8. Diseño de la codificación binaria del repertorio de Instrucciones.

Se desea elegir los códigos binarios de las instrucciones. Esta parte, en forma indirecta especifica lo que debe realizar un programa ensamblador, para traducir a código numérico de máquina las instrucciones simbólicas de máquina o assembler.

Este proceso también sirve de especificación para diseñar el procesador, ya que una vez que se lea la instrucción desde la memoria, de acuerdo a los formatos, debe decodificarse los campos; información que a su vez debe generar secuencias internas para realizar las transferencias que implementan las operaciones necesarias sobre los datos.

8.2. Especificaciones:

a) Largo de palabra de 32 bits.
Esto debido a que no se desea tener instrucciones que estén codificadas en más de una palabra. Lo cual se establece para no tener dos o más lecturas de memoria por instrucción, lo que implica una ejecución más lenta; debido a que los accesos a memoria son más lentos que las operaciones en el procesador.
Además como se verá más adelante, esto frena la posibilidad de disponer de una segmentación adecuada, ya que para ejecutar la instrucción siguiente a una "larga", el procesador debe esperar a que la primera haya entrado en una etapa de decodificación.

b) Se desea disponer de 32 registros. Entonces la especificación de un registro requiere 5 bits. Un número elevado de registros es razonable en repertorios carga-almacenamiento, en donde las operaciones se efectúan entre registros solamente.

c) Se desea tener instrucciones con direccionamiento inmediato. Es decir que en el mismo código de la instrucción se tenga el valor de una constante. Por razones estadísticas, vistas anteriormente, se desea tener valores inmediatos de 16 bits, y las siguientes 8 instrucciones: addi, addiu, slti, sltiu, andi, ori, xori, lui .

Una posibilidad es direccionamiento inmediato con resultado en un registro, esto implica 5 bits para el registro del operando y 16 para el inmediato (lo cual da 21 bits, entonces hay cierta holgura, ya que la instrucción es de 32 bits). Para hacer aún más flexible el direccionamiento se decide disponer de otro registro, para depositar el resultado. Es decir, la forma general de las operaciones inmediatas es:

Rt = Rs opi inmediato_16

Donde opi es el código de la operación.
Entonces, este tipo de instrucciones ocupará 26 bits en campos de operandos. Dejando sólo 6 disponibles de los 32 de una palabra.

d) Para una segmentación eficiente se decide que las operaciones aritméticas y lógicas sean en base a registros, y no con la memoria. Esto demanda 15 bits: dos registros con los operandos y un tercer registro para depositar el resultado.
Por razones estadísticas se requieren: add, addu, sub, subu, and, or, xor, nor, slt, sltu, mult, multu, div, divu.

Las multiplicaciones y divisiones entregan resultados en 64 bits. Se dispone de dos registros (Hi y Lo) para este propósito. Y se requieren 4 instrucciones adicionales para mover los resultados a los registros del procesador y viceversa.
Entonces el tipo de direccionamiento es:

Rd = Rs op Rt

Lo cual requiere 15 bits para especificar los tres operandos. Si además se incorporan, en este formato, las 6 instrucciones de corrimiento, se requieren 5 bits adicionales para especificar, en forma inmediata el número de posiciones de corrimiento. Esto deja libres 12 bits, de los 32 de una palabra.
En total: 14+4+6 = 24 instrucciones de este tipo.

e) Se especifica además que las transferencias entre memoria y registros puedan manipular bytes, medias palabras y palabras completas.

El direccionamiento de memoria debe poder manejar arreglos y estructuras. Se determina hacerlo, mediante un direccionamiento base más un offset. El offset por razones estadísticas debe ser de 16 bits. En total se requieren 12 instrucciones de este tipo.
Entonces el tipo de direccionamiento es:

Rs= M[Rt+offset_16]

Entonces, este tipo de instrucciones ocupará 26 bits en campos de operandos. Dejando 6 bits disponibles de los 32 de una palabra. Se tiene 7 instrucciones de carga y 5 de almacenamiento, en total 12 instrucciones de transferencias de datos.

f) Se desea disponer de instrucciones de punto flotante. Se requieren 26 instrucciones de este tipo, como se verá más adelante.

g) Se dispone de trece instrucciones de salto y bifurcación.
Se tienen 9 bifurcaciones y 4 saltos.

h) También se requieren instrucciones para manejar los coprocesadores (cerca de 10 instrucciones)

En resumen, se requieren: 13 de bifurcación y salto,
26 de punto flotante,
12 de transferencias,
24 aritmético lógicas y corrimiento,

8 inmediatas, y
10 (aproximadamente) de control de procesadores.

En total 93 instrucciones.

En total, más de 64 y menos de 128. Esto implica que con un código de operación de 7 bits, podría darse un código de operación único a cada instrucción. Pero las del grupo c) y e) permiten sólo 6 bits para el código, ya que requieren 26 bits para especificar operandos.

Se decidió en el diseño usar 6 bits para el código de operación. Y se emplea el método de expandir el código de operación con campos que queden disponibles. Ver la página A-54 en el apéndice del texto guía.

Por ejemplo se emplea el código 000000 para las instrucciones aritmético lógicas. Como éstas requieren 15 bits, para especificar los operandos de entrada y salida, más 6 del código de operación, quedan 11 bits disponibles. Se emplean 6 para expandir el código (lo cual permite 64 instrucciones de este tipo) y aún quedan 5 bits. Estos últimos se pueden utilizar como campo inmediato en instrucciones de corrimiento. Este esquema define el formato de tipo R:

8.3 Formato R.

Código	**R**eg. Fuente 1	**R**eg. Fuente 2	**R**eg. Destino	Inmediato sht	Funct
6	Rs 5	Rt 5	Rd 5	5	6

Se muestra una lista (no completa) de las instrucciones que emplean el formato R. Puede verse, que para el siguiente grupo, que el Opcode es siempre 000000 y el resto del código de operación se encuentra en funct. Se han empleado los registros $t1, $t2 y $t3 que tienen códigos binarios: 01001, 01010, 01011 respectivamente, para codificar los campos binarios de las instrucciones.

Las siguientes con código de operación 000000, se diferencian con el campo funct.

Opcode	rs	rt	rd	shamt	funct	Tipo	nemónico	campos	Obs.
000000	00000	01010	01001	11111	000000	RI	sll	$t1, $t2, 31	5 bits inm en shamt
000000	00000	01010	01001	00000	000010	RI	srl	$t1, $t2, 00	shift cero
000000	00000	01010	01001	11111	000011	RI	sra	$t1, $t2, 31	shift de 31
000000	01011	01010	01001	00000	000100	R	sllv	$t1, $t2, $t3	shift en registro
000000	01011	01010	01001	00000	000110	R	srlv	$t1, $t2, $t3	shift en registro
000000	01011	01010	01001	00000	000111	R	srav	$t1, $t2, $t3	shift en registro
000000	01001	00000	00000	00000	001000	R	jr	$t1	salto relativo reg.
000000	01001	00000	11111	00000	001001	R	jalr	$t1	salto y enlace reg.
000000	00000	00000	00000	00000	001100	R	syscall		Control Procesador
000000	00000	00000	00000	00000	001101	R	break	0	Control Procesador
000000	00000	00000	01001	00000	010000	R	mfhi	$t1	Resultado Mult.Div.
000000	00000	00000	01001	00000	010010	R	mflo	$t1	Resultado Mult.Div.
000000	01001	00000	00000	00000	010001	R	mthi	$t1	Resultado Mult.Div.
000000	01001	00000	00000	00000	010011	R	mtlo	$t1	Resultado Mult.Div.
000000	01010	01011	00000	00000	011000	R	mult	$t2, $t3	Multiplicación.
000000	01010	01011	00000	00000	011001	R	multu	$t2, $t3	Multiplic.Unsigned.

000000	01010	01011	00000	00000	011010	R	div	$t2, $t3	División.
000000	01010	01011	00000	00000	011011	R	divu	$t2, $t3	División Unsigned.
000000	01010	01011	01001	00000	100000	R	add	$t1, $t2, $t3	Suma con signo.
000000	01010	01011	01001	00000	100001	R	addu	$t1, $t2, $t3	Suma sin signo.
000000	01010	01011	01001	00000	100010	R	sub	$t1, $t2, $t3	Resta con signo.
000000	01010	01011	01001	00000	100011	R	subu	$t1, $t2, $t3	Resta sin signo.
000000	01010	01011	01001	00000	100100	R	and	$t1, $t2, $t3	Lógicas.
000000	01010	01011	01001	00000	100101	R	or	$t1, $t2, $t3	Lógicas.
000000	01010	01011	01001	00000	100110	R	xor	$t1, $t2, $t3	Lógicas.
000000	01010	01011	01001	00000	100111	R	nor	$t1, $t2, $t3	Lógicas.
000000	01010	01011	01001	00000	101010	R	slt	$t1, $t2, $t3	Comparación.
000000	01010	01011	01001	00000	101011	R	sltu	$t1, $t2, $t3	Comparación Unsig.
000000	00000	00000	00000	00000	000000	R	nop		No operation.

La instrucción nop es por no operation; es decir, el procesador no debe realizar nada.

Nótese que el campo para corrimientos se rellena con ceros en caso de no emplearse.

Como este grupo tiene menos que 64 instrucciones (funct tiene 6 bits) puede codificarse de tal forma que la red combinacional, que detecta el código de operación(y que deberá estar en la unidad de control del procesador), resulte más simple. Por ejemplo, nótese que las operaciones aritméticas y lógicas tienen 10 en los bits más significativos de funct. Las operaciones de la unidad de multiplicación y división entre enteros tienen 01 en los bits más significativos de funct.

La tabla anterior permite **ensamblar** instrucciones, se entiende por esto: pasar de assembler a código binario.

El proceso de ensamblar.

Ejemplo 1.

Para la instrucción assembler: add $t1, $t2, $t3 se tienen los siguientes campos binarios:

000000 01010 01011 01001 00000 100000 add $t1, $t2, $t3

Notar la ubicación del campo registro destino, en el código binario y en el assembler simbólico.

Si se agrupan las cifras binarias en grupos de a cuatro bits, para expresar en hexadecimal, se obtiene:
0000 0001 0100 1011 0100 1000 0010 0000 add $t1, $t2, $t3

Leyendo la secuencia en hexadecimal, se obtiene: 0x014B4820 que representa al código simbólico: add $t1, $t2, $t3

Ejemplo 2.

Si se desea ensamblar: add $t3, $t2, $t1
Debemos cambiar los contenidos del destino (Rd) y el segundo registro de operandos(Rt), resultando los campos:
000000 01010 01001 01011 00000 100000 add $t3, $t2, $t1

Y agrupando, para expresar en hexadecimal:
0000 0001 0100 1001 0101 1000 0010 0000 add $t3, $t2, $t1
Entonces: 0x01495820 representa a add $t3, $t2, $t1

La siguientes instrucciones de control del procesador, de tipo R, tienen código de operación: 010000, y expanden el código de operación en el campo funct.

010000	10000	00000	00000	00000	010000	rfe
010000	10000	00000	00000	00000	000001	tlbr
010000	10000	00000	00000	00000	000010	tlbwi
010000	10000	00000	00000	00000	000110	tlbwr

La especificación c) define el formato I. Algunas de las instrucciones se muestran a continuación. Nótese que ahora el código de operación representa a la instrucción.

8.4. Formato I.

Código	Reg. Fuente	Reg. Destino	Campo **I**nmediato
6	Rs 5	Rt 5	Inm16

001000	01010	01001	0001000000000000	addi	$t1, $t2, 0x1000
001001	01010	01001	0001000000000000	addiu	$t1, $t2, 0x1000
001010	01010	01001	0001000000000000	slti	$t1, $t2, 0x1000
001011	01010	01001	0001000000000000	sltiu	$t1, $t2, 0x1000
001100	01010	01001	0001000000000000	andi	$t1, $t2, 0x1000
001101	01010	01001	0001000000000000	ori	$t1, $t2, 0x1000
001110	01010	01001	0001000000000000	xori	$t1, $t2, 0x1000
001111	00000	01001	0001000000000000	lui	$t1, 0x1000

Ejemplo 3.

Ensamblar ori $t5, $t3, 0x1234

Los valores de los campos son los siguientes:
Cop = 001101; Rs = $t3 = 01011(decimal 11); Rt = $t5 = 01101(decimal 13)
Inm16 = 0x1234 = 0001 0010 0011 0100 en binario.

Ensamblando los campos, se logra:
001101 01011 01101 0001 0010 0011 0100

Expresando en grupos de a 4 bits:
0011 0101 0110 1101 0001 0010 0011 0100

Finalmente: 0x356D1234 representa a: ori $t5, $t3, 0x1234

El proceso de desensamblar.

Si se desea desensamblar un código, dado en hexadecimal (así lo presentan algunos editores hexadecimales que permiten ver archivos binarios) deberá primero escribirse en binario.

En caso de instrucciones de largo fijo, la tarea de desensamblar es más sencilla; ya que se conoce, en forma simple, dónde comienza cada instrucción

Luego buscar, en los 6 primeros bits (en el caso de MIPS), el código de operación (si es preciso debe extenderse la búsqueda de la operación al campo de operación expandido). De esta forma se puede conocer el formato de la instrucción; luego se determinan los campos, y se interpretan los registros o campos inmediatos (si los hubiera).

Existen programas desensambladores, para facilitar esta tediosa tarea. Se suelen emplear para efectuar ingeniería inversa, y conocer así cómo se han logrado hacer algunos programas. Con estos conocimientos pueden encontrarse errores menores en los programas, reemplazar algunas instrucciones (introducir virus en los programas, sacar protecciones, etc.).

Cargas a registros.

Las especificaciones en e), permiten usar el mismo formato I, con la interpretación siguiente, para las **cargas**:

Código	Reg. Base	Reg. Destino	Offset en complemento dos
6	Rs 5	Rt 5	16

100000	01010	01001	0000000000000000	lb $t1, 0($t2)
100001	01010	01001	0000000000000000	lh $t1, 0($t2)
100010	01010	01001	0000000000000000	lwl $t1, 0($t2)
100011	01010	01001	0000000000000000	lw $t1, 0($t2)
100100	01010	01001	0000000000000000	lbu $t1, 0($t2)
100101	01010	01001	0000000000000000	lhu $t1, 0($t2)
100110	01010	01001	0000000000000000	lwr $t1, 0($t2)

La instrucción queda especificada por el código de operación.

Almacenar Registros.

Para los **almacenamientos**.

Código	Reg. Base	Reg. Fuente	Offset en complemento dos
6	Rs 5	Rt 5	16

101000	01010	01001	0000000000000000	sb $t1, 0($t2)
101001	01010	01001	0000000000000000	sh $t1, 0($t2)
101010	01010	01001	0000000000000000	swl $t1, 0($t2)
101011	01010	01001	0000000000000000	sw $t1, 0($t2)
101110	01010	01001	0000000000000000	swr $t1, 0($t2)

Direccionamiento relativo a PC.

En las instrucciones siguientes, que emplean direccionamiento relativo a PC, se empleará para el campo label, la constante binaria 0000000000001000, para confeccionar ejemplos. Más adelante se verán detalles sobre el ensamblamiento de este tipo de instrucciones.

Para bifurcaciones, también se emplea formato I.

Código	Reg. Fuente 1	Reg. Fuente 2	Offset en complemento dos, relativo a PC
6	Rs 5	Rt 5	16

000100	01010	01011	0000000000001000	beq $t2, $t3, label
000101	01010	01011	0000000000001000	bne $t2, $t3, label

Para flexibilizar la programación de condiciones se agregan, dos bifurcaciones, comparando un registro con cero. También emplean formato I.

Código	Reg. Fuente 1	Campo fijo	Offset en complemento dos, relativo a PC
6	Rs 5	Rt 5	16

000110	01001	00000	0000000000001000	blez $t1, label
000111	01001	00000	0000000000001000	bgtz $t1, label

El siguiente conjunto de bifurcaciones, comparando Registro con cero, tienen formato I, y todas tienen código de operación 000001, y expansión del código de operación en el campo entre los bits 20 y 16. Esto puede efectuarse, ya que este tipo de instrucciones no requieren dos registros, por lo que se emplea el espacio destinado normalmente a Rt, para la continuación del código.

Código	Reg. Fuente 1	Exp. COp	Offset en complemento dos, relativo a PC
6	Rs 5	5	16

000001	01001	00000	0000000000001000	bltz $t1, label
000001	01001	00001	0000000000001000	bgez $t1, label
000001	01001	10000	0000000000001000	bltzal $t1, label
000001	01001	10001	0000000000001000	bgezal $t1, label

8.5. Formato J. (en jumps)

Finalmente, los saltos a direcciones requieren especificar 32 bits, sin embargo 6 se emplean en el código de operación. Para resolver esto, al campo de 26 bits, disponible en la instrucción, se le agregan dos ceros (ya que todas las direcciones de instrucciones están alineadas y terminan en los bits 00). Los 4 bits que faltan, para especificar una dirección de 32 bits, se toman de los primeros 4 bits del PC. Esto permite saltos dentro de un segmento de la memoria. Como se tienen 4 bits para especificar el segmento, se considera dividida en 16 segmentos a la memoria.

Para confeccionar ejemplos, se emplea para el campo de 26 direcciones la constante: 00000100000000000000100100

Formato J. (en jumps)

Código de Operación.	Campo de dirección de 26 bits add_26
6	26

000010	00000100000000000000100100	j label
000011	00000100000000000000100100	jal label

8.6. Ejemplo 4. Saltos y bifurcaciones..

Referencias en saltos y bifurcaciones.

```
        .text
        .globl main

main:   beq     $t1,$t2,abajo
arriba: add     $t1,$t1,$t1
        j       abajo

abajo:  beq     $t1,$t2,arriba
        j       arriba

        j       main
```

El ejemplo muestra referencias hacia delante (salto hacia instrucciones que están cargadas en direcciones mayores de la memoria que la que se está ejecutando) y hacia atrás (el caso del salto hacia el rótulo arriba).

Ensamblamiento de saltos y bifurcaciones en MIPS.

Se efectuará el estudio, apoyándonos en el simulador SPIM, para esto cargamos el programa y observamos la ventana de texto. En ella aparecen las direcciones y contenidos

de las instrucciones en hexadecimal, además del assembler. SPIM agrega una columna central con campos de direccionamiento numérico (en decimal) y el símbolo de éste.

Se tiene el siguiente listado de un programa, entregado por SPIM (no se marcó bare-machine en: Simulator, settings, execution).

Dirección hex	Instrucción hex	Assembler simbólico numérico	Assembler simbólico
[0x00400020]	0x112a0003	beq $9, $10, 12 [abajo-x00400020]	beq $t1,$t2,abajo
[0x00400024]	0x01294820	add $9, $9, $9	arriba: add $t1,$t1,$t1
[0x00400028]	0x0810000b	j 0x0040002c [abajo]	j abajo
[0x0040002c]	0x112afffe	beq $9, $10, -8 [arriba-0x0040002c]	abajo: beq $t1,$t2,arriba
[0x00400030]	0x08100009	j 0x00400024 [arriba]	j arriba

El simulador presenta diferencias, respecto a lo que hemos planteado, en los códigos de los branch, si no se escoge bare machine.

Más adelante se darán detalles de esto. Pero la máquina nativa es la que estamos estudiando y deberemos marcar esa casilla para interpretar correctamente los campos de direccionamiento. El listado anterior se logra salvando el estado de la simulación en un archivo con extensión log.
Si se marca la casilla bare machine (sólo se ejecutarán las instrucciones propias del procesador, no estarán disponibles las macro instrucciones), se obtiene:

Dirección hex	Instrucción hex	Assembler simbólico numérico	Assembler simbólico
[0x00400020]	0x112a0002	beq $9, $10, 8 [abajo-x00400020]	beq $t1,$t2,abajo
[0x00400024]	0x01294820	add $9, $9, $9	arriba: add $t1,$t1,$t1
[0x00400028]	0x0810000b	j 0x0040002c [abajo]	j abajo
[0x0040002c]	0x112afffd	beq $9, $10, -12 [arriba-0x0040002c]	abajo: beq $t1,$t2,arriba
[0x00400030]	0x08100009	j 0x00400024 [arriba]	j arriba

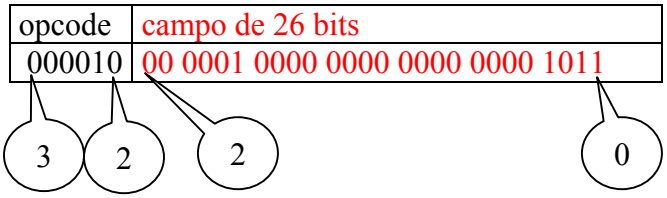
Direccionamiento en saltos:
a) Salto hacia delante.

```
        j       abajo
abajo:
```

Del listado del simulador, y de las dos primeras columnas, se obtiene para el salto hacia abajo, la dirección donde está almacenado el salto, y la secuencia hexadecimal de la instrucción de salto.

Dirección Instrucción
00400028 0810000B
las 26 últimas cifras binarias de la instrucción, es el campo de direccionamiento.
La siguiente dirección, en la primera columna del listado, es la dirección del rótulo abajo:
0040002C

Veamos ahora cómo se generó la instrucción de salto. Ésta tiene dos campos, los que se han escrito en binario a continuación:



El procesador interpreta esta instrucción, formando la dirección de salto, según:

Los primeros 4 bits los toma de los 4 bits más significativos del PC (bits 31 a 28).
En el caso del ejemplo, este campo es 0000.
Luego vienen los 26 bits de la instrucción (25 a 0).
Finalmente, se agregan dos ceros, ya que es un offset de direcciones de instrucciones (no bytes).

Entonces la dirección de salto resulta:
0000 00 0001 0000 0000 0000 0000 1011 00

Agrupando, para leer en hexadecimal:
0000-00 00-01 00- 00 00- 00 00- 00 00- 00 10 -11 00
Resulta: 0 0 4 0 0 0 2 C
Que es la dirección del rótulo abajo. Y así figura en la tercera columna de assembler numérico.

b) Saltos hacia atrás.
Al rótulo *arriba* le corresponde la dirección: [0x00400024]
La instrucción: j arriba, tiene por código: 0x08100009
Entonces, se obtiene empleando un desarrollo similar al anterior:
0000 00 0001 0000 0000 0000 0000 1001 00 que expresada en hexadecimal es la dirección del símbolo arriba.

c) Branch hacia adelante.
El rótulo *abajo* equivale a [0x0040002c].
La instrucción: beq $t1, $t2, abajo está codificada según: 0x112a0002 y está ubicada en la dirección: 0x00400020.

El código de la instrucción en binario: 0001 0001 0010 1010 0000 0000 0000 0010
Según campos: 000100 01001 01010 0000000000000010
El opcode 000100 es beq.
Rs es 01001, es el registro $9. Cuyo símbolo es $t1.
Rt es 01010, es el registro $10. Cuyo símbolo es $t2.

El *offset de instrucciones*, es un campo de 16 bits, y su valor es: 0x0002

El *offset de bytes*, en binario, es: 0000000000000001000 que equivale a +8 decimal, en un campo de 16 bits con signo (este valor es el que coloca en la tercera columna el simulador; es decir la denominada assembler simbólico numérico).

La dirección de salto se calcula sumando a la dirección de la instrucción más 4, el offset de bytes.
Entonces: (0x00400020 + 4) + 0x8 = 0x0040002C

d) Branch hacia atrás.
El rótulo arriba es equivalente a la dirección: 0x00400024
La instrucción: beq $t1, $t2, arriba está ubicada en: 0x0040002c
Su *ensamble* es entonces:

El opcode de beq es 000100.
$t1 es el registro $t9, y corresponde a 01001.
$t2 es el registro $t10, y su valor en campo de 5 bits es: 01010

Desde la ubicación de la próxima instrucción al beq, el rótulo arriba se encuentra tres instrucciones hacia atrás. Entonces el valor del offset es –3 en decimal.
+3 en un campo de 16 bits es: 0000 0000 0000 0011
El complemento a 1, es: 1111 1111 1111 1100
Y el complemento dos de +3(que equivale a –3), es entonces: 1111 1111 1111 1101
Lo cual equivale a: 0xFFFD
Finalmente, la instrucción binaria resulta: 000100 01001 01010 1111 1111 1111 1101
Que en hexadecimal es: 0x112afffd que es el código de la instrucción.

8.7 Resumen:

Se describe con detalle los procesos de ensamblado y desensamblado ya que más adelante estos mismos procesos deberán ser programados electrónicamente.