** , almacenada en la dirección 4000, y tomando como referencia que la última instrucción **call** esta almacenada en la dirección 3000, **¿cuál sería la nueva dirección del PC al terminar jmp?** Dibuje el mapa de memoria.



Ejemplo de Llamado a Subrutina

Se usa una subrutina para realizar una suma de dos números contenidos en registros.

! Rutina invocante

:

ld [x], %r1

ld [y], %r2

call add_1

st %r3, [z]

:

x: 53

y: 10

z: 0

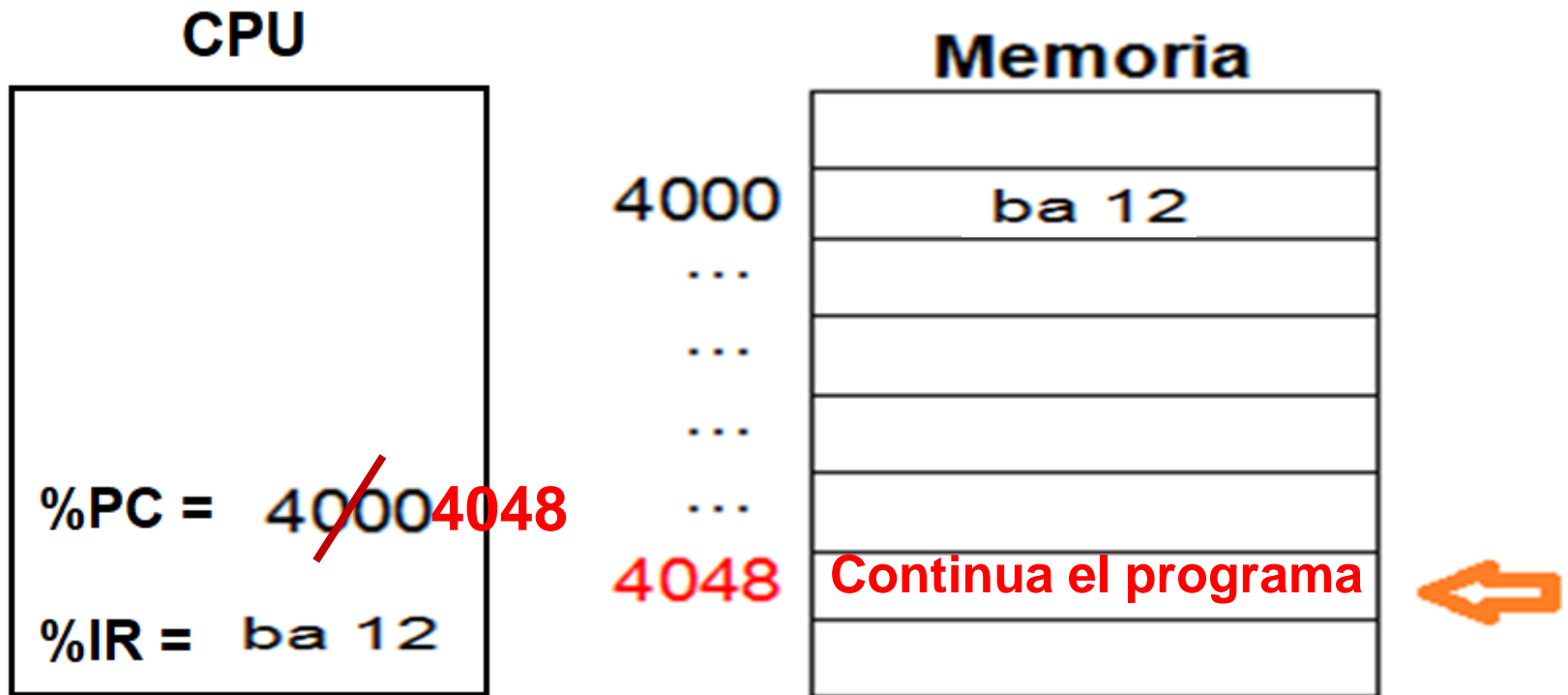
! Rutina invocada

! %r3 \leftarrow %r1 + %r2

add_1: addcc %r1, %r2, %r3
 jmpl %r15 + 4, %r0

Ejercicio

Dada la instrucción **ba 12**, almacenada en la dirección 4000 de memoria. ¿Cuál sería la dirección hacia donde salta?. Dibuje el mapa de memoria.



$4000 + (4 \text{ registros por palabras} \times 12 \text{ palabras a saltar}) = 4000 + 48 = 4048$ dirección a donde salta.

Ejercicio

- *Convierta a código máquina cada una de las instrucciones escritas en assembler.*

call 5

[illegible]

jmpl %r15 +4, %r0

10 00000 111000 01111 1 000000000000100

ba 12

00 0 1000 010 0000000000000000000000001100

PROGRAMAS EN LENGUAJES DE BAJO NIVEL

Directivas o Pseudo Operaciones ARC

- No son códigos de operación, son instrucciones para que el ensamblador realice ciertas operaciones en el momento del ensamble.
- Siempre **comienzan con un punto (.)**.

Pseudo-Op	Uso	Significado
<code>.equ</code>	<code>X .equ #10</code>	Asignar a X el valor $(10)_{16}$
<code>.begin</code>	<code>.begin</code>	Comienzo de traducción
<code>.end</code>	<code>.end</code>	Fin de traducción
<code>.org</code>	<code>.org 2048</code>	Cambiar contador de posición a 2048
<code>.dwb</code>	<code>.dwb 25</code>	Reservar un bloque de 25 palabras
<code>.global</code>	<code>.global Y</code>	Y se usa en otro módulo
<code>.extern</code>	<code>.extern Z</code>	Z está definido en otro módulo
<code>.macro</code>	<code>.macro M a, b, ...</code>	Definir macro M con parámetros formales: a, b, ...
<code>.endmacro</code>	<code>.endmacro</code>	Fin de definición de Macro
<code>.if</code>	<code>.if <cond></code>	Ensamblar si <cond> es cierta
<code>.endif</code>	<code>.endif</code>	Fin estructura condicional

Algunas directivas las veremos más adelante.

Ejemplo de Programa ARC

Un programa ARC en lenguaje assembler que suma dos enteros:

! Este programa suma dos numeros enteros

```
.begin
.org 2048
prog1: ld      [x], %r1      !Carga x en %r1
      ld      [y], %r2      !Carga y en %r2
      addcc   %r1, %r2, %r3  !%r3 ← %r1 + %r2
      st      %r3, [z]      !Guarda %r3 en z
      jmp1    %r15 + 4, %r0  !Retorna de subrutina

x:     15
y:     9
z:     0
      .end
```

A partir de la dirección 2048 se cargará el código.

Como solo se pueden sumar números que estén almacenados en registros de ARC, el programa comienza cargando en %r1 y %r2 los contenidos de la direcciones X e Y de la memoria.

Ejercicio

Indique cuales son las etiquetas, las directivas, las instrucciones, los datos y los comentarios.



Ejercicio

Al programa en assembler que suma dos números enteros páselo a código máquina.

2048	ld [x], %r1	1100 0010 0000 0000 0010 1000 0001 0100
2052	ld [y], %r2	1100 0100 0000 0000 0010 1000 0001 1000
2056	addcc %r1,%r2,%r3	1000 0110 1000 0000 0100 0000 0000 0010
2060	st %r3, [z]	1100 0110 0010 0000 0010 1000 0001 1100
2064	jmp1 %r15+4, %r0	1000 0001 1100 0011 1110 0000 0000 0100
2068	15	0000 0000 0000 0000 0000 0000 0000 1111
2072	9	0000 0000 0000 0000 0000 0000 0000 1001
2076	0	0000 0000 0000 0000 0000 0000 0000 0000

SIMULARDOR

ARC

(disponible en el Aula Virtual)

Ensambladores de dos pasadas

- La mayoría de los ensambladores recorren dos veces el texto escrito en lenguaje simbólico.
- En la **primera pasada** determinan:
 - *direcciones* de todos los datos e instrucciones del programa.
 - seleccionan que instrucción del lenguaje máquina debe generarse para cada instrucción en lenguaje simbólico, pero sin generar aún el código máquina.
 - realizan operaciones aritméticas.
 - insertan las definiciones de etiquetas y constantes en una **tabla de símbolos**. A cada símbolo (etiqueta) se le ingresa en la tabla el valor correspondiente a la posición en que se encuentran.
- En la **segunda pasada** se genera el **código máquina**, insertando en el mismo los valores de los símbolos ya conocidos de la tabla anterior.

Ejemplo de Programa ARC

Un programa que suma cinco números

Figura 4.14, página 124,
capítulo 4 del libro del
autor Murdocca.

```
! Este programa suma LENGTH numeros
! Uso de los Registros: %r1 - Longitud del arreglo a
!                               %r2 - Dirección de inicio del arreglo a
!                               %r3 - La suma parcial
!                               %r4 - Puntero dentro del arreglo a
!                               %r5 - Contiene un elemento de a

        .begin                                ! Comienzo del ensamblado
        .org 2048                             ! Inicio del programa en 2048
a_start .equ 3000                             ! Dirección del arreglo a
        ld [length], %r1 ! %r1 ← long. del arreglo a
        ld [address], %r2 ! %r2 ← dirección de a
        andcc %r3, %r0, %r3 ! %r3 ← 0
loop:   andcc %r1, %r1, %r0 ! Verifica nº de elementos restantes.
        be done           ! Finalizar cuando length=0
        addcc %r1, -4, %r1 ! Decrementar longitud arreglo
        addcc %r1, %r2, %r4 ! Dirección próximo elemento
        ld %r4, %r5        ! %r5 ← Memoria[%r4]
        addcc %r3, %r5, %r3 ! Sumar nuevo elemento en r3
        ba loop            ! Repetir lazo.

done:   jmp1 %r15 + 4, %r0 ! Retorno a rutina de llamada

length: 20                ! 5 numeros (20 bytes) en a
address: a_start
        .org a_start      ! Inicio del arreglo a
a:       25                ! length/4 valores siguientes
        -10
        33
        -5
        7

        .end                                ! Fin ensamblado
```

Ejercicio

Indique cuales son las etiquetas, las directivas, las instrucciones, los datos y los comentarios.

