Departamento de Tecnología Electrónica
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Programación en ensamblador

Estructura de Computadores

P. Ruiz de Clavijo Vázquez <pruiz@us.es>

Este manual es material complementario para teoría y laboratorio de la asignatura de primer curso Estructura de Computadores de la titulación Grado en Ingeniería Informática, Ingeniería del software. Se ha desarrollado como apoyo docente

1. Computador CS2010

El objetivo de este manual es aprender a usar el lenguaje ensamblador del procesador CS2010 mediante ejemplos. En el manual no se explica la arquitectura interna del procesador, el manual describe el uso del procesador CS2010 desde el nivel del programador. Se explicará el juego de instrucciones ensamblador mediante ejemplos comparados con Java y/o C. Al final del documento también se explica como obtener el código máquina de los programas en ensamblador.

A lo largo del documento se han resaltado en rojo aquellos conceptos teóricos que el alumno debe dominar para superar la evaluación de esta parte del temario. Muchos de los ejemplos en ensamblador mostrados se comparan con los lenguajes de alto nivel Java/C, pero considere que no siempre existe una equivalencia exacta. Java y C tienen muchas sentencias y estructuras básicas funcionales idénticas. Antes de empezar con el juego de instrucciones en ensamblador se propone partir de un símil del ensamblador CS2010 con el lenguaje Java/C. Éste símil se utilizará a lo largo de este documento y consiste en conocer qué recursos están disponibles en el CS2010 para trabajar con datos e intentar usarlos en Java/C con las mismas restricciones existentes en la programación ensamblador. Así, los recursos disponibles en el CS2010 a nivel de programación ensamblador son:

- El CS2010 dispone de 8 registros de 8 bits para operar con ellos: R0,R1,R2,R3,R4,R5,R6 y R7.
- El CS2010 dispone de una memoria RAM de 256 palabras de 8 bits.

Para poder comparar Java/C con el lenguaje ensamblador de la forma más congruente posible, en todos los programas mostrados en Java/C las variables están limitadas a:

- Se dispondrá únicamente de 8 variables globales declaradas como: `int R0,R1,R2,R3,R4,R5,R6,R7;`
- Existe un único vector/array global correspondiente a toda la memoria RAM declarado como: `int RAM[]=new int[256];` (Java) o `int RAM[256];` (C).

Para ser congruente con el ensamblador, en los ejemplos Java/C no se puede declarar ninguna otra variable ni global ni local. En ensamblador hay que realizar siempre los programas con los 8 registros y la memoria RAM.

Siendo meticulosos se puede decir que las variables tipo `int` de Java/C son de 32bits, y no de 8bits, pero para los ejemplos mostrados en este manual no se considera relevante.

2. Juego de instrucciones del CS2010

El juego de instrucciones del CS2010 está contenido en una tabla de instrucciones, junto a 2 tablas más (tabla de formato y tabla de saltos condicionales) que forman el material de examen para esta parte de la asignatura. A lo largo de este manual se irán presentando las tablas según se necesiten. Se comenzará por algunas instrucciones con 2 operandos:

- ADD: Sumar 2 registros.
- SUB: Restar 2 registros.
- MOV: Copiar el valor de registro en otro.
- LDI: Inicializa un registro con un valor. Al valor se le llama dato o valor inmediato.

En general, el lenguaje ensamblador de los procesadores no dispone de instrucciones con más de 2 operandos. Por ello cuando se realiza una operación binaria (con 2 operandos como A+B) el resultado sobreescribe a alguno de los dos operandos Es muy importante conocer cual es el orden de los operandos en cada una de las instrucciones para no cometer errores. En el CS2010 las operaciones con 2 operandos siempre guardan el resultado de la operación en el operando derecho, por ejemplo, ADD R0,R1 realiza la suma, y equivale en Java/C a R0=R0+R1. En la suma anterior no importaría equivocarse en el orden de los sumandos, pero por ejemplo, la resta SUB R1,R2 corresponde en Java/C a R1=R1-R2, si se invierten los operandos se cometería un error.

En el caso de instrucciones como LDI cuyo segundo operando es un valor denominado inmediato, y debe considerar lo siguiente:

1. El procesador CS2010 es de 8 bits, por lo que el valor inmediato estará en el rango 0-255 (en base 10) ó $00-$FF en base 16 (hexadecimal).

2. El valor inmediato puede estar en diferentes bases numéricas. Se establece la siguiente notación para la programación en lenguaje ensamblador:
   2.1. Si está precedido por el símbolo $ se considera que el valor es hexadecimal
   2.2. Si está precedido por 0x se considera el valor hexadecimal, siendo esta notación compatible con Java/C.
   2.3. Si no está precedido por ningún símbolo se considera base 10, del mismo modo que en los programas Java/C.

Ejemplo 1.- Dados los programas indicados indique el contenido de los registros tras la ejecución.

Programa

```
LDI R1,23
LDI R2,$23
LDI R3,0x23
STOP
```

Registros

Reg	Contenido
R0	-
R1	$17
R2	$23
R3	$23
...	...

Programa

```
LDI R1,$28
LDI R2,$02
ADD R1,R2
SUB R2,R2
STOP
```

Registros

Reg	Contenido
R0	-
R1	$2A
R2	$00
R3	-
...	...

Programa

```
LDI R0,$02
LDI R1,$FF
ADD R1,R0
SUB R2,R2
SUB R2,R1
STOP
```

Registros

Reg	Contenido
R0	$02
R1	$01
R2	$FF
R3	-
...	...

Programa

```
LDI R0,$44
ADD R0,R0
ADD R0,R0
ADD R0,R0
STOP
```

Registros

Reg	Contenido
R0	$20
R1	-
R2	-
R3	-
...	...

En las operaciones aritméticas de los ejemplos anteriores hay que ser cuidadoso y entender que se están haciendo con una ALU de 8 bits. En una suma o resta en ensamblador, el registro que recibe el dato captura el valor de la salida de la ALU, pero ignorando el posible acarreo (carry), por eso $FF+$02 = $01 y en la resta ocurre lo mismo. Realmente el acarreo no se pierde, el CS2010 lo captura en un biestable para su posterior uso, pero esto se explicará en otra sección de este manual.

Otro aspecto importante del ensamblador la es saber trabajar correctamente con aritmética con signo o sin signo. En el ejemplo anterior se realizó esta resta $00 - 01 = $FF y se obtuvo el resultado indicado ($FF). Si fueran números con signo y en notación complemento a 2 con 8 bits, se puede comprobar que $FF corresponde al valor -1. En cambio si es sin signo, el resultado es erróneo.

Entonces, ¿está operando el CS2010 con números con signo o sin signo?. La respuesta es que la ALU trabaja con números binarios cuya aritmética es compatible con la notación complemento a 2. Este aspecto ya se estudió en la asignatura CED del primer cuatrimestre. Pero si terminó de entenderlo puede usar la siguiente regla: el CS2010 dispone de una ALU con un sumador que opera en aritmética con signo o signo indiferentemente, realmente es el programador al usar los registros o valores quien decide si está operando con o sin signo. Tanto la operación ADD como SUB son válidas para ambas notaciones. No se recomienda mezclar en un programa operaciones entre registros y/o valores con signo y sin signo, a no ser que entienda perfectamente lo que está haciendo.

Tratadas las operaciones aritméticas básicas, en los siguientes ejemplos se va a trabajar sobre instrucciones de acceso a memoria RAM, concretamente con las siguientes:

- STS: Copiar el valor de registro en una dirección concreta de memoria RAM.
- LDS: Copiar un dato desde una dirección concreta de la memoria RAM a un registro.

Para mejorar la comprensión a partir de ahora se intentará mostrar un código Java/C, parecido en funcionalidad a los ejemplos ensamblador. Considere que no hay una correspondencia exacta entre este código de alto nivel y los ejemplos mostrados.

Ejemplo 2.- Dado el programa mostrado y algunas palabras de memoria RAM con un valor inicial obtenga el valor final de los registros y de la memoria RAM. (Los valores modificados se muestran en rojo).

Programa ensamblador

```
LDS R5,$51
LDS R6,$52
ADD R5,R6
STS $0x53,R5
LDS R5,$51
SUB R6,R5
STS $0x01,R6
STOP
```

Memoria RAM

Dirección	Contenido
0x00	$00
0x01	$01
...	
0x50	$00
0x51	$15
0x52	$16
0x53	$2B
...	...

Registros

Reg	Contenido
R0	
R1	
R2	
R3	
R4	
R5	$15
R6	$01
...	...

Para comprender mejor el programa anterior puede intentar seguir la ejecución del fragmento de código escrito en Java/C con la misma funcionalidad. Hay se suponer que en Java/C el array RAM ya está inicializado a los valores de la tabla anterior:

Programa ensamblador

```
LDS R5,$51
LDS R6,$52
ADD R5,R6
STS $0x53,R5
LDS R5,$51
SUB R6,R5
STS $0x01,R6
STOP
```

Código Java/C

```
int RAM[]=new int[256];
int R0,R1,R2,R3,R4,R5,R6,R7;

R5 = RAM[0x51];
R6 = RAM[0x52];
R5 = R5 + R6;
RAM[0x53] = R5;
R5 = RAM[0x51];
R6 = R6 – R5;
RAM[0x01] = R6;
```

En la comparación anterior del código Java/C con ensamblador se observa que para cada instrucción ensamblador hay una equivalente Java/C y viceversa. Esto ocurre en este ejemplo porque está preparado para este fin, pero para cualquier sentencia en Java/C no se puede obtener una única instrucción ensamblador equivalente. Normalmente por cada sentencia Java/C necesitará varias instrucciones en ensamblador.

Ejemplo 3.- Se desea rellenar la memoria RAM a partir de la palabra $50 con la secuencia 5, 6, 7, 8, 9, 10, 11, 12, 13, 14.

Memoria RAM deseada

Dirección	Contenido
...	...
$50	5
$51	6
$52	7
$53	8
$54	9
$55	10
$56	11
$57	12
...	...

Propuesta 1

```
LDI R0,5
STS $0x50,R0

LDI R0,6
STS $0x51,R0

LDI R0,7
STS $0x52,R0

LDI R0,8
STS $0x53,R0

LDI R0,9
STS $0x54,R0

LDI R0,10
STS $0x55,R0
...
```

Propuesta 2

```
LDI R0,$5
LDI R1,1
LDI R2,$50

ST  (R2),R0
ADD R0,R1
ADD R2,R1

ST  (R2),R0
ADD R0,R1
ADD R2,R1

ST  (R2),R0
ADD R0,R1
ADD R2,R1

...
```

En la propuesta 1 se inicializa directamente cada palabra de memoria con la instrucción STS y en la propuesta 2 se usa la instrucción ST. Esta instrucción está pensada para escribir en memoria RAM usando un índice, por ejemplo, `ST (R2),R0` equivale en Java/C a `RAM[R2] = R0;`. En realidad, para este tipo de programa ninguna de las 2 soluciones sería válida, se debería hacer mediante un bucle como el de la solución propuesta a continuación (en Java/C):

Código Java/C

```
int RAM[]=new int[256];
int R0,R1,R2,R3,R4,R5,R6,R7;

R0 = 5;
R2 = 50;
while(R0 != 15)
{
    RAM[R2] = R0;
    R0 = R0 + 1;
}
```

Fíjese en la propuesta 2 como cada bloque de 3 instrucciones es repetitivo. Ese bloque de 3 instrucciones formaría el cuerpo de un bucle, pero los bucles en ensamblador se tratarán en la sección 4, ahora el objetivo es comprender el funcionamiento de las siguientes instrucciones de acceso a memoria:

- LD: Copiar una palabra de memoria a un registro usando otro registro como índice.
- ST: Copiar un registro a una dirección de memoria usando otro registro como índice.

Llegados a este punto es fundamental conocer una nomenclatura para los operandos de las instrucciones en función del tipo de datos que utilice. Las instrucciones según el tipo de dato que usan y como acceden a él se pueden agrupar en tipos de modos de direccionamiento. Los modos de direccionamiento se consideran un concepto fundamental en esta asignatura y son los siguientes:

- Direccionamiento inmediato.
- Direccionamiento directo de registro.
- Direccionamiento directo de memoria (también llamado direccionamiento de memoria absoluto).
- Direccionamiento indirecto de memoria.

Las instrucciones LD/ST operan en modo indirecto de memoria. Para entender todos los modos s continuación se muestran ejemplos con diferentes modos de direccionamiento, comparados con una posible equivalencia en Java/C.

Ejemplo 4.- ¿Qué tipo de direccionamiento utiliza cada unas de las instrucciones en ensamblador en cada uno de sus 2 operandos?

Programa Ensamblador	*Ejemplo Java*
	`int RAM[]=new int[256];` `int R0,R1,R2,R3,R4,R5,R6,R7;`
`LDI R2,$01`	`R0 = 0x01;`
`LDS R3,$A0`	`R3 = RAM[0xA0];`
`LD  R4,(R2)`	`R4 = RAM[R2];`
`STS $A1,R4`	`RAM[0xA1] = R4`
`ST (R5),R0`	`RAM[R5] = R0`

El modo de direccionamiento indirecto a memoria se utiliza principalmente para trabajar con vectores (arrays) de datos. En ensamblador existe una memoria RAM que se puede considerar como un array de 256 palabras, de hecho, se puede considerar como el único array disponible para el programador en ensamblador. Como no pueden declarar nuevos arrays de datos, para trabajar con varios se usan fragmentos de RAM, es decir, se particiona la memoria RAM en fragmentos de un tamaño determinado estableciendo claramente la posición de origen de cada vector y su tamaño. Evidentemente, cuando se usen varios vectores sobre la única memoria RAM existente, éstos no deben solaparse.

En el siguiente ejemplo se muestra un programa que trabaja con un vector, aunque se debería hacer con un bucle, como ya se ha dicho se tratará posteriormente.

Ejemplo 5.- El programa indicado usa R0 como resultado de una operación sobre un vector de 5 elementos. La memoria RAM ya tiene los valores inicializados. ¿Qué operación es la que realiza el programa?. Rellene el resultado final con el contenido de los registros

Programa

```
LDI R0,$00
LDI R1,$01
LDI R2,$A0

LD  R3,(R2)
ADD R0,R3
ADD R2,R1

LD  R3,(R2)
ADD R0,R3
ADD R2,R1

LD  R3,(R2)
ADD R0,R3
ADD R2,R1

LD  R3,(R2)
ADD R0,R3
ADD R2,R1

LD  R3,(R2)
ADD R0,R3
ADD R2,R1

STOP
```

Memoria RAM

Dirección	Contenido
...	
0xA0	$02
0xA1	$05
0xA2	$12
0xA3	$0A
0xA4	$02
0xA5	$01
...	...

Registros

Reg	Contenido
R0	
R1	
R2	
R3	
R4	
R5	
R6	
...	...

Antes de terminar esta sección comentar que en el computador CS2010 existen 2 instrucciones aritméticas más, ADDI y SUBI, con 2 operandos y uno de ellos en modo inmediato. Estas instrucciones son muy útilles y su formato es `ADDI Rdestino,DatoInmediato` y `SUBI Rdestino,DatoInmediato`

- El primer operando está en modo directo de registro.
- El segundo operando es un valor inmediato.

Ejemplo 6.- Indique el resultado final de los registros tras la ejecución de los siguientes programas en ensamblador.

Programa

```
LDI R0,$02
LDI R1,$11
ADDI R0,1
ADDI R1,$AA
STOP
```

Registros

Reg	Contenido
R0	
R1	
...	...

Programa

```
LDI R0,$02
LDI R1,$11
SUBI R0,1
SUBI R1,$10
STOP
```

Registros

Reg	Contenido
R0	
R1	
...	...

3. Instrucciones de salto

Las instrucciones de salto permiten que el programa no se ejecute linealmente, son instrucciones en las que se indica la siguiente instrucción por donde continuar el programa. La instrucción de salto más simple es el salto incondicional JMP. Es una instrucción con un único argumento, que es un número que indica la instrucción por la que seguir.

Para entender esta instrucción debe saber que la memoria de programa del CS2010 almacena una instrucción por cada palabra de memoria. Una manera rudimentaria de saltar es contar las instrucciones del código ensamblador comenzando por cero, y escribir una instrucción JMP con dicho número como

operando.

Ejemplo 7.- El siguiente programa intenta realizar la operación del Ejemplo 5.- mediante un bucle, pero no termina de funcionar por que el bucle es infinito

Programa con salto incorrecto

```
LDI R0,$00
LDI R1,$01
LDI R2,$A0

LD  R3,(R2)
ADD R0,R3
ADD R2,R1

JMP 3  ; Forma incorrecta de sato

STOP
```

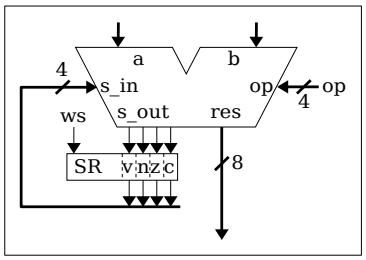
Programa con salto usando etiquetas

```
    LDI R0,$00
    LDI R1,$01
    LDI R2,$A0

BUCLE_INFINITO:

    LD  R3,(R2)
    ADD R0,R3
    ADD R2,R1

    JMP BUCLE_INFINITO
        ; Forma correcta de salto
        ; mediante etiquetas

STOP
```

En ensamblador para realizar saltos se utilizan las denominadas etiquetas. Son identificadores de texo (sin espacios) terminados por ":". Se deben poner antes de la instrucción a la que se desea saltar desde alguna otra parte del programa.

En el ejemplo anterior el salto es incondicional lo que implica que el programa ejecuta de manera ininterrumpida las 3 instrucciones que están desde la etiqueta hasta el salto. Existen otros tipos de saltos que se denominan saltos condicionales y son las instrucciones en ensamblador utilizadas para implementar las estructuras `if (condicion) {} else {}` de los lenguajes de programación como Java/C, etc. El modo de realizar en ensamblador estas sentencias es completamente diferente a la sintaxis Java/C, se requieren varias instrucciones y conocimientos sobre el funcionamiento interno del procesador.

En ensamblador para implementar este tipo de estructuras condicionales se necesitan dos pasos, en el primero se realizar una comparación para evaluar la condición y en el segundo se realiza un salto condicional. Para entenderlo hay que hacer las siguientes consideraciones sobre el funcionamiento interno del microprocesador CS2010:



- El microprocesador sólo puede usar como condición los biestables: V N Z C que son las salidas almacenadas de la última operación matemática realizada por la ALU.
- El procesador dispone de una instrucción especial parecida a JMP, pero que sólo salta bajo cierta condición. Esta instrucción ejecutará un salto si y sólo si están a un valor determinado uno o varios de los 4 biestables V N Z C

Ejemplo 8.- En el siguiente ejemplo se muestra una forma de implementar en lenguaje ensamblador una estructura `if (condicion) {} else {}`.

Programa JAVA/C

```
if (R0 == 5)
  {
    R1=1;
  }
else
  {
    R2=2;
  }
```

Programa ensamblador

```
    SUBI R0,5
    BRZS PONER_1
    LDI R2,2
    STOP
PONER_1:
    LDI R1,1
    STOP
```

Hay que plantearse la siguientes cuestiones tras ver el código ensamblador:

- ¿Cómo funciona la instrucción BRZS?
- ¿Por qué en la tabla de instrucciones no está la instrucción BRZS?
- ¿Por qué se ha realizado una resta sobre el registro R0?
- ¿Cree que R0 debería mantener si valor, ya que la resta lo cambia y en código Java/C no cambia?

Para terminar de comprender las instrucciones de salto, se ha marcado en la tabla de instrucciones mostrada a continuación la instrucción BRxx y la columna que indica, para cada instrucción, los biestables que cambian. A partir de aquí deben usarse los términos Banderas o Flags para esos 4 biestables.

Bits del código de operación					NEMÓNICO	FORMATO	TIPO	SINTAXIS	EFECTO[1]	VNZC[2]
15	14	13	12	11						
0	0	0	0	0	ST	A	memoria	ST (Rbase),Rfuente	MEM[Rbase]←Rfuente	----
0	0	0	0	1	LD	A	memoria	LD Rdestino,(Rbase)	Rdestino←MEM[Rbase]	----
0	0	0	1	0	STS	B	memoria	STS dirección,Rfuente	MEM[dirección]←Rfuente	----
0	0	0	1	1	LDS	B	memoria	LDS Rdestino,dirección	Rdestino←MEM[dirección]	----
0	0	1	0	0	CALL	C	salto	CALL dirección	MEM[SP]←PC, SP←SP-1, PC←dirección	----
0	0	1	0	1	RET	-	salto	RET	PC←MEM[SP+1], SP←SP+1	----
0	0	1	1	0	BR*xx*	C	salto	BR*xx* dirección	*xx*:PC←dirección	----
0	0	1	1	1	JMP	C	salto	JMP dirección	PC←dirección	----
0	1	0	0	0	ADD	A	aritmético/lógica	ADD Rdestino,Rfuente	Rdestino←Rdestino+Rfuente	****
0	1	0	0	1	-	-	-	-	no documentado	UUUU
0	1	0	1	0	SUB	A	aritmético/lógica	SUB Rdestino,Rfuente	Rdestino←Rdestino-Rfuente	****
0	1	0	1	1	CP	A	estado	CP Rdestino,Rfuente	NOP	****
0	1	1	0	0	-	-	-	-	no documentado	UUUU
0	1	1	0	1	-	-	-	-	no documentado	UUUU
0	1	1	1	0	-	-	-	-	no documentado	UUUU
0	1	1	1	1	MOV	A	movimiento de datos	MOV Rdestino,Rfuente	Rdestino←Rfuente	----
1	0	0	0	0	-	-	-	-	no documentado	UUUU
1	0	0	0	1	-	-	-	-	no documentado	UUUU
1	0	0	1	0	CLC	-	estado	CLC	C←0	---*
1	0	0	1	1	SEC	-	estado	SEC	C←1	---*
1	0	1	0	0	ROR	A o B	desplazamiento	ROR Rdestino	Rdestino←SHR(Rdestino,C)	****
1	0	1	0	1	ROL	A o B	desplazamiento	ROL Rdestino	Rdestino←SHL(Rdestino,C)	****
1	0	1	1	0	-	-	-	-	no documentado	UUUU
1	0	1	1	1	STOP	-	especial	STOP	lleva el procesador a espera	----
1	1	0	0	0	ADDI	B	aritmético/lógica	ADDI Rdestino,dato	Rdestino←Rdestino+dato	****
1	1	0	0	1	-	-	-	-	no documentado	UUUU
1	1	0	1	0	SUBI	B	aritmético/lógica	SUBI Rdestino,dato	Rdestino←Rdestino-dato	****
1	1	0	1	1	CPI	B	estado	CPI Rdestino,dato	NOP	****
1	1	1	0	0	-	-	-	-	no documentado	UUUU
1	1	1	0	1	-	-	-	-	no documentado	UUUU
1	1	1	1	0	-	-	-	-	no documentado	UUUU
1	1	1	1	1	LDI	B	movimiento de datos	LDI Rdestino,dato	Rdestino←dato	----

[1] (sin tener en cuenta el registro de estado y el incremento del PC)

[2] El caracter '-' denota "no modificado", '*' denota "modificado de forma definida", 'U' denota "no documentado"

Tabla 1. Conjunto de instrucciones del procesador CS2010.

Para entender la instrucción Brxx hay que utilizar también la tabla de saltos condicionales incluida con

el material de examen:

I_{10}	I_9	I_8	CONDICIÓN	MNEMÓNICO	NOTAS
0	0	0	Z	ZS,EQ	será cierta justo tras realizar la resta A-B si y solo si A=B
0	0	1	C	CS, LO	será cierta justo tras realizar la resta A-B si y solo si A<B asumiendo notación base 2 sin signo
0	1	0	V	VS	será cierta si y solo si el dato recién calculado no es representable en notación complemento a 2
0	1	1	N xor V	LT	será cierta justo tras realizar la resta A-B si y solo si A<B asumiendo notación complemento a 2
1	-	-	?	-	estas condiciones no están definidas y no deben utilizarse

Tabla 2. Instrucciones de salto condicional.

De la columna MNEMÓNICO se deducen todas las instrucciones posibles de salto condicional que son las mostradas en la siguiente tabla:

Instrucción	Equivalencia pseudocódigo
BRZS etiqueta	IF (Z) JMP etiqueta
BREQ etiqueta	IF(Z) JMP etiqueta
BRCS etiqueta	IF(C) JMP etiqueta
BRLO etiqueta	IF(C) JMP etiqueta
BRVS etiqueta	IF(V) JMP etiqueta
BRLT etiqueta	IF(N⊗V) JMP etiqueta

Tabla 3. Juego de instrucciones de salto condicional del computador CS2010.

El juego de instrucciones de salto condicional permite realizar un salto condicional según la funcionalidad descrita en la columna **NOTAS** de la tabla 2, pero sólo si previamente se han restado ambos números. Es decir, cada una de las instrucciones de la tabla corresponde a una comparación de dos valores (restados previamente) que puede realizar el procesador.

Como se ha dicho, para antes de usar una instrucción BRxx se debe ejecutar una instrucción de resta (SUB / SUBI), así, el procesador almacenará el valor los indicadores de la ALU en las banderas V N Z C. Tras esto, se puede usar una o varias instrucciones BRxx, la cual saltará sí y solo sí se cumple la condición mostrada en la segunda columna de la tabla 2.

Hay que considerar un aspecto más con los saltos condicionales, para usarlos correctamente se debe establecer si los números con los que se trabaja están en notación con signo o sin signo. Supongamos que se quiere realizar la siguiente comparación `if(R0 < R1) { … }`, antes de empezar con el código ensamblador se establece si los números con los que se operará tienen signo o no, y así, escoger la instrucción BRxx adecuada. Fíjese en los siguientes ejemplos.

Ejemplo 9.- Obtenga los programas en ensamblador equivalentes al código Java/C, donde se comparan 2 números y se ha establecido que se está trabajando con números sin signo.

Programa JAVA/C

```
if (R0 < R1)
  {
    R2=R1;
  }
else
  {
    R2=R0;
  }
```

Programa ensamblador

```
    CP   R0,R1
    BRLO MENOR_R0
    MOV R2,R0
    STOP
MENOR_R0:
    MOV R2,R1
    STOP
```

Programa JAVA/C

```
if (R0 < 5)
  {
    R1=2;
  }
else
  {
    R1=1;
  }
```

Programa ensamblador

```
    CPI   R0,5
    BRLO MENOR_5
    LDI R1,1
    STOP
MENOR_5:
    LDI R1,2
    STOP
```

En los ejemplos se han resaltado las instrucciones CP, CPI y BRLO para discutir sobre lo siguiente:

- ¿Cuál es la función de las instrucción CP /CPI?
- ¿No había que realizar una resta antes de la comparación?
- ¿Por qué se usa la instrucción BRLO y no la instrucción BRLT?

Respecto a las instrucciones CP y CPI, ambas son parecidas a SUB y SUBI respectivamente. Estas instrucciones realizan una resta de los operandos pero no guardan el resultado de la resta en el registro del operando izquierdo. La finalidad de estas instrucciones es realizar comparaciones números previo a un salto condicional, de hecho, CP hace referencia a *Compare* y CPI a *Compare Inmediate*. Con estas instrucciones se consigue no destruir el valor de uno de los números comparados, y que el código sea equivalente a los mostrados en el los ejemplos en Java/C.

La segunda consideración sobre la instrucción de salto condicional es la existencia de instrucciones de comparación en función de, si la aritmética es con signo o sin signo. Como ya se estudió en el primer cuatrimestre en CED trabajar en notación complemento a 2 o en binario no afecta a las operaciones que debe realizar la ALU, es decir, la suma y la resta (ADD y SUB) son válidas e idénticas en ambas notaciones. Esto quiere decir que, es el programador cuando usa un dato quien decide si es con signo o no, y vale tanto para dato inmediatos, registros o palabras de memoria. Aunque ya se indicó anteriormente, se vuelve recuerda que es recomendable que, una vez decidido si los números son con signo o no, no se cambie de criterio a lo largo del programa ya que habría que ser cuidadoso para no cometer errores.

Hechos estos comentarios, fíjese que en el enunciado del Ejemplo 9.- se ha establecido que se trabajará con números con signo. Sin esta indicación no es posible saber que instrucción de salto condicional usar tras comparar el valor de 2 números. Para no equivocarse, recuerde que en la última columna de la tabla de saltos condicionales (Tabla 2, pág.10) está descrita la condición bajo la que se produce el salto y la notación (signo o sin signo). Las descripciones para BRLO y BRLT de esa tabla son:

- BRLO: será cierta justo tras realizar la resta A-B sí y solo sí A<B asumiendo notación base 2 sin signo.
- BRLT: será cierta justo tras realizar la resta A-B sí y solo sí A<B asumiendo notación complemento a 2.

Puesto que en el enunciado se indicaba que los números se trataban en el programa como números sin signo, el salto correcto para comparar R0<R1 y R0<5 es BRLO.

Esta solución es sencilla pero según la condición a evaluar se puede complicar. En ejemplo cuando se

realiza la comparación *menor que*, existen instrucciones en la Tabla 2 (pág.10) para realizar la comparación solicitada. Pero esto no siempre es así, ya que hay otras comparaciones con “>”, “>=” o “<=” sin equivalencia en la Tabla 2. En los siguientes ejemplos se resolverán estas situaciones.

Ejemplo 10.- Obtenga el programa ensamblador equivalente al fragmento de código Java/C de la derecha donde se ha establecido que se trabajará con números con signo.

Programa JAVA/C

```
if (R0 >= R1)
  {
    R2=R1;
  }
else
  {
    R2=R0;
  }
```

Programa equivalente

```
if (R0 < R1)
  {
    R2=R0;
  }
else
  {
    R2=R1;
  }
```

Solución ensamblador

```
    CP   R0,R1
    BRLT MENOR_R0
    MOV R2,R1
    STOP
MENOR_R0:
    MOV R2,R0
    STOP
```

La solución ha consistido en obtener un programa equivalente usando otra comparación. Aunque existen más soluciones, la mostrada ha cambiado el operador “>=” por “<”. El operador “<” sí existe en la tabla de comparaciones (Tabla 2, pág.10) y corresponde a la instrucción BRLO, ya que se trabaja con números con signo. Pero tras cambiar el operador, también ha sido necesario realizar los cambios indicados en rojo en el programa para que sea equivalente. Con el programa equivalente se obtiene el ensamblador directamente.

Ejemplo 11.- Trabajando con números sin signo obtenga el programa ensamblador equivalente al código mostrado.

Programa JAVA/C

```
if (R0 > 5)
  {
    R2=1;
  }
else
  {
    R2=0;
  }
```

Programa equivalente

```
if(R0 < 5)
    R2=0;
else if(R0 == 5)
    R2=0;
else
    R2=1;
```

Solución ensamblador

```
    CPI  R0,5
    BRLO PONER_1
    LDI  R2,0
    BREQ PONER_1
    STOP
PONER_1:
    LDI R2,1
    STOP
```

En la solución propuesta se ha optado de nuevo por adaptar el programa original para poder pasarlo a ensamblador de forma más fácil. Se ha cambiado la comparación inicial por 2 comparaciones que sí son posibles con el juego de instrucciones CS2010. Además se ha optimizado un poco el código de hecho cabe preguntarse:

- En la segunda comparación, ¿por qué tras la instrucción LDI R2,0 no ha sido necesario realizar otra vez CPI R0,5?.

Para entenderlo en importante conocer el funcionamiento interno (a nivel de miro-operaciones) de la ejecución de cada una de las instrucciones del CS2010. Por ello, en el juego de instrucciones del CS2010 mostrado en la Tabla 1(pág. 9) se ha resaltado la última columna que corresponde a las banderas de la ALU explicadas anteriormente. Esa columna contiene 5 indicaciones para cada instrucción de la tabla. Los indicadores hacen referencia a las banderas V N Z C. Como ejemplo se muestran sólo las instrucciones CP , LDI y la nota inferior que hace referencia a las banderas:

Bits del código de operación					NEMÓNICO	FORMATO	TIPO	SINTAXIS	EFECTO[1]	VNZC[2]
15	14	13	12	11						
0	1	0	1	1	CP	A	estado	CP Rdestino,Rfuente	NOP	****
1	1	1	1	1	LDI	B	movimiento de datos	LDI Rdestino,dato	Rdestino←dato	----

[1] (sin tener en cuenta el registro de estado y el incremento del PC)

[2] El caracter '-' denota “no modificado”, '*' denota “modificado de forma definida”, 'U' denota “no documentado”

El texto inferior resaltado hace referencia al efecto que tiene cada instrucción sobre las banderas V N Z C. En el caso de las instrucciones CP y LDI son:

- La instrucción CP tiene en la columna la indicación “* * * *”, cada “*” hace referencia a una bandera y por ser “*” el significado es que las 4 banderas cambian tras la ejecución.
- La instrucción LDI tiene en la columna la indicación “- - - - -”, cada “-” hace referencia a una bandera y por ser “-” denota que la bandera no es modificada.

En resumen se puede decir, la instrucción CP resta 2 números guardando en las banderas las salidas V, N, Z, C de la ALU. En cambio la instrucción LDI carga un valor inmediato en un registro pero no altera ninguna bandera. Si busca la instrucción BRxx en la tabla de instrucciones verá que tampoco modifica ninguna bandera del procesador, por eso en el Ejemplo 11.- con una única instrucción CPI se puede resolver.

4. Realización de bucles

Los bucles son estructuras muy comunes en todos los lenguajes de programación de alto nivel. En ensamblador, las instrucciones de salto condicional son la base para realizar cualquier tipo de bucle como puede ser el bucle `while(condicion) {…}` del lenguaje Java/C.

Ejemplo 12.- Realice el bucle indicado en el programa Java/C, en ensamblador del CS2010.

Programa JAVA/C

```
int R0,R1;

R0 = 10;
R1 = 0;
while( R0 != 0 )
{
    R1 = R1 + 5
    R0 = R0 - 1;
}
```

Programa ensamblador

```
    LDI  R0,10
    LDI  R1,0

BUCLE:
    CPI  R0,0
    BREQ FIN
    ADDI R1,5
    SUBI R0,1
    JMP BUCLE
FIN:
    STOP
```

En el ejemplo anterior se han resaltado en rojo las instrucciones correspondientes al bucle quedando 2 instrucciones en el cuerpo del mismo. En caso de bucles tipo FOR la solución es parecida

Ejemplo 13.- Realizar el bucle indicado en el programa Java/C, en ensamblador.

Programa JAVA/C

```
int R0,R1;
R1=25;

for(R0=1;R0==5;R0++)
{
    R1 = R1 + 7
}
```

Programa ensamblador

```
    LDI  R1,25
    LDI  R0,1
BUCLE:
    CPI  R0,5
    BREQ FIN
    ADDI R1,7
    ADDI R0,1
    JMP BUCLE
FIN:
    STOP
```

La única consideración en este caso es saber exactamente como operan los bucles tipo FOR que es `for(iniciación ; condición ; sentencia final)`. La sentencia inicial no entra en el cuerpo del bucle, la condición se comprueba antes de entrar en cada vuelta de bucle y la sentencia final debe ejecutarse al final del cuerpo del bucle, es decir, antes de saltar hacia atrás.

5. Vectores / Arrays

Las instrucciones con modo de direccionamiento indirecto a memoria se utilizan principalmente para trabajar con vectores (denominados también arrays). Se debe considerar que en lenguaje ensamblador el único vector existente es la totalidad de la memoria RAM disponible. En el procesador CS2010 esta memoria es una memoria RAM de 256 palabras [0-255] de 8 bits.

El procedimiento para poder usar varios vectores en un programa consiste en subdividir la memoria RAM en fragmentos mas pequeños evitando que se solapen. Así, para realizar operaciones con vectores se utilizan bucles con las instrucciones LD y ST usando registros como índice del vector.

Ejemplo 14.- Realizar un programa que escriba en un vector 10 números pares comenzando desde el valor 8 y, estableciendo que el vector comienza en la dirección $50 de memoria.

Programa JAVA/C

```
int RAM[] = new int[256];
int R0 = 8;
int R1 = 0x50;
int R2 = 0;
while(R2 != 10)
{
  RAM[R1] = R0;
  R0 = R0+2;
  R1 = R1+1;
  R2 = R2+1;
}
```

Programa ensamblador

```
    LDI  R0,8
    LDI  R1,0x50
    LDI  R2,0

BUCLE:
    CPI  R1,10
    BREQ FIN
    ST   (R1),R0
    ADDI R0,2
    ADDI R1,1
    ADDI R2,1
    JMP BUCLE
FIN:
    STOP
```

En la solución propuesta se ha utilizado como índice para recorrer el vector el registro R1 y se han marcado en rojo las instrucciones ensamblador encargadas de recorrer el vector. Considere que este programa se podría haber realizado con un registro menos, sin usar R2, intente obtener esa solución.

Ejemplo 15.- Realizar un programa sume 2 vectores de 10 números. El primer vector comenzará $10, el segundo en $20 y el vector resultante debe escribirlo a partir de $30.

Programa JAVA/C

```
int RAM[]=new int[256];
int R0=0x10;
int R1=0x20;
int R2=0x30;
int R3=10;
while(R3 != 0)
{
  RAM[R3] = RAM[R0] + RAM[R1];
  R0 = R0 + 1;
  R1 = R1 + 1;
  R2 = R2 + 1;
  R3 = R3 - 1;
}
```

Programa ensamblador

```
    LDI  R0,0x10
    LDI  R1,0x20
    LDI  R2,0x30
    LDI  R3,10
BUCLE:
    LD   R4,(R0)
    LD   R5,(R1)
    ADD  R4,R5
    ST   (R3),R4
    ADDI R0,1
    ADDI R1,1
    ADDI R2,1
    SUBI R3,1
    BRZS FIN
    JMP  BUCLE
FIN:
    STOP
```

El programa anterior se podría optimizar no usando el registro R3 como contador y haciendo otra comparación en la condición del bucle. Intente obtener esa solución.

6. Subrutinas

En los lenguajes de programación alto nivel existen estructuras de código llamadas funciones (métodos en los lenguajes de programación orientada a objetos). La finalidad de las funciones es reutilizar fragmentos de códigos y conseguir programas más compactos y legibles. En lenguaje ensamblador estas estructuras se suelen llamar subrutinas.

En ensamblador se utilizan dos instrucciones CALL y RET para implementar subrutinas. La primera (CALL) sirve para realizar la llamada una subrutina y la segunda (RET) para retornar cuando ha terminado. RET sólo puede ser usada dentro de una subrutina para retornar de la misma y la vuelta es a la siguiente instrucción desde la que fue llamada con la instrucción CALL.

Ejemplo 16.- La función escrita en JAVA/C devuelve la suma de los n primeros números indicados como parámetro. Por ejemplo suma_n_numeros(7) devuelve: 7+6+5+4+3+2+1. Obtenga una subrutina equivalente en lenguaje ensamblador. Indique cuales son los registros que se usarán como parámetros de entrada y de salida.

Método/Función en JAVA/C

```
int suma_n_numeros(int n)
{
    int R0=n,R1=0;
    while(R0 != 0)
    {
       R1 = R1 + R0;
       R0 = R0 - 1;
    }
    return R1;
}
```

En primer lugar, para realizar una subrutina en ensamblador es necesario considerar las principales diferencias frente a funciones/métodos de lenguajes de alto nivel. Se pueden resumir como:

- Las subrutinas en ensamblador no admiten parámetros de entrada. Hay que utilizar algún mecanismo, como por ejemplo, reservar algunos registros para uso exclusivo en las subrutinas.
- Las subrutinas en ensamblador no pueden devolver valores. Hay que utilizar algún mecanismo, como por ejemplo, indicar que registro o palabras de memoria devuelve el valor o valores.

Vistas las consideraciones anteriores, la función en Java/C propuesta se puede reescribir teniendo en cuenta las restricciones expuestas. Así, considerando que existen sólo variables globales (registros R0 a R7), se debe decidir cuales de ellos se utilizarán como parámetro de entrada y cual como de salida.

Método/Función en JAVA/C

```
// R6 el parámetro de entrada 'n'
// En R7 estará el valor a devolver
void suma_n_numeros()
{
    R7 = 0;
    while(R6 != 0)
    {
       R7 = R7 + R6;
       R6 = R6 - 1;
    }
    return;
}
```

Subrutina en ensamblador

```
;R6 el parámetro de entrada 'n'
;En R7 estará el valor a devolver

suma_n_numeros:
    LDI  R7,0
    CPI  R6,0
bucle_interno:
    BRZS fin_suma_n_numeros
    ADD  R7,R6
    SUBI R6,1
    JMP  bucle_interno
fin_suma_n_numeros:
    RET
```

Dada la solución propuesta se plantean las siguientes cuestiones:

- ¿Por qué el salto "bucle_interno" está después de la comparación con cero (CPI R6,0)? ¿No es necesario realizar la comparación con cero en cada iteración del bucle?
- ¿Donde está la instrucción de STOP del fin de programa?
- ¿Cómo se utiliza/llama esta subrutina desde un programa que la use?

La respuesta a la primera cuestión consiste en fijarse en la tabla de instrucciones la instrucción JMP no cambia las banderas. Justo antes de JMP se restó 1 a R6 y si esta resta resultó cero, en bandera Z queda cuardado un 1. Cuando se salta a la etiqueta bucle_interno y se ejecuta BRZS se comprueba si la bandera Z es 1, sólo puede ser 1 si la última resta resultó cero.

La segunda cuestión es un error habitual cuando no se ha comprendido el objetivo de la subrutinas. La instrucción STOP es para parar el procesador, en cambio, una subrutina es un fragmento de código reutilizable, Las subrutinas serán llamadas desde varios lugares del programa principal y siempre debe terminar volviendo al programa que lo llamó mediante RET.

Para realizar la llamada a una subrutina antes de usar la instrucción CALL es necesario poner en los registros usados como parámetros de entrada el valor. En el siguiente ejemplo se muestra como realizar llamada.

Ejemplo 17.- En el ejemplo anterior se ha desarrollado una subrutina en ensamblador llamada `suma_n_numeros` que recibe en R6 un parámetro de entrada y devuelve en R7 el calculo realizado. Realice un programa que llame a la subrutina con el valor 3 y almacene el resultado en la dirección $10 de memoria.

Ejemplo JAVA/C

```
int RAM[]=new int[256];
int R0,R1,R2,R3,R4,R5,R6,R7;

// Definición del método/función anterior
int suma_n_numeros(int n);

// Programa
void main(void)
{
  RAM[0x10] = suma_n_numeros(3);
}
```

Programa en ensamblador

```
LDI  R6,3
CALL suma_n_numeros
STS  0x10,R7
STOP
```

7. Obtención del código máquina

Otro aspecto que se debe manejar en esta asignatura es la obtención del código máquina a partir del código ensamblador y viceversa. Esta tarea es realizada por programas informáticos, y es un software poco complejo de desarrollar. La obtención del código máquina para este procesador (CS2010) es muy simple por las siguientes características:

- Cada instrucción ocupa una única palabra de la memoria de código del procesador, y es de 16bits.
- La memoria de código tiene una capacidad de 256 palabras, lo que equivale a un programa como máximo de 256 instrucciones.
- La primera instrucción que se ejecuta al iniciar el procesador es la instrucción de la palabra de memoria de código $00.

El procedimiento consiste obtener para la codificación binaria o hexadecimal de cada instrucción ensamblador utilizando la tabla de formato de instrucción del CS2010 y la tabla de instrucciones de CS2010, ambas disponibles en la hoja de examen. En la tabla de formato de instrucción se indican los 16 bits de cada palabra de memoria de código en las columnas y se agrupan en campos mostrados en las filas, como son *código de operación*, *registro destino*, *dato inmediato*, etc.

<table>
<tr><th><u>formato</u></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>A
instrucción con operando registro</td><td colspan="5" rowspan="3">código de operación</td><td colspan="3" rowspan="2">registro destino (fuente en ST)</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td colspan="3">registro fuente (registro base en ST/LD)</td></tr>
<tr><td>B
instrucción con operando memoria o inmediato</td><td colspan="8">dato inmediato / dirección del dato</td></tr>
<tr><td>C
instrucción de salto</td><td colspan="3">condición de salto</td><td colspan="8">dirección de salto</td></tr>
</table>

Tabla 4. Tabla de formato de instrucción del CS2010.

Para todas las instrucciones, los bits 5 bits más significativos (15,14,13,12,11) corresponden al campo código de operación, el cual, es único para cada instrucción. Estos bits son fáciles de obtener, sólo hay que buscar en la tabla de instrucciones del procesador la instrucción y copiar las 5 primeras columnas, que ya están expresadas en binario.

Para el resto de campos hay que deducir los que hay que utilizar según sea una instrucción y otra. Por

eso en la Tabla 4 hay 3 filas con 4 posibles formatos: A, B y C. En un primer ejemplo se va ha suponer que ya se ha obtenido formato de la instrucción LDI siendo exactamente el siguiente:

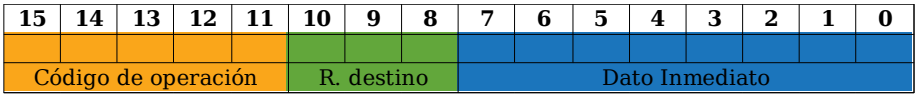


Figura 1. Formato de para la instrucción LDI (Formato tipo B).

Esta figura indica que en una instrucción LDI los 16 bits se rellenan usando sus 5 bit de código de operación, después el número en binario del registro destino codificado con 3 bits (el primer operando de LDI). En los 8 bits restantes, indicados en azul, irá el valor inmediato que aparece en el segundo operando de LDI, en binario.

Ejemplo 18.- Conociendo el formato de la instrucción LDI y dado el programa ensamblador obtenga el código máquina. Debe rellenar la memoria de código correctamente, ya que es donde reside el programa.

Programa

```
LDI R1,23
LDI R2,$23
LDI R3,0x23
STOP
```

Código máquina

DIR	Contenido
00	11111 001 00010111
01	11111 010 00100111
02	11111 011 00100111
03	10111 000 00000000
...	...

Bits del código de operación					SINTAXIS
15	14	13	12	11	
0	0	0	0	0	ST (Rbase),Rfuente
0	0	0	0	1	LD Rdestino,(Rbase)
0	0	0	1	0	STS dirección,Rfuente
0	0	0	1	1	LDS Rdestino,dirección
0	0	1	0	0	CALL dirección
0	0	1	0	1	RET
0	0	1	1	0	BR*xx* dirección
0	0	1	1	1	JMP dirección
0	1	0	0	0	ADD Rdestino,Rfuente
0	1	0	0	1	-
0	1	0	1	0	SUB Rdestino,Rfuente
0	1	0	1	1	CP Rdestino,Rfuente
0	1	1	0	0	-
0	1	1	0	1	-
0	1	1	1	0	-
0	1	1	1	1	MOV Rdestino,Rfuente
1	0	0	0	0	-
1	0	0	0	1	-
1	0	0	1	0	CLC
1	0	0	1	1	SEC
1	0	1	0	0	ROR Rdestino
1	0	1	0	1	ROL Rdestino
1	0	1	1	0	-
1	0	1	1	1	STOP
1	1	0	0	0	ADDI Rdestino,dato
1	1	0	0	1	-
1	1	0	1	0	SUBI Rdestino,dato
1	1	0	1	1	CPI Rdestino,dato
1	1	1	0	0	-
1	1	1	0	1	-
1	1	1	1	0	-
1	1	1	1	1	LDI Rdestino,dato

En la solución se han respetado los colores usados en la figura 1 para que pueda observar donde va cada campo. Además ha sido necesario usar las primeras columnas de la tabla de instrucciones para obtener los 5 bits de código de operación. La cuarta instrucción (STOP) almacenada en la dirección 3 de la memoria de programa tiene el código de operación 10111 según la tabla pero el resto de campos se han puesto a cero. Se plantean las siguientes cuestiones:

- ¿Cuantos argumentos tiene la instrucción STOP?
- ¿Los campos no usados se rellenan con cero?

No todas las instrucciones del CS2010 utilizan los 16 bits de código de operación, por eso, la instrucción STOP se codifica únicamente con 5 bits del código de operación ya que es una instrucción sin argumentos. El resto de bits no utilizados, en el ejemplo se han puesto a cero, pero podrían contener cualquier valor ya que no se usan.

Ejemplo 19.- Obtenga el código máquina del programa correspondiente al primer ejemplo de salto usando anteriormente.

Programa

```
    LDI R0,$00
    LDI R1,$01
    LDI R2,$A0

BUCLE_INFINITO:

    LD  R3,(R2)
    ADD R0,R3
    ADD R2,R1

    JMP BUCLE_INFINITO
    STOP
```

Código máquina

DIR	Contenido
00	11111 000 00000000
01	11111 001 00000001
02	11111 010 00000010
03	00001 011 00000010
04	01000 000 00000011
05	01000 010 00000001
06	00111 000 ?????????
07	10111 000 00000000

En este ejemplo aparecen 3 nuevas instrucciones ADD, LD y JMP para las cuales hay que deducir el formato correspondiente. Se debe deducir usando la columna formato de la tabla de instrucciones y la tabla de formato de instrucción. En la figura se ha resaltado la columna formato que puede ser A, B o C para cada instrucción. Con esta información y dados los nombres de los operandos de la instrucción (registro destino / registro fuente) hay que deducir el formato de cada instrucción, fíjese en los ejemplos:

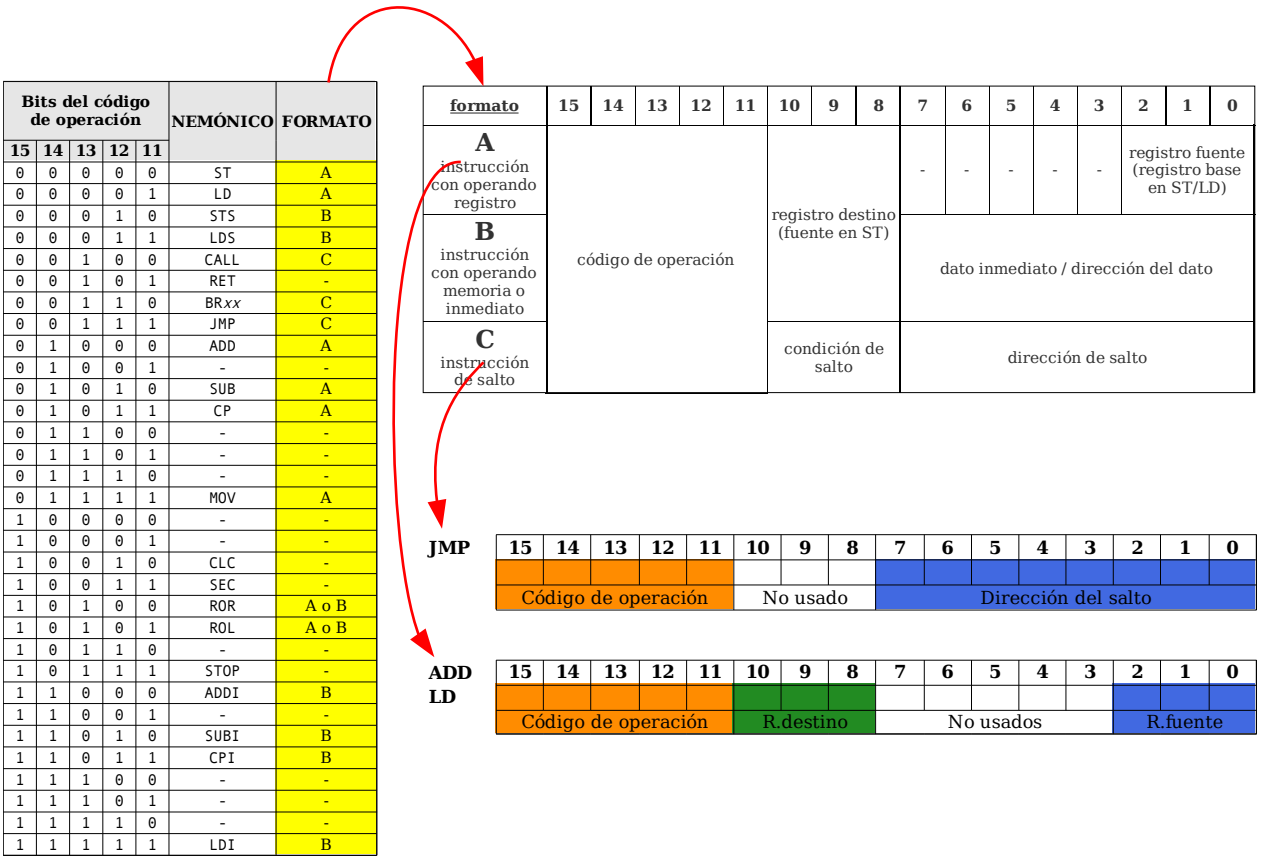


Figura 2. Formato de las instrucciones ADD, LD y JMP.

De nuevo en la solución se han respetado los colores y puede observar como aparecen bits en estas instrucciones no usados. De nuevo, los bits no usados se han establecido a cero, pero podrían tener cualquier valor. Las instrucciones como ADD y LD tienen 2 operandos y ambos son registros. Es un error

habitual no situar cada operando en el lugar adecuado. Cuando codifique en binario cada uno de los registros y no debe equivocarse en la ubicación los campos *R.destino* y *R.fuente*. Con la instrucción ST la ubicación de estos campos se invierten y puede crear confusión.

Por otro lado, la solución está incompleta. En la dirección $06 de la memoria de código hay un campo con interrogantes. Se plantean las siguientes cuestiones:

- ¿A qué instrucción ensamblador corresponde la instrucción de la dirección $06 con 8 interrogantes?
- ¿Cuál es el valor que hay que establecer en el campo relleno con interrogantes?

Si se cuentan las instrucciones en ensamblador del programa empezando desde cero, la instrucción $06 es la instrucción `JMP BUCLE_INFINITO`. Aquí la dificultad es codificar el único operando de esta instrucción que es BUCLE_INFINITO. Según el formato de la instrucción JMP, este valor corresponde al campo *Dirección del salto* y el cual contendrá el valor numérico de la instrucción a la que debe saltar (en binario).

Según el programa ensamblador `JMP BUCLE_INFINITO` debe saltar a la instrucción `LD R3,(R2)`, y esta instrucción está en la posición $03 de la memoria de código, o lo que es lo mismo, la instrucción 3 si se cuentan las instrucciones ensamblador empezando por cero. Así, esta instrucción se codifica con la secuencia binaria: 00111 000 00000011.

Ejemplo 20.- Obtenga el código máquina para el programa ensamblador mostrado. Este programa realiza un bucle mediante la instrucción de salto condicional BREQ.

Programa ensamblador

```
    LDI  R1,25
    LDI  R0,1
BUCLE:
    CPI  R0,5
    BREQ FIN
    ADDI R1,7
    ADDI R0,1
    JMP BUCLE
FIN:
    STOP
```

Código máquina

DIR	Contenido
00	11111 001 00011001
01	11111 000 00000001
02	11011 000 00000101
03	00110 ??? 00000111
04	11000 001 00000111
05	11000 000 00000001
06	00111 000 00000010
07	10111 000 00000000

En este ejemplo aparecen las instrucciones instrucciones CPI, ADDI y BREQ, para las que hay que deducir su formato. Las dos primeras tienen el mismo formato que LDI (formato tipo B), el cual fue mostrado en la figura 1 (pág. 18). En cambio las instrucciones BRxx tienen un formato tipo C, al igual que JMP pero con una diferencia que se explica a continuación.

Además, la solución propuesta la tiene incompleta 3 bits de la instrucción $03. Esta instrucción es la correspondiente `BREQ FIN` del programa ensamblador. Se plantean las siguientes cuestiones:

- ¿A qué campo del formato de instrucción corresponden los 3 bits con interrogante?
- ¿Cuál es el valor de los 3 bits de la instrucción $03?

Para deducir el formato de la instrucción BRxx se vuelven a usar las tablas de la figura 2 (pág 19) en donde se indica que es tipo C (igual que JMP), pero en este caso, el salto es condicional y aparece un

campo adicional de 3 bis llamado condición. Este formato se muestra a continuación:

Brxx	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Código de operación					Condición			Dirección del salto							

0	0	0	**BRZS / BREQ**
0	0	1	**BRCS / BRLO**
0	1	0	**BRVS**
0	1	1	**BRLT**

El campo de condición admite 8 combinaciones, pero no todas se utilizan. En el formato de instrucción se han añadido en verde los 4 posibles valores para este campo y la instrucción a la que hace referencia. Estas instrucciones provenían de la tabla de saltos condicionales y se deben deducir de la tabla de saltos condicionales del procesador. En las 3 primeras columnas de la tabla de saltos condicionales están los 3 bits $I_{10,}$ I_9 e I_8. Fíjese en la correspondencia entre los valores de los 3 bits de la tabla mostrada a continuación y el campo de condición de la instrucción BRxx:

I_{10}	I_9	I_8	CONDICIÓN	MNEMÓNICO	NOTAS
0	0	0	Z	ZS,EQ	será cierta justo tras realizar la resta A-B si y solo si A=B
0	0	1	C	CS, LO	será cierta justo tras realizar la resta A-B si y solo si A<B asumiendo notación base 2 sin signo
0	1	0	V	VS	será cierta si y solo si el dato recién calculado no es representable en notación complemento a 2
0	1	1	N xor V	LT	será cierta justo tras realizar la resta A-B si y solo si A<B asumiendo notación complemento a 2
1	-	-	?	-	estas condiciones no están definidas y no deben utilizarse

Tras la explicación, el código máquina de la instrucción `BREQ FIN` es 00110 000 00000111. En el siguiente ejemplo aparecen 2 saltos condicionales diferentes, intente resolverlo para ver si lo ha comprendido.

Ejemplo 21.- Rellene la memoria de código de programa correspondiente al programa ensamblador de unos de los ejemplos anteriores (Ejemplo 11.- pág 12).

Programa ensamblador

```
    CPI   R0,5
    BRLO PONER_1
    LDI  R2,0
    BREQ PONER_1
    STOP
PONER_1:
    LDI R2,1
    STOP
```

Código máquina

DIR	Contenido
00	
01	
02	
03	
04	
05	
06	

Ejemplo 22.- En este último ejemplo se suele cometer un error muy habitual. Consiste en no codificar correctamente los dos operandos de la instrucción ST resaltada en rojo. Intente obtener el código máquina y compruebe si es capaz de codificar correctamente esta instrucción

Programa ensamblador

```
    LDI  R0,8
    LDI  R1,0x50
    LDI  R2,0

BUCLE:
    CPI  R1,10
    BREQ FIN
    ST   (R1),R0
    ADDI R0,2
    ADDI R1,1
    JMP BUCLE
FIN:
    STOP
```

Código máquina

DIR	Contenido
00	
01	
02	
03	
04	
05	
06	
07	
08	
09	

8. Otras instrucciones

En este manual no se tratan las instrucciones ROR, ROL, CLC y SEC. Tras estudiar este manual la información sobre estas instrucciones será fácil de entender usando el resto de material de la asignatura.