6 Capítulo 6 máquina rudimentaria

6.1 Traduce a lenguaje mnemónico de la MR las siguientes instrucciones expresadas en hexadecimal.

a) 1006h
b) E20h
c) 80A1h
d) EA64h
e) C6F1h
f) E846h
g) A825h

6.2 Pasa a notación binaria y hexadecimal las siguientes instrucciones escritas en lenguaje mnemónico de la MR.

a) ADD R0, R1, R2
b) SUBI R4, #1, R5
c) BGE 43
d) BR 12
e) AND R1, R7, R6
f) ASR R4, R2

6.3 Indica cuál es el resultado de ejecutar las siguientes instrucciones, dando el contenido final de los registros y posiciones de memoria que se modifican y el valor de los indicadores de condición. Se supone para cada caso que el contenido inicial de los registros y posiciones de memoria es el siguiente:

R1=2454h
R2=5656h
R3=FFFFh
R4=0000h
R5=0002h

Memoria

Dirección	Contenido
00h	0339h
01h	63AFh
02h	EA00h
03h	3304h
04h	7834h
05h	54AAh

a) SUB R1,R2,R5
b) ADDI R3, #1, R1
c) LOAD 3(R5), R1
d) STORE R4, 1(R4)
e) ASR R2, R7

6.4 Indica qué operación realiza la siguiente secuencia de instrucciones.

D024h
C864h
D844h

6.5 La instrucción *SWAP a, b* está presente en algunos lenguajes de alto nivel. Esta instrucción intercambia los valores de las variables *a* y *b*. Escribe en lenguaje máquina de la MR un programa que se comporte como esta instrucción, suponiendo que las variables *a* y *b* se encuentran:

a) La variable *a* en la posición de memoria 60h y la variable *b* en el registro R2 de la Unidad de Proceso.
b) La variable *a* en la posición de memoria 25h y la variable *b* en la posición 1Dh.

6.6 Escribe en lenguaje máquina de la MR un programa que cuente el número de bits que valen 1 en un número *NUM* almacenado en memoria. Supón que el número NUM se encuentra en la dirección 03h y que las instrucciones se almacenan a partir de la dirección 04h.

6.7 Escribe en lenguaje máquina de la MR un programa que realice la división por 2^n de un número NUM. El exponente n es un número positivo menor de 16. Utiliza desplazamientos para realizar la división, y supón que:

- el número NUM se encuentra en la dirección 13h
- el exponente n la dirección 17h
- las instrucciones se almacenan a partir de la dirección 2Ah

6.8 Escribe en lenguaje máquina de la MR un programa que indique si un número NUM es par o impar. El número NUM esta en la posición 03h de memoria. Utiliza la instrucción AND para averiguar el valor del bit de menor peso (0 par, 1 impar).
El programa debe poner un 1 en la posición de memoria 02h si el número es impar, y un 0 si es par. Supón que las instrucciones están almacenadas a partir de la dirección 04h.

6.9 Traduce el siguiente programa, escrito en lenguaje máquina de la MR (hexadecimal), a código de mnemónicos. Indica qué operación realiza, suponiendo que la primera instrucción

ejecutable se encuentra en la dirección 02h y que en las posiciones 00h y 01h hay datos.

Dirección	Contenido
00h	0009h
01h	XXXXh
02h	0800h
03h	D004h
04h	D824h
05h	8809h
06h	D224h
07h	DB09h
08h	8005h
09h	5001h

6.10 Indica qué operación realiza la siguiente secuencia de instrucciones

CA24h
D145h
C945h

6.11 Dado el siguiente fragmento de programa, escrito en algún lenguaje de alto nivel, tradúcelo a lenguaje máquina de la MR suponiendo que está almacenado a partir de la dirección de memoria 00h.

si R1>R2 ***entonces*** ejecutar la instrucción de la dirección 25h
sino seguir ejecutando en secuencia
fsi

6.12 Dado el siguiente fragmento de programa, escrito en algún lenguaje de alto nivel, tradúcelo a lenguaje máquina de la MR suponiendo que está almacenado a partir de la dirección de memoria A0h.

si R1>R2 ***entonces*** ejecutar la instrucción de la dirección 25h
sino ejecutar la instrucción de la dirección 64h
fsi

6.13 Dado el siguiente fragmento de programa, escrito en algún lenguaje de alto nivel, tradúcelo a lenguaje máquina de la MR suponiendo que está almacenado a partir de la dirección de memoria 38h.

si (R1>R2) o (R3 < 0) ***entonces*** ejecutar la instrucción de la dirección 25h
sino ejecutar la instrucción de la dirección 64h
fsi

6.14 Dado el siguiente fragmento de programa, escrito en algún lenguaje de alto nivel, tradúcelo a lenguaje máquina de la MR suponiendo que está almacenado a partir de la dirección de memoria 22h.

mientras (R1>R2) ***y*** (R3<=R4) ***hacer***
R1 := R1 - 1;
R3 := R3 + 1;
fmientras

6.15 Indica cómo afectaría a los siguientes elementos de la MR el aumentar de 8 a 16 el número de registros del Banco de Registros.

a) Tamaño de la instrucción
b) Anchura de la memoria
c) Buses internos de datos de la Unidad de Proceso
d) Registros internos de la Unidad de Proceso

6.16 Indica cómo afectaría a los siguientes elementos de la MR el aumentar el tamaño de la memoria hasta tener una memoria de 64 Kpalabras en lugar de 256 palabras.

a) Registro Contador de Programa (PC)
b) Registro de direcciones (R@)
c) Tamaño de la instrucción (Registro RI) y anchura de la memoria
d) Anchura del Banco de Registros y buses internos

6.17 Indica qué modificaciones habría que hacer en los siguientes elementos de la MR si se desease trabajar con números en complemento a dos de 32 bits, en lugar de hacerlo con números de 16 bits.

a) La ALU
b) La Memoria
c) El Banco de Registros
d) Los buses de datos internos de la Unidad de Proceso.
e) Los registros internos de la Unidad de Proceso

6.18 Realiza el diseño de la Unidad de Evaluación de la Condición de Salto.

6.19 Rediseña la ALU para que, además de los indicadores de condición Z y N, también calcule los indicadores:

* V: desbordamiento para enteros
* C: desbordamiento para naturales

6.20 Contesta a los siguientes apartados:

a) Rediseña la ALU para que un único sumador realice las operaciones de suma y de resta que necesitan las instrucciones ADD, SUB, ADDI y SUBI.

b) ¿Es necesario modificar la codificación de las instrucciones?

6.21 Queremos substituir la instrucción ASR de la MR por una instrucción que compare dos operandos fuente (CMP Rf1 Rf2) y modifique el indicador de condición Z de la manera que se indica a continuación, sin importar el valor que se guarde en el indicador de condición N:

* Z=1 si los dos operandos son iguales
* Z=0 si los dos operandos son distintos

a) Diseña la nueva ALU y rediseña el hardware que rodea la ALU.

b) ¿Qué implicaciones tiene el intercambiar la instrucción ASR por la instrucción CMP en el formato de las instrucciones?

6.22 Suponiendo que el tiempo de respuesta de los distintos elementos de la Unidad de Proceso es el siguiente:

- 10 ns. para cada multiplexor y decodificador
- 50 ns. para el sumador de direcciones y el incrementador conectado a la entrada del PC
- 40 ns. para el bloque que realiza la evaluación de la condición de salto
- 20 ns. para leer un registro del Banco de Registros
- 100 ns. el tiempo de respuesta de la ALU
- 100 ns. el tiempo requerido para leer o escribir un dato en la memoria RAM

Responde a las siguientes preguntas, despreciando el tiempo que pueda tardar en reaccionar la Unidad de Control.

a) Calcula el tiempo requerido para ejecutar cada una de las fases de cada instrucción (cada estado del grafo de estados reducido).
b) Indica la frecuencia máxima de reloj a la que puede funcionar la MR.

6.23 Suponiendo que se incluye un registro RMEM entre el multiplexor SELDAT y la memoria, que no afecte al camino entre la salida de la memoria y el registro IR, tal y como se muestra en la figura, contesta a los siguientes apartados:

a) Diseña el nuevo grafo de estados para ejecutar las instrucciones de la MR de forma que se optimice el número de estados necesario para ejecutar la instrucción LOAD. Indica la tabla de salidas de los nuevos estados
b) Contesta a los apartados a) y b) del problema 6.22 utilizando el nuevo grafo de estados.
c) ¿Qué mejora supone (en tiempo de ejecución) incluir el registro RMEM en la Unidad de Proceso?

6.24 Queremos diseñar una máquina con el mismo juego de instrucciones que la MR:

a) Propón una codificación de instrucciones en el supuesto de que dicha máquina:

- tenga 16 registros de propósito general de 32 bits.
- trabaje con números en complemento a dos de 32 bits.
- tenga una memoria de 2^{22} palabras de 32 bits.

¿Cuál es el tamaño máximo del operando inmediato?

b) ¿Qué modificaciones habría que hacer en el resto de la Unidad de Proceso?

c) ¿Cómo se puede aumentar el número de instrucciones aritméticas hasta 64? Qué repercusiones tendría este aumento en el resto de parámetros de las instrucciones?

d) ¿Se podrían añadir más instrucciones de salto sin modificar el resto de formatos de instrucción? ¿Cuántas y cuáles sugerirías, suponiendo que en la Unidad de Proceso se incluyesen los indicadores de condición C (acarreo) y V (desbordamiento)?

6.25 Contesta a las siguientes preguntas:

a) Diseña un Banco de Registros que posea dos puertos de lectura. El hecho de que un Banco de Registros tenga dos puertos de lectura implica que se pueden leer dos registros al mismo tiempo.

b) ¿Qué cambios en el diseño de la Unidad de Proceso de la MR supondría substituir su Banco de Registros por el que se propone en el apartado a)? ¿Tendría que cambiar el formato de las instrucciones?

6.26 Queremos añadir al juego de instrucciones de la MR una instrucción aritmética que multiplique por dos un operando fuente almacenado en un registro y deje el resultado en un registro destino (MULDOS Rf, Rd).

a) ¿Qué implicaciones tiene en el formato de las instrucciones el añadir esta instrucción?
b) Diseña la nueva ALU y rediseña el hardware que rodea la ALU.

6.27 Dado el siguiente programa, escrito en lenguaje mnemónico de la MR:

```
00h BR 03h
01h SUB R0, R0, R0
02h BR 05h
03h ADD R0, R0, R0
04h BGE 01h
```

Contesta a las siguientes preguntas:

a) ¿En qué orden se ejecutan las instrucciones?
b) ¿Cuántos ciclos tarda en ejecutarse el programa (suponiendo el grafo más simplificado para la Unidad de Control)?
c) Indica los diferentes valores que toma el PC en cada una de las fases de ejecución de las instrucciones durante la ejecución del programa.
d) Compara la secuencia de direcciones almacenada en el PC con la secuencia de direcciones correspondiente a la ejecución de instrucciones. ¿Se corresponden ambas secuencias? Justifica la respuesta.

6.28 Dado el siguiente bucle, escrito en lenguaje mnemónico de la MR:

```
00h SUB R3, R2, R0
01h BL 08h
02h SUB R4, R2, R0
03h BNE 06h
04h ADDI R2, #1, R2
05h BR 07h
06h SUB R4, #2, R4
07h BG 00h
```

Suponiendo que inicialmente R3=100_{10} y R2=49_{10}, contesta a los siguientes apartados:

a) Describe qué hace el cuerpo del bucle
b) ¿Cuántas veces se ejecuta el bucle?

6.29 ¿Qué se le indica a la Unidad de Proceso de la MR con estas señales de control?

a) Ld_PC = 1, Ld_IR = 1 y PC/@ = 0
b) Ld_PC = 1, Ld_IR = 1 y PC/@ = 1
c) Ld_RA = 1 y CRf= 1
d) ERd = 1
e) Ld_RZ = 1, Ld_RN = 1, ERd = 1, OPERAR = 1 y CRf = 2

6.30 Diseña la Unidad de Control de la MR como un sistema lógico secuencial cuya función de estado se realiza de forma mínima a tres niveles, y cuya función de salida se realiza con una memoria ROM.

6.31 Contesta a los siguientes apartados:

a) ¿Por qué se ha de realizar la decodificación de la instrucción?

b) ¿Por qué se intenta reducir el número de estados de la Unidad de Control de la MR?

6.32 Si el Banco de Registros de la Unidad de Proceso de la MR poseyese dos puertos de lectura (ver problema 6.25):

a) ¿ Podría ser más corto el tiempo de ejecución de alguna instrucción?

b) ¿Mejoraría el rendimiento global de la MR durante la ejecución de instrucciones?.

6.33 Si la ALU de la MR, además de los indicadores de condición Z y N también calculara los indicadores:

- V: desbordamiento para enteros
- C: desbordamiento para naturales

sería interesante incluir más instrucciones de salto:

a) ¿Qué instrucciones de salto se podrían añadir?

b) ¿Sería necesario cambiar el formato de las instrucciones?

6.34 Traduce el siguiente programa, escrito en un lenguaje de alto nivel, a lenguaje ensamblador de la MR y posteriormente a lenguaje máquina, suponiendo que:

- se almacena a partir de la dirección 00h
- las variables *a* y *b* se encuentran en las posiciones de memoria 00h y 01h respectivamente
- la primera instrucción ejecutable del programa está en la dirección 02h
- la instrucción *swap a, b* intercambia los valores de las variables *a* y *b*

```
programa ejemplo1;
var a, b: enteros;
a := 13;
b := 16;
mientras (a > 10) hacer
        a := a -1;
        b := b +2;
fmientras;
si (a<b)  entonces swap (a, b)
          sino b :=a-1
fsi;
fprograma.
```

6.35 Traduce el siguiente programa, escrito en un lenguaje de alto nivel, a lenguaje ensamblador de la MR y posteriormente a lenguaje máquina, suponiendo que se almacena a partir de la dirección 00h.

```
programa ejemplo2;
var a, b, c : enteros;
a := 13;
b := 1;
c := 0;
mientras (a > 10) o (b < 20) hacer
        si ( a < 11 ) y ( b > 18 ) entonces c := c + 3 fsi;
```

```
        a := a - b;
fmientras;
fprograma.
```

6.36 El siguiente programa calcula el máximo común divisor de dos números *a* y *b* según el algoritmo de restas de Euclides:

```
programa ejemplo3;
var a=5, b=15, mcd: enteros;
mientras (a <> b) hacer
        si (a>b)  entonces a:=a-b
                  sino  b:=b-a
        fsi;
fmientras;
mcd := a;
fprograma.
```

a) Tradúcelo a lenguaje ensamblador de la MR

b) Traduce el programa escrito en lenguaje ensamblador del apartado a) a lenguaje máquina, tanto en binario como en hexadecimal, suponiendo que:

- se almacena a partir de la dirección 0Fh
- las variables *a* y *b* se almacenan en las posiciones 0Fh y 10h respectivamente
- la variable *mcd* se almacena en la posición 11h
- la primera instrucción ejecutable del programa está en la dirección 12h.

6.37 Dado el siguiente fragmento de programa escrito en lenguaje ensamblador de la MR:

```
N=6
X=8
Y=12
.org 16
adrA:   .dw X,Y
        .dw 0, 0
```

```
adrB:  .dw N, X+Y, N+2
adrV:  .dw 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
       .dw 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

a) Indica el contenido de la tabla de símbolos.

b) Ensambla las siguientes instrucciones utilizando la anterior declaración de datos. Indica el valor final de los registros una vez ejecutadas las instrucciones.

- LOAD adrA+1(R0), R1
- LOAD adrB-4(R0), R2
- ADD R1, R2, R3
- STORE adrV+N(R2), R3

c) Sin añadir más etiquetas, escribe en lenguaje ensamblador de la MR un programa que:

- Asigne el valor N a la posición 14 del vector adrV.
- Asigne el contenido de la posición 10 del vector adrV, más el contenido de la posición 2*N de la memoria, a la posición X+Y del vector.

6.38 Escribe en un lenguaje de alto nivel un programa que calcule el cuadrado de un número positivo *a* mediante la suma acumulada *a* veces. Tradúcelo a lenguaje ensamblador de la MR y posteriormente a lenguaje máquina, suponiendo que se almacena a partir de la dirección 00h.

6.39 El siguiente algoritmo descrito en un lenguaje de alto nivel calcula el factorial de un número *a*. Tradúcelo a lenguaje ensamblador de la MR y posteriormente a lenguaje máquina, suponiendo que se almacena a partir de la dirección 14h. Indica el contenido de la tabla de símbolos.

Programa FACTORIAL;
var a, ind1, ind2, asumar, acum, factorial: ***enteros***;
a:= 9;
ind1:= a;
asumar := a;
acum:= 0;

```
mientras ( ind1 > 2) hacer
        ind2 := ind1 - 1;
        mientras ( ind2 > 0 ) hacer
                acum := acum + asumar;
                ind2 := ind2 - 1;
        fmientras;
        asumar := acum;
        acum := 0;
        ind1 := ind1 - 1;
fmientras;
factorial := asumar;
fprograma.
```

6.40 Escribe en un lenguaje de alto nivel un programa que calcule el número de letras *"a"* de una frase terminada en punto. Traduce el programa a lenguaje ensamblador y lenguaje máquina de la MR, suponiendo que:

- El programa se encuentra almacenado a partir de la dirección 80h.
- La frase se encuentra almacenada a partir de la dirección 02h en posiciones consecutivas de memoria.
- Cada letra ocupa una posición de memoria y está codificada utilizando el código ASCII. En este sistema de codificación, cada símbolo se codifica con 7 bits; en este caso, en los 7 bits de menor peso de la posición de memoria que ocupa. Los 9 bits de mayor peso están a 0. La letra "*a*" se codifica mediante el número 61h, y el carácter "." mediante el número 2Eh.
- Los valores 61h y 2Eh se pueden almacenar en las direcciones 00h y 01h. La frase nunca tiene más de 125 letras, y siempre termina en ".".

6.41 Escribe en un lenguaje de alto nivel un programa que calcule la suma de los elementos de un vector al mismo tiempo que busca los elementos máximo y mínimo. Tradúcelo a lenguaje ensamblador de la MR y posteriormente a lenguaje máquina, suponiendo que se almacena a partir de la dirección 3Ah.

6.42 Escribe en un lenguaje de alto nivel un programa que sume, elemento a elemento, dos vectores A y B, dejando el resultado en un tercer vector C (C[i] := A[i] + B[i]). El tamaño de los vectores es de 10 elementos. Traduce el programa a lenguaje ensamblador de la MR y posteriormente a lenguaje máquina, suponiendo que los vectores se almacenan a partir de la dirección de memoria 00h y el programa se almacena a partir de la dirección 80h.

6.43 Escribe en lenguaje ensamblador de la MR las macros que permitan ejecutar las siguientes instrucciones, pertenecientes a los juegos de instrucciones de otros computadores:

a) CLR Rd	Pone un 0 en el registro destino
b) CLR A(Ri)	Pone un 0 en la posición de memoria M[A+(Ri)]
c) INC Rd	Incrementa en una unidad el valor del registro Rd
d) DEC Rd	Decrementa en una unidad el valor del registro Rd
e) ADD3 Rf1, Rf2, Rd	Suma el contenido de 3 registros del Banco de Registros y deja el resultado en uno de ellos (Rd)
f) ASL Rd	Desplaza un bit a la izquierda el registro Rd
g) MOV Rf, Rd	Copia el valor del registro Rf en el registro Rd
h) MOV Rf, A(Ri)	Copia el valor del registro Rf en la posición de memoria M[A+(Ri)]
i) MOV A(Ri), B(Rj)	Copia el valor de la posición de memoria M[A+(Ri)] en la posición de memoria M[B+(Rj)]
j) SWAP Rf1, Rf2	Intercambia los valores de los registros Rf1 y Rf2
k) SWAP A(Ri), B(Rj)	Intercambia los valores de las posiciones de memoria M[A+(Ri)] y M[B+(Rj)]
l) ASRN #n, Rd	Desplaza n bits a la derecha el registro Rd. El número n ha de ser mayor que 0 y menor que 16
m) ASLN #n, Rd	Desplaza n bits a la izquierda el registro Rd. El número n ha de ser mayor que 0 y menor que 16
n) INC A(Ri)	Incrementa el contenido de la posición de memoria M[A+(Ri)]

6.44 Escribe en un lenguaje de alto nivel un programa que calcule la suma de los elementos de un vector V que cumplan ser mayores que un cierto valor MÍNIMO y menores que un cierto valor MÁXIMO (MÍNIMO < V[i] < MÁXIMO). Tradúcelo a lenguaje ensamblador de la MR y posteriormente a lenguaje máquina, suponiendo que el programa se almacena a partir de la dirección 3Ah.

6.45 Traduce a lenguaje ensamblador de la MR el siguiente programa escrito en un lenguaje de alto nivel. El programa calcula el factorial de un número "a" (en el ejemplo, a=8):

```
programa FACTORIAL;
var a, cont, factorial: enteros;
a:= 8;
cont := 1;
factorial := 1;
mientras ( cont <= A ) hacer
        cont := cont + 1;
        factorial := factorial * cont;
fmientras;
fprograma.
```

Diseña y usa la macro *MUL $1, $2, $3*, que multiplica el contenido de dos registros ($1, $2) y deja el resultado en un tercer registro ($3). Los tres registros deben ser diferentes.
Haz el preensamblado (expansión de macros) y el postensamblado (creación de la tabla de símbolos y generación de código) del programa, suponiendo que el programa se almacena a partir de la dirección de memoria 00h.

6.46 Dado el siguiente programa escrito en lenguaje ensamblador de la MR:

```
N:      .dw 4
RESUL:  .rw 1
        .def CMP $1, $2
            sub $1, $2, R0
        .enddef
        .def CMP $1, $i2
            subi $1, $i2, R0
        .enddef
        .def CLR $1
            add R0, R0, $1
        .enddef
        .def MOV $1, $2
            add R0, $1, $2
        .enddef
        .def MOV $i1, $2
            addi R0, $i1, $2
        .enddef
        .def MOV $d1, $2
            load $d1, $2
        .enddef
        .def MOV $1, $d2
            store $1, $d2
        .enddef
        .def INC $1
            addi $1, #1. $1
        .enddef
        .def DEC $1
            subi $1, #1. $1
        .enddef
        .begin INI
INI:    MOV N(R0), R1
        MOV R1, R3
```

```
        CLR R4
M1:     CMP R1, #2
        BLE FM1
        SUBI R1, #1, R2
M2:     CMP R2, #0
        BLE FM2
        ADD R4, R3, R4
        DEC R2
        BR M2
        FM2:   MOV R4, R3
        CLR R4
        DEC R1
        BR M1
FM1:    MOV R3, RESUL(R0)
        .end
```

a) Realiza el preensamblado (expansión de macros).

b) Realiza el postensamblado (generación de la tabla de símbolos y traducción a lenguaje máquina) suponiendo que el programa se almacena a partir de la posición 00h de memoria.

c) Traduce el programa a alto nivel, indicando qué operación realiza. Para ello, intenta analizar qué operación realiza cada macro.

Solución a los Problemas del Capítulo 6

6.1

a) 1006h = 0001 0000 0000 011 = 00 010 000 00000110 = LOAD 6(R0), R2
b) 4E20h = 0100 1110 0010 0000 = 01 001 110 00100000 = STORE R1, 2Oh(R6)
c) 80A1h = 1000 0000 10100001 = 10 000 000 10100001 = BR A1h
d) EA64h = 1110 1010 0110 0100 = 11 101 010 011 00 100 = ADD R2, R3, R5
e) C6F1h = 1100 0110 1111 0001 = 11 000 110 11110 001 = SUBI R6, #-2, R0
f) E846h = 1110 1000 0100 0110 = 11 101 000 010 00 110 = ASR R2, R5
g) BNE 25h = 1010 1000 0010 0101 = 10 101 000 00100101 = BNE 25h

6.2

a) ADD R0, R1, R2 = 11 010 000 001 00 100 = 1101 0000 0010 0100 = D024h
b) SUBI R4, #1, R5 = 11 101 100 00001 001 = 1110 1100 0000 1001 = EC09h
c) BGE 43 = 10 110 000 00101011 = 1011 0000 0010 1011 = B02Bh
d) BR 12 = 10 000 000 0000 1100 = 1000 0000 0000 1100 = 800Ch
e) AND R1, R7, R6 = 11 110 001 111 00 111 = 1111 0001 1110 0111 = F1E7h
f) ASR R4, R2 = 11 010 xxx 100 00 110 = 1101 0xxx 1000 0110 = D086h (por ejemplo)

6.3

a) SUB R1,R2,R5

La instrucción realiza la operación R5 := R1 - R2
Después de ejecutar la instrucción, R5 vale 2454h - 5656h = CDFEh. Los registros R1 y R2 no varían. Los indicadores de condición se activan según el valor escrito en R5. Por tanto, Z=0 y N=1.

b) ADDI R3, #1, R1

La instrucción realiza la operación R1 := R3 + 1
Después de ejecutar la instrucción, R1 vale FFFFh + 0001h = 0000h (hay un bit de acarreo que se desprecia porque el sumador de la ALU no lo tiene en cuenta).
El registro R3 no varía.
Los indicadores de condición se activan según el valor escrito en R1. Por tanto, Z=1 y N=0.

c) LOAD 3(R5), R1

La instrucción realiza la operación R1 := M[(R5)+3] := M[2+3] := M[5] := 54AAh
Después de ejecutar la instrucción, el registros R5 no varía.
Los indicadores de condición se activan según el valor escrito en R1. Por tanto, Z=0 y N=0.

d) STORE R4, 1(R4)

La instrucción realiza la operación M[(R4)+1] := R4; Por tanto, M[1] := 0000h
Después de ejecutar la instrucción, la posición de memoria 1 pasa a contener un 0.
El registro R4 no varía.
Los indicadores de condición no varían.

e) ASR R2, R7

La instrucción realiza la operación R7 := R2 >> 1
Como R2=0101 0110 0101 0110, el resultado de desplazar un bit a la derecha manteniendo el signo es 0010 1011 0010 1011 = 2B2Bh.
Después de ejecutar la instrucción, R7 vale 2B2Bh.
El registro R2 no varía.
Los indicadores de condición se activan según el valor escrito en R7. Por tanto, Z=0 y N=0.

6.4

La traducción a lenguaje mnemónico es la siguiente:

D024h = 1101 0000 0010 0100 = 11 010 000 001 00 100 = ADD R0, R1, R2
C864h = 1100 1000 0110 0100 = 11 001 000 011 00 100 = ADD R0, R3, R1

D844h = 1101 1000 0100 0100 = 11 011 000 010 00 100 = ADD R0, R2, R3

La secuencia de instrucciones intercambia el valor de los registros R1 y R3, utilizando el registro R2 como registro auxiliar.

R2 := R0 + R1 = R1 (porque R0=0 siempre)
R1 := R0 + R3 = R3 (por la misma razón)
R3 := R0 + R2 = R2

6.10

La traducción a lenguaje mnemónico es la siguiente:

CA24h = 1100 1010 0010 0100 = 11 001 010 001 00 100 = ADD R2, R1, R1
D145h = 1101 0001 0100 0101 = 11 010 001 010 00 101 = SUB R1, R2, R2
C945h = 1100 1001 0100 0101 = 11 001 001 010 00 101 = SUB R1, R2, R1

La secuencia de instrucciones intercambia el valor de los registros R1 y R2 sin utilizar ningún registro auxiliar.
Supongamos que R1=x, R2=y.

R1 := R2 + R1 = y + x
R2 := R1 - R2 = (y + x) - y = x
R1 := R1 - R2 = (y + x) - x = y

6.11

Dirección	Contenido
00h	SUB R1, R2, R0 = 11 000 001 010 00 101
01h	BG 25h = 10 111 000 0010 0101

6.12

Dirección	Contenido
A0h	SUB R1, R2, R0 = 11 000 001 010 00 101
A1h	BLE 64h = 10 011 000 0110 0100
A2h	BR 25h = 10 000 000 0010 0101

6.13

Dirección	Contenido
38h	SUB R1, R2, R0 = 11 000 001 010 00 101
39h	BG 25h = 10 111 000 0010 0101
3Ah	SUBI R3, #0, R0 = 11 000 011 00000 001
3Bh	BL 25h = 10 010 000 0110 0100
3Ch	BR 64h = 10 000 000 0110 0100

6.14

Dirección	Contenido
22h	SUB R1, R2, R0 = 11 000 001 010 00 101
23h	BLE 29h= 10 011 000 00101001
24h	SUB R3, R4, R0 = 11 000 011 100 00 101
25h	BG 29h = 10 111 000 00101001
26h	SUBI R1, #1, R1= 11 001 001 00001 001
27h	ADDI R3, #1, R3= 11 011 011 00001 000
28h	BR 22h= 10 000 000 00100010

6.15

a) Al disponer de 16 registros, son necesarios 4 bits para codificar a qué registro se quiere acceder en cada momento. Por tanto, por cada registro usado en cada instrucción hay que aumentar en un bit el tamaño de la instrucción.

- Instrucciones aritméticas con un operando inmediato:

  2 bits de CO
  4 bits Rd
  4 bits Rf1
  5 bits operando inmediato
  3 bits de operación

  TOTAL: 18 BITS

- Instrucciones aritméticas con operandos en registros:

  2 bits de CO
  4 bits Rd
  4 bits Rf1
  4 bits Rf2
  3 bits de operación

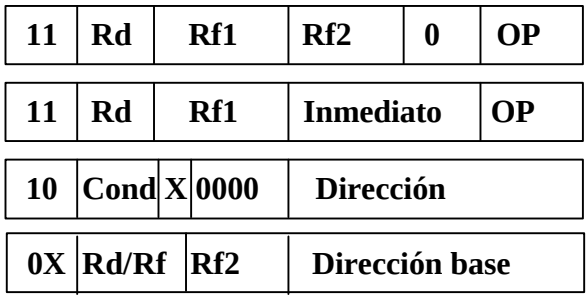
  TOTAL: 17 BITS

- Instrucciones de salto: Como no usan ningún registro, no se ven alteradas.
- Instrucciones de acceso a memoria:

2 bits CO
4 bits Rf (Store) o Rd (Load)
4 bits Ri
8 bits dirección base

TOTAL: 18 bits

Por tanto, el tamaño de la instrucción se vería incrementado hasta 18 bits. Una posible nueva codificación para las instrucciones sería la siguiente:



Aritmético/lógicas reg-reg

Aritmético/lógicas reg-inmediato

De salto (podría haber 16)

De acceso a memoria

b) Al aumentar el tamaño de la instrucción, también sería preciso aumentar la anchura de la memoria hasta 18 bits, con el objeto de que cada instrucción pudiese ser almacenada en una posición de memoria.
Como efecto lateral, al codificar las instrucciones como indica la figura se observa que sobra un bit para las instrucciones de salto y para las instrucciones aritmético-lógicas registro-registro. Este bit puede ser utilizado para incrementar el número de las mismas si es añadido al campo *Cond*. Por ejemplo, se podrían tener siete instrucciones que salten suponiendo que los números están escritos en complemento a dos y otras siete que salten suponiendo que son números naturales, y aún sobrarían dos codificaciones. Esta es una distinción muy común en prácticamente todos los computadores.

c) El bus que va desde la memoria al registro IR debería ser de 18 bits. Los 16 bits de menor peso se conectarían a las entradas 0 y 1 del multiplexor SELDAT. El resto de buses no se verían alterados.

d) Únicamente habría que aumentar el tamaño del registro IR hasta 18 bits.

ACLARACIÓN: A pesar de aumentar la anchura de la memoria, en las respuestas de este problema se ha considerado que el tamaño de los operandos sigue siendo de 16 bits (y no de 18). Si los operandos fuesen de 18 bits habría que cambiar las respuestas de los apartados c) y d).

6.16

a) Para direccionar 64 Kbytes hacen falta 16 bits. Por tanto, el PC debería ser de 16 bits.

b) El registro R@ contiene la dirección de salto cuando se ejecutan instrucciones de salto o la dirección de un dato cuando se ejecutan instrucciones de acceso a memoria. En cualquier caso, y por la misma razón que en el apartado a), debe ser de 16 bits.

c) Al aumentar el tamaño de una dirección a 16 bits, es preciso cambiar el formato de las instrucciones de salto para que contengan una dirección de 16 bits en lugar de una dirección de 8 bits (si se quiere mantener el direccionamiento absoluto). Por tanto, estas instrucciones pasarían a tener 24 bits, lo que implica que la anchura de la memoria y del registro de instrucción sería también de 24 bits.

Las instrucciones aritmético-lógicas y las de acceso a memoria no se verían modificadas. En el caso de las instrucciones de acceso a memoria, la dirección se calcularía sumando el contenido del registro índice (sus 16 bits permiten acceder a cualquier posición de memoria) con los 8 bits de menor peso de la instrucción. No obstante, y aprovechando que el tamaño de la instrucción se ha incrementado debido a las instrucciones de salto, se podría disponer de un desplazamiento de 16 bits en las instrucciones Load y Store.

Otra posible solución, que no necesitaría efectuar ningún cambio en la Unidad de Proceso, consistiría en utilizar el modo base+desplazamiento también en las instrucciones de salto, en lugar del modo absoluto. Esto implicaría que la dirección de salto se calcularía igual que la dirección de acceso a memoria en una instrucción Load o Store. Para ello, basta con substituir los bits 10-8 de la instrucción de salto (que contienen un cero) por la codificación del registro índice que se utilizaría para calcular la dirección de salto. Este cambio en la forma de calcular las direcciones de salto no requiere ninguna modificación de la Unidad de Proceso. El tamaño de la instrucción, y por tanto la anchura de la memoria, continuaría siendo de 16 bits.

d) Un registro del banco de registros es de 16 bits y puede direccionar hasta 64 Kbytes. Por tanto, no se requiere modificar el Banco de Registros. Sin embargo, el bus de salida del Banco de Registros que está conectado con la entrada del sumador debería ser de 16 bits, en lugar de 8 bits.

6.17

a) La ALU debería tener operadores de 32 bits (un sumador de 32 bits, un restador de 32 bits y 32 puertas AND para la instrucción AND).

b) La anchura de la memoria debería ser también de 32 bits, para que cada dato pudiese estar contenido en una única dirección. Las instrucciones pueden almacenarse, por ejemplo, en los 16 bits de menor peso de cada posición de memoria.

c) El Banco de Registros debería tener registros de 32 bits para albergar los operandos.

d) Todos los buses de datos que se comunican con el banco de registros y con la memoria deberían ser de 32 bits.

e) El registro RA debería ser de 32 bits para contener el primer operando de las instrucciones aritmético-lógicas. Sin embargo, el PC y el R@ seguirían siendo de 8 bits, ya que el bus de direcciones de la memoria sigue siendo de ese tamaño. El registro de instrucciones tampoco se vería alterado, ya que las instrucciones siguen siendo de 16 bits.

6.18 Existen varias opciones para diseñar la Unidad de Evaluación de la Condición de Salto. Veamos como se diseñaría con puertas lógicas.
El valor de la señal *Cond* se puede expresar con la función:

$$Cond = \overline{IR_{13}}..\overline{IR_{12}}.\overline{IR_{11}}. + (\overline{IR_{13}}..\overline{IR_{12}}.IR_{11}).Z + (\overline{IR_{13}}..IR_{12}.\overline{IR_{11}}).N + (\overline{IR_{13}}..IR_{12}.IR_{11}).(N+Z) +$$
$$+ (IR_{13}.\overline{IR_{12}}.IR_{11}).\overline{Z} + (IR_{13}..IR_{12}.\overline{IR_{11}}).\overline{N} + (IR_{13}.IR_{.12}\,IR_{11})(\overline{N+Z}.)$$

Simplificando por Karnaugh a tres niveles obtenemos la expresión:

$$Cond = .\overline{IR_{12}}.\overline{IR_{11}}. + \overline{IR_{13}}..IR_{11}.Z + \overline{IR_{13}}..IR_{12}.N + IR_{13}.\overline{IR_{12}}.\overline{Z} + IR_{13}..\overline{IR_{11}}.\overline{N} + IR_{13}.\overline{N.Z}$$

6.19

V: desbordamiento para enteros representados en complemento a dos.

El desbordamiento en la suma de números enteros se produce cuando los dos operandos a sumar son del mismo signo y el resultado es de signo contrario. El desbordamiento en la resta de números enteros se produce cuando los dos operandos a restar son de distinto signo y el resultado tiene el signo del sustraendo. Por tanto:

$$V = OPERAR.\overline{IR_1}.\overline{IR_0}(A_{15}.B_{15}.\overline{O_{15}} + \overline{A_{15}}.\overline{B_{15}}.O_{15}) + OPERAR.\overline{IR_1}.IR_0(A_{15}.\overline{B_{15}}.\overline{O_{15}} + \overline{A_{15}}.B_{15}O_{15})$$

C: desbordamiento para naturales.

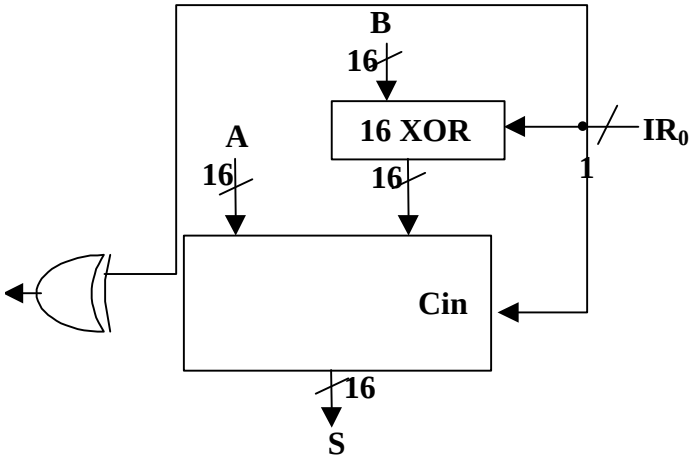
El desbordamiento en la suma y en la resta de números naturales se comprueba con el bit de acarreo de la operación realizada. Por tanto:

$$C = OPERAR.\overline{IR_1}.\overline{IR_0}Cout\_suma + OPERAR.\overline{IR_1}.IR_0.Cout\_resta$$

La realización de V y de C se puede hacer con cualquiera de los métodos estudiados para diseñar un sistema lógico combinacional.

6.20

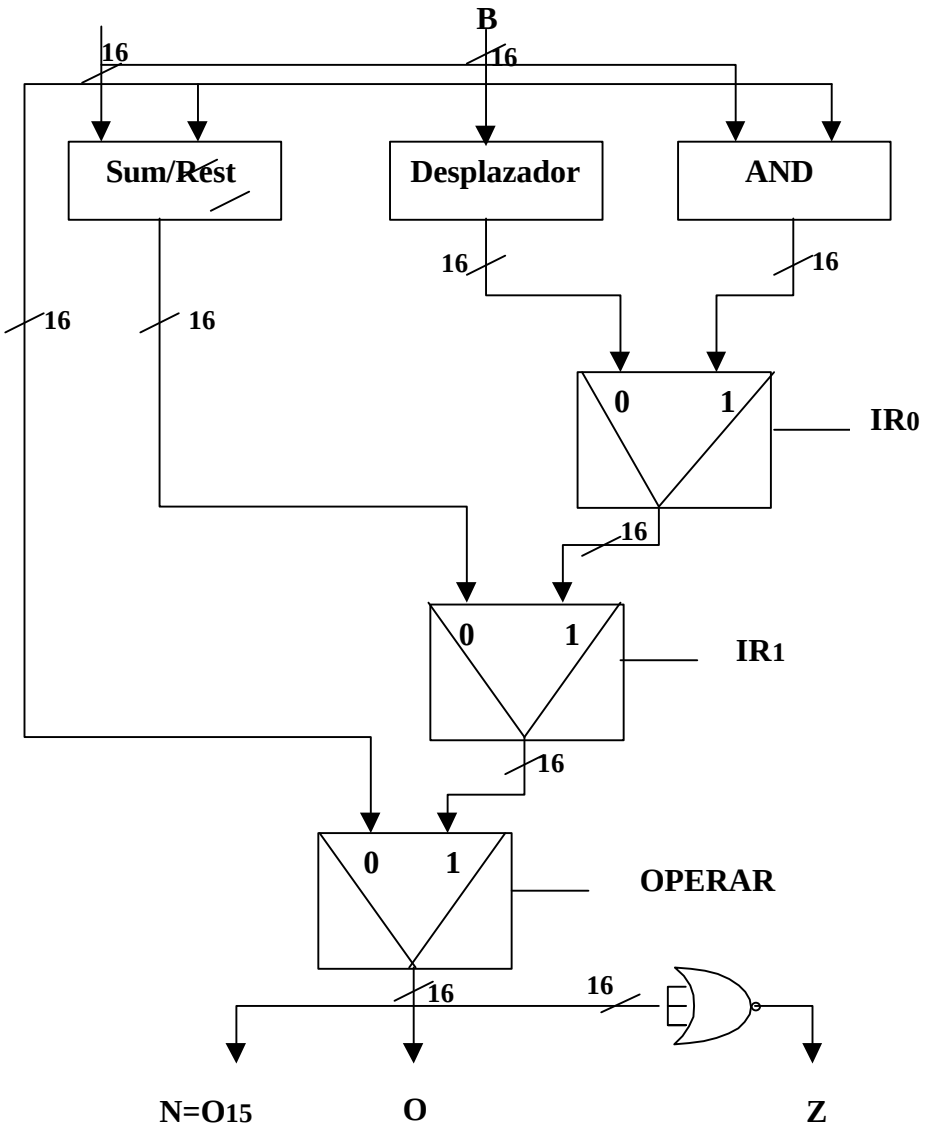
a) El circuito que realiza la Suma/Resta es:



El bit IR_0 permite seleccionar de forma automática la operación a realizar. Cuando vale 0 se realiza una suma, y cuando vale 1 se realiza una resta.

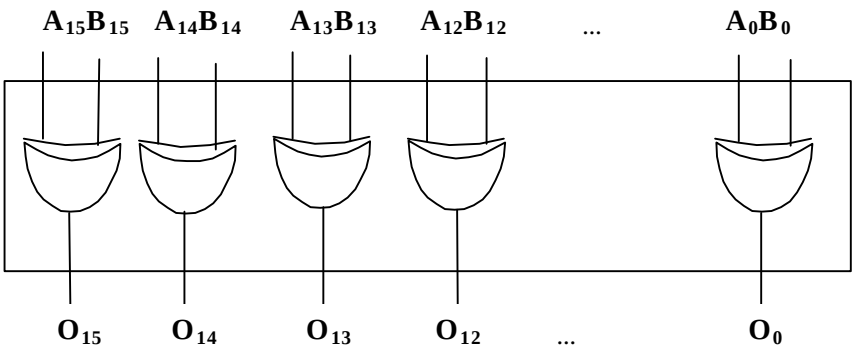
b) El formato de las instrucciones no tiene que cambiar porque, como se ha visto en el apartado a), la selección de la operación a realizar en el sumador/restador se realiza de forma automática mediante el bit IR_0.

El diseño final de la ALU es el mostrado en la siguiente figura:



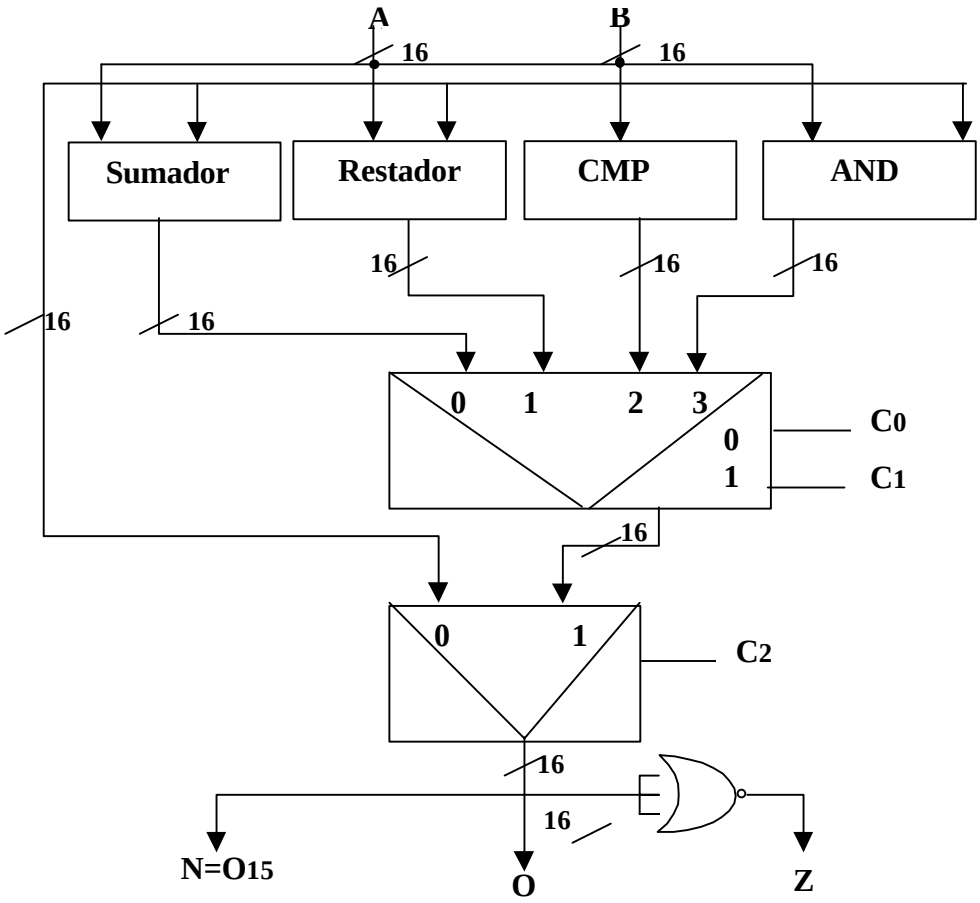
6.21

a) La nueva operación de comparar se puede realizar mediante el siguiente circuito:



BLOQUE CMP

Usando este bloque, la nueva ALU quedaría diseñada de la siguiente forma:



b)

Solución 1

Si el hardware del bloque CMP sustituye al que realiza la ASR, el código de la instrucción CMP puede ser el que tenía la instrucción ASR.

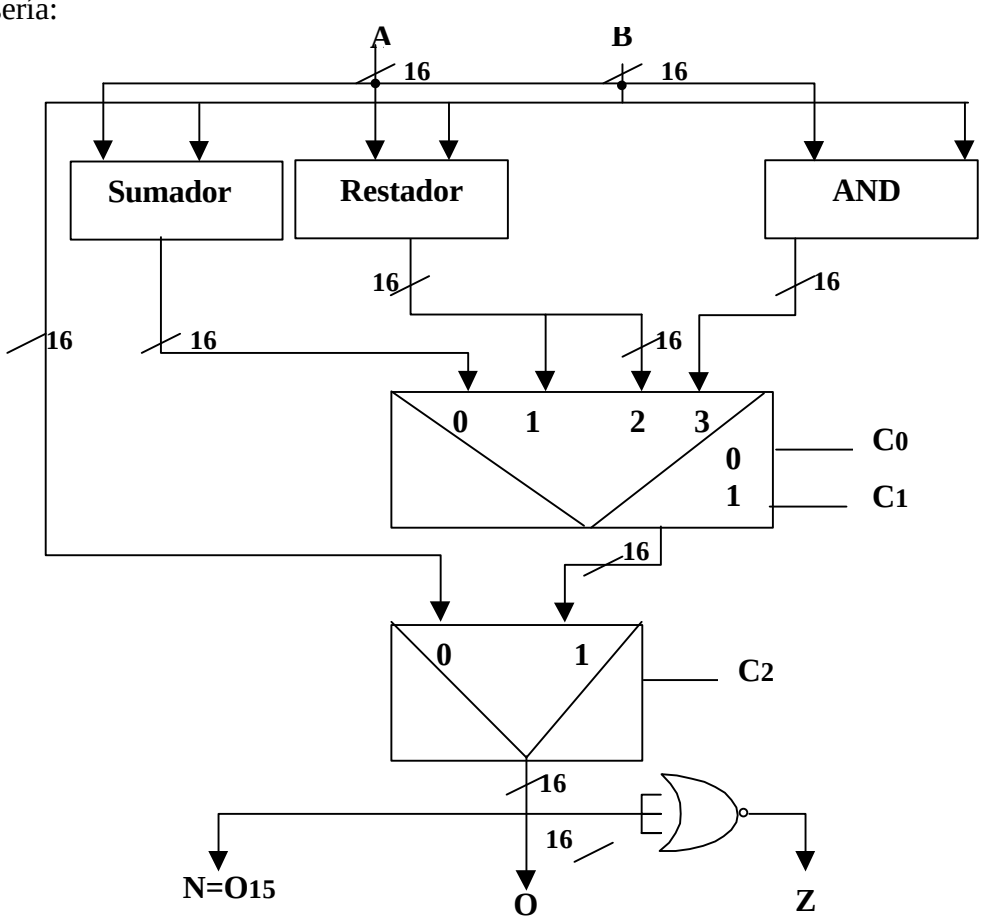
Con la ejecución de la instrucción CMP no nos interesa guardar el resultado de realizar la XOR de los dos operandos fuente ($O_{15\text{-}0}$). Si en el formato de esta instrucción indicamos que el registro destino es R0, no se guardará el resultado. Esta solución permite tratar la nueva instrucción igual que las demás instrucciones aritmético/lógicas.

11	000	Rf1	Rf2	00	110

CMP Rf1, Rf2

Solución 2

Otra posible solución es usar el propio restador de la ALU para implementar la operación CMP. De este modo, no sería necesario el uso de un bloque específico como el diseñado en el apartado a). Manteniendo la misma codificación que en la solución 1, la nueva ALU sería:



La instrucción de comparar CMP Rf1, Rf2 se traduciría igual que en la solución 1:

11	**000**	**Rf1**	**Rf2**	**00**	**110**	**CMP Rf1, Rf2**

Solución 3

Otra posible solución sería añadir la nueva instrucción CMP sin necesidad de eliminar ninguna de las anteriores. No sería necesario realizar ninguna modificación en la ALU, y bastaría codificarla como una instrucción de restar que deja el resultado en el registro R0. La nueva codificación sería, por tanto:

11	**000**	**Rf1**	**Rf2**	**00**	**101**	**CMP Rf1, Rf2 = SUB Rf1, Rf2, R0**

6.22

a) El tiempo mínimo requerido para completar un estado del grafo depende del retardo provocado por los diferentes circuitos combinacionales que intervienen en la ejecución de las operaciones realizadas en el estado.

Estado FETCH

La dirección almacenada en el PC atraviesa el multiplexor SELDIR (10 ns.) y llega al bus de direcciones de memoria. La memoria suministra el dato (100 ns.) al IR. En total, 110 ns.
Simultáneamente con la acción anterior, la dirección presente en la salida del multiplexor SELDIR (10 ns.) se incrementa en 1 (50 ns.) para estar presente en la entrada del PC. En total, 60 ns.
Como ambas operaciones se realizan en paralelo (simultáneamente), el tiempo mínimo requerido para ejecutar este estado es 110 ns.

Estado DECO

En este estado se realiza, además de la decodificación de la instrucción (cuyo tiempo despreciamos por realizarse en la Unidad de Control), el cálculo de la condición de salto, la carga del primer operando de las instrucciones aritmético-lógicas en el registro

RA y la carga de la dirección de acceso a memoria de las instrucciones Load y Store en el registro R@.

- El cálculo de la condición de salto requiere 40 ns.
- Para almacenar el primer operando de las instrucciones aritmético-lógicas en RA es preciso lanzar los 3 bits que codifican el registro fuente a través del multiplexor SELDIR (10 ns.) y leer el operando del Banco de Registros (20 ns.). En total, 30 ns.
- Para almacenar la dirección de acceso a memoria de las instrucciones Load y Store en el registro R@ hay que sumar el contenido del registro índice (30 ns. para acceder al Banco de Registros, 10 ns. para seleccionar los bits en SELDIR y 20 ns. para leer el registro) con la dirección base contenida en la instrucción (50 ns. para realizar la suma).
- En total, 80 ns.

Por tanto, el tiempo requerido por esta fase es de 80 ns.

Estado LOAD

La dirección contenida en R@ se lanza a la memoria a través del multiplexor SELDIR (10 ns.). La memoria sirve el dato (100 ns.), que pasa a través del multiplexor SELDAT (10 ns.) y de la ALU (100 ns.) para estar presente a la entrada del Banco de Registros. En total, 220 ns.

Estado STORE

La dirección contenida en R@ se lanza a la memoria a través del multiplexor SELDIR (10 ns.). En paralelo, el Banco de Registros envía a la memoria el dato que debe ser escrito, para lo cual tarda 30 ns. (10 ns. de SELDIR y 20 ns. de lectura del registro). La memoria tarda 100 ns. en escribirlo, por lo que el tiempo total es de 130 ns.

Estado ARIT

En este estado se realiza la fase de ejecución de las instrucciones aritmético-lógicas y, en paralelo, el fetch de la siguiente instrucción. Como se ha visto anteriormente, el fetch requiere 110 ns.

- Para ejecutar una instrucción aritmético-lógica registro-registro, el segundo operando se lanza desde el Banco de Registros (30 ns., 10 ns. de SELDIR y 20 ns. de lectura del registro) a través del multiplexor SELDAT (10 ns.) y la ALU realiza la operación correspondiente (100 ns.). En total, 140 ns.

  El tiempo requerido para escribir el resultado en el Banco de Registros no se

contabiliza, ya que esta escritura se realiza en paralelo con el fetch de la siguiente instrucción (y tarda menos tiempo).

- Para ejecutar una instrucción aritmético-lógica registro-inmediato, el segundo operando se lanza desde el registro IR (previa extensión de signo) a través del multiplexor SELDAT (10 ns.) y la ALU realiza la operación correspondiente (100 ns.). En total, 110 ns.
  Como en el caso anterior, el tiempo requerido para escribir el resultado en el Banco de Registros no se contabiliza.

Por tanto, el tiempo requerido para ejecutar esta fase es de 140 ns.

Estado BRANCH

En este estado se realiza fetch desde el registro R@, lo cual tarda el mismo tiempo que realizar el fetch desde el PC. Por tanto, este estado requiere 110 ns.

b) La frecuencia máxima depende del tiempo que tarde el estado que necesita más tiempo. Como se ha visto en el apartado a), el estado LOAD es el que tarda más tiempo: 220 ns. Esto implica que el período del reloj debe ser superior o igual a 220 ns, y como el período (T) es el inverso de la frecuencia (F), tenemos que:

$T \ 220 .10^{-9}$ s => F <= 1 / T ; F <= 1 / $(220 .10^{-9}) = 10^{9} / 220 = 4545454.5$ Hz = 4.54 MHz.

F <= 4.54 MHz.

6.23

a) La nueva Unidad de Proceso y el nuevo grafo de estados de la Unidad de Control son los siguientes:

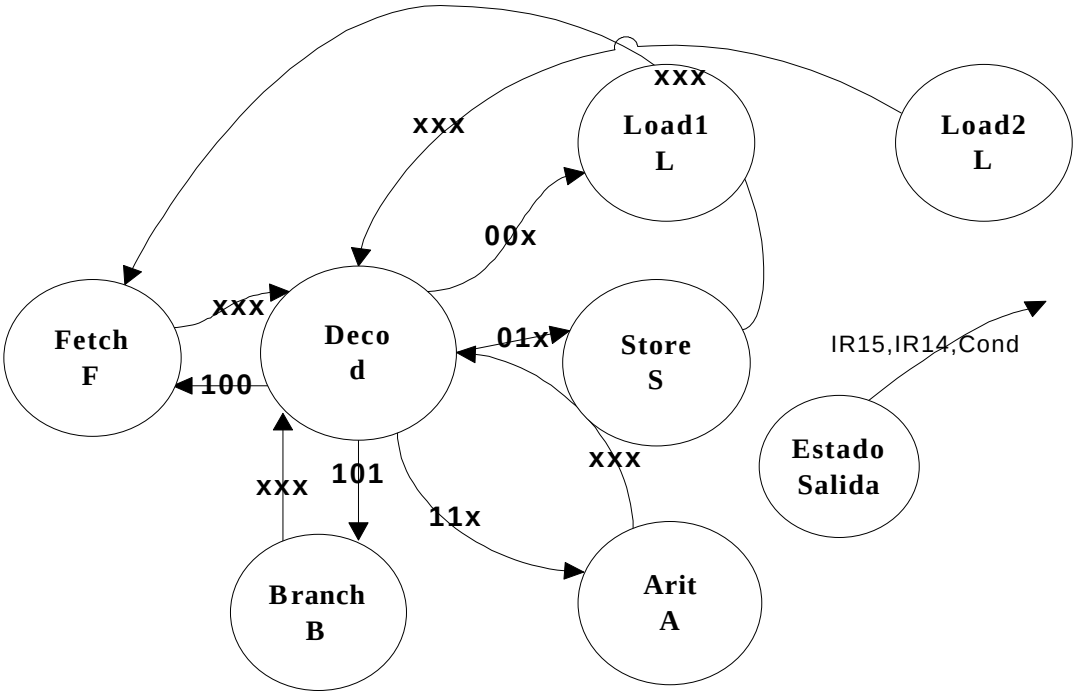


TABLA DE SALIDAS

Salidas UC	Load1	Load2
Ld_RA	0	0

Ld_IR	0	1
Ld_PC	0	1
Ld_R@	0	0
Ld_RMEM	1	0
Ld_RZ	0	1
Ld_RN	0	1
Erd	0	1
$\overline{L}$/E	0	0
$\overline{PC}$/@	1	0
CRf	X	X
OPERAR	X	0

En el estado LOAD1 se carga el operando fuente de la instrucción LOAD, una vez ha sido leído de la memoria, en el registro RMEM.
En el estado LOAD2 se almacena este operando en el registro destino, y en paralelo se realiza el fetch de la siguiente instrucción. El número de ciclos que tarda en ejecutarse una instrucción LOAD no aumenta a pesar de que sí se incrementa el número de estados del grafo, ya que después de realizar el estado LOAD2 se pasa a la fase de Decodificación, en lugar de pasar a la fase de Fetch como se hacía en el grafo original. Las tablas de salida de ambos estados se muestran en la Tabla.

b) El tiempo mínimo requerido por cada fase depende del retardo provocado por los diferentes circuitos combinacionales que intervienen en la ejecución de la fase. Como únicamente varían los estados LOAD 1 y LOAD2 en el grafo, y la inserción del registro RMEM sólo afecta a la ejecución de las instrucciones LOAD, el resto de los estados requieren el mismo tiempo estudiado para el Problema 6.22.

Estado FETCH: 110 ns.

Estado DECO: 80 ns.
Estado STORE: 130 ns.
Estado ARIT: 140 ns.
Estado BRANCH:110 ns.

Estado LOAD1

La dirección contenida en R@ se lanza a la memoria a través del multiplexor SELDIR (10 ns.). La memoria sirve el dato (100 ns.) y el resultado se escribe, al final del ciclo, en el registro RMEM. Por tanto, el tiempo requerido para este estado es de 110ns.

Estado LOAD2

El dato contenido en el registro RMEM pasa a través del multiplexor SELDAT (10 ns.) y de la ALU (100 ns.) para estar presente a la entrada del Banco de Registros. En total, 110 ns. En paralelo se realiza el fetch de la siguiente instrucción, que tarda 110 ns. Por tanto, el tiempo requerido por este estado es de 110 ns.

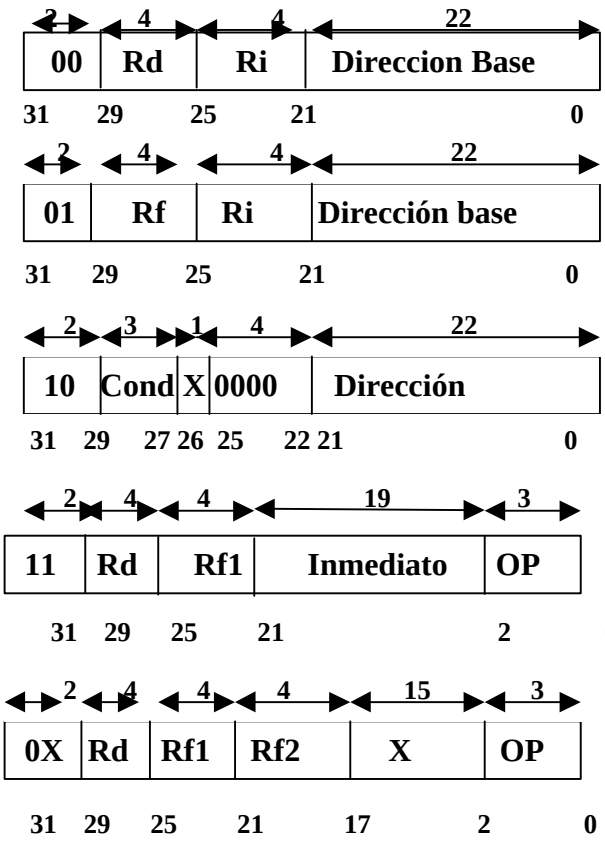
La frecuencia mínima depende del tiempo que tarde la fase de ejecución más larga, que en este caso es la fase ARIT con 140 ns. Esto implica que el período del reloj debe ser superior o igual a 140 ns, y como el período (T) es el inverso de la frecuencia (F), tenemos que:
T 140.10^{-9} s => F <= 1 / T; F <= 1 / ($140 .10^{-9}$) = 10 9 / 140 = 7.142857.1 Hz = 7.14 MHz.
F <= 7.14 MHz.

c) De los apartados anteriores se desprende que, a pesar de aumentar el numero de estados del grafo, el numero de estados (ciclos) necesario para ejecutar cada instrucción es el mismo. No obstante, la frecuencia del reloj puede ser mayor, por lo que los programas se ejecutarán más rápidamente en la MR modificada. En concreto, la MR irá 7.14 / 4.54 = 1.57 veces más rápida al ejecutar cada instrucción. Esto supone un incremento de más del 50 % de velocidad.

6.24 Como El ancho de la memoria es de 32 bits, las instrucciones pueden tener también una anchura de 32 bits. El campo de la instrucción que codifica un registro del Banco de

Registros necesita 4 bits, ya que el Banco de Registros tiene 16 registros.

a) Una posible codificación sería la siguiente, que mantiene los mismos códigos de operación (CO), los mismos códigos de condición (COND) y la misma codificación para cada instrucción aritmético-lógica, se presenta en la figura. Como puede verse, el tamaño máximo para el operando inmediato es de 19 bits.



Formato de la instrucción de Load

Formato de la instrucción de Store

Formato de las instrucciónes de Salto

Formato de las instrucciones de suma y resta con inmediato

Formato de las instrucciones de suma, resta, desplazamiento a la derecha y AND lógica

b) En cuanto al resto de la Unidad de Proceso, habría que modificar los siguientes elementos:

- El Banco de Registros debería tener 16 registros de 32 bits
- Los buses que interconectan el Banco de Registros y la memoria con otros elementos de la Unidad de Proceso deberían ser de 32 bits.
- El multiplexor SELREG debería tener 4 entradas de datos de 4 bits, una sin usar.
- Los registros IR y RA deberían ser de 32 bits.
- Los registros PC y R@ deberían ser de 22 bits.
- El sumador de direcciones y el incrementador deberían ser de 22 bits.
- El multiplexor SELDIR debería ser de 2 entradas de datos de 22 bits.
- El multiplexor SELDAT debería ser de 4 entradas de datos de 32 bits.
- La ALU debería operar con 32 bits.

c) Para aumentar el número de instrucciones aritmético-lógicas hasta 64 basta con dotar al campo OPS de 6 bits. Esto implica que el tamaño del operando inmediato pasa a ser, como máximo, de 16 bits. El resto de instrucciones no se ven alteradas.

d) Como el bit 26 del formato de las instrucciones de salto no se utiliza, podría añadirse a los bits 29-27 que codifican el campo COND. La MR tendría entonces 4 bits para codificar instrucciones de salto, lo que arroja un total de 16. Como 7 ya están definidas, se podrían definir hasta 9 más. Por ejemplo, se podrían definir instrucciones que saltasen en función de si se ha producido desbordamiento, teniendo en cuenta que los operandos sean naturales o enteros. También se podrían definir instrucciones similares a las que posee la MR, pero que produjesen el salto considerando que los operandos son naturales en lugar de enteros.

Una posible lista de instrucciones es la siguiente:

- Saltar si se produce desbordamiento con operandos enteros(V=1)
- Saltar si se produce desbordamiento con operandos naturales (C=1)
- Saltar si mayor o igual, considerando los números como operandos naturales (C=0)
- Saltar si menor o igual, considerando los números como operandos naturales (C=1 o Z=1)
- Saltar si mayor, considerando los números como operandos naturales (C=0 y Z=0)
- Saltar si menor, considerando los números como operandos naturales (C=1)
- Saltar si igual, considerando los números como operandos naturales (Z=1)
- Saltar si no igual, considerando los números como operandos naturales (Z=0)

6.25

a) Para permitir la lectura simultánea de dos registros del Banco de Registros, éste ha de tener dos multiplexores de salida, cada uno de ellos controlado por un bus de tres bits que codifica el registro a leer.

b) En la figura se muestra un Banco de Registros con dos puertos de lectura. En el contexto de la MR, un Banco de Registros con dos puertos de lectura puede ser muy interesante en la ejecución de:

- Las instrucciones aritmético/lógicas, pues se pueden leer simultáneamente los dos operandos fuente (en el caso de que ambos estén almacenados en el Banco de Registros). Si esta lectura simultánea se realiza en la fase de ejecución de la instrucción (estado ARIT), no es necesario el registro RA a la entrada de la ALU. Si la lectura simultánea se realiza en la fase de decodificación de la instrucción (estado DECO) se requiere otro registro en la entrada B de la ALU, pero conseguimos que el estado ARIT requiera menos tiempo (ver problema 6.22), el de leer del Banco de Registros el segundo operando.
- La instrucción STORE, ya que se pueden leer simultáneamente el registro fuente y el registro índice. Si esta lectura simultánea se realiza en la fase de ejecución de la instrucción (estado STORE), no es necesario el registro R@. Si la lectura simultánea se realiza en la fase de decodificación de la instrucción (estado DECO) se requiere un registro en la entrada de datos de la memoria (Min), pero conseguimos que el estado STORE requiera menos tiempo (ver problema 6.22), el de leer del Banco de Registros el registro fuente.

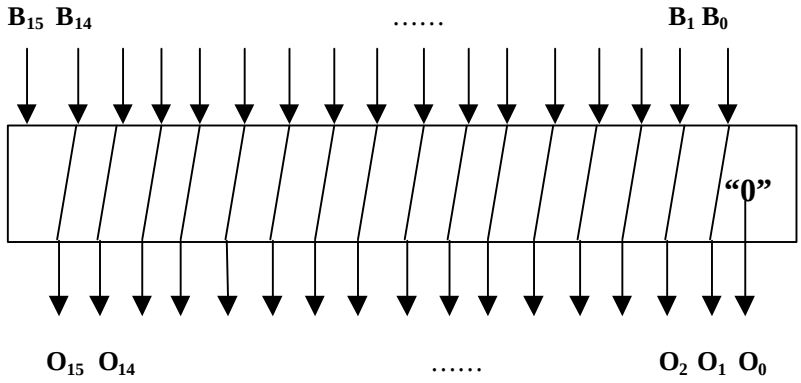
El hardware que controla el banco de registros se ha de modificar, y queda como se muestra en la siguiente figura. El control ha de generar las nuevas señales CRf_0 y CRf_1 en el estado adecuado. El formato de las instrucciones no se ha de modificar.

6.26

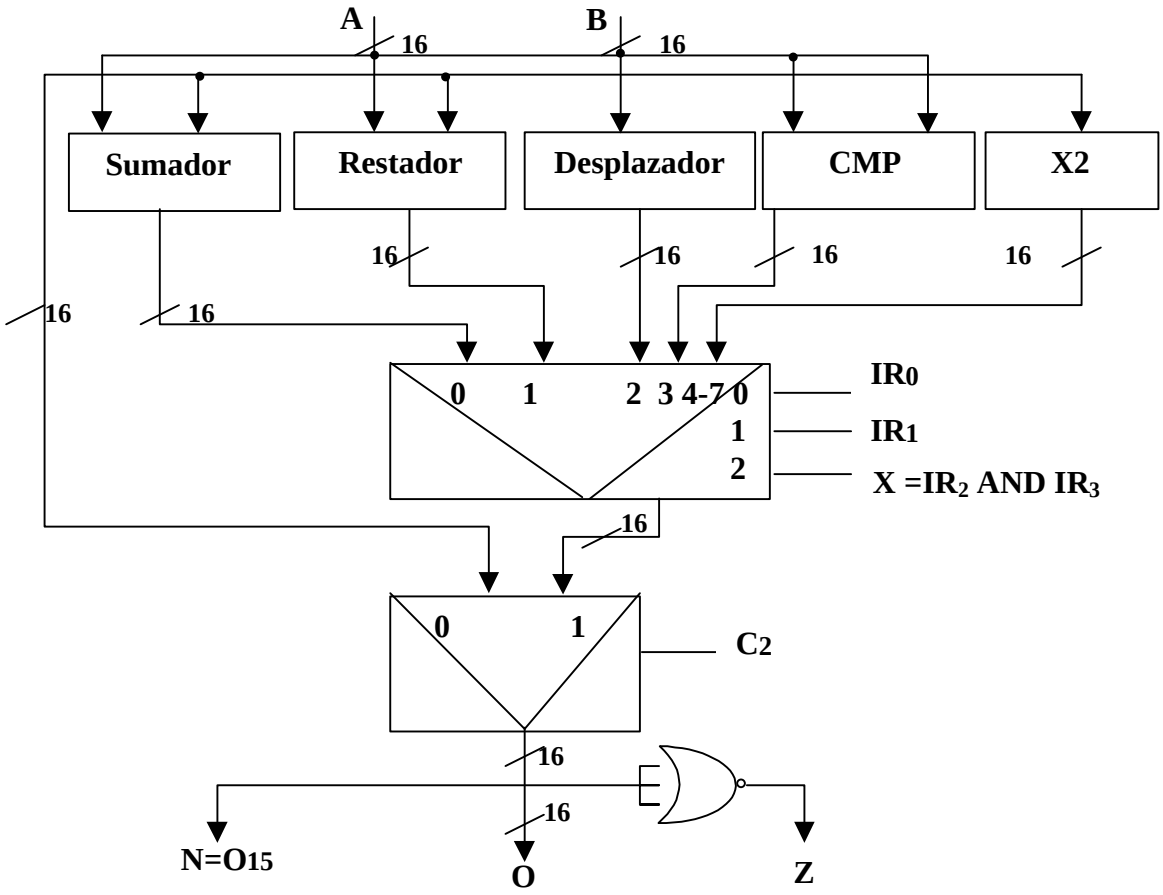
a) El formato de las instrucciones aritmético-lógicas permite codificar cuatro instrucciones que trabajen con inmediato (codificadas en los bits IR1 y IR0) y 16 instrucciones que trabajen con registros del procesador (usando los bits IR1 y IR0 y los bits IR4 y IR3, no usados actualmente). De esta manera podemos tomar la codificación siguiente para las instrucciones aritmético-lógicas que trabajan con registros:

11	**Rd**	**Rf1**	**Rf2**	**00**	**100**	**Suma**
11	**Rd**	**Rf1**	**Rf2**	**00**	**101**	**Resta**
11	**Rd**		**Rf2**	**00**	**110**	**Desplazamiento a la derecha**
11	**Rd**	**Rf1**	**Rf2**	**00**	**111**	**And-lógica**
11	**Rd**		**Rf2**	**01**	**100**	**Multiplicar por dos**

b) La multiplicación por 2 consiste básicamente en desplazar los bits del operando fuente a la izquierda, añadiendo un cero en el bit de menor peso de la salida. Así el bloque que realiza esta operación es:



Sabiendo la codificación de las instrucciones (apartado a) veamos como se ha de modificar la ALU. Al añadir la operación de multiplicar por dos, la salida de la ALU se ha de seleccionar de entre las salidas de los cinco bloques que posee (sumador, restador, desplazador, and-lógica y multiplicar por dos). De esta manera se requieren tres bits de selección, dos son los bits IR_0 y IR_1, y el tercer bit, X, es una función que identifica la instrucción MULDOS de forma única, $X = IR_2$ AND IR_3.



6.27

a) Las instrucciones se ejecutan en el siguiente orden:

1) 00h BR 3
2) 03h ADD R0, R0, R0
3) 04h BGE 1
4) 01h SUB R0, R0, R0
5) 02h BR 5

A partir de aquí, se continúa ejecutando con la instrucción que se halle en la posición 5 de memoria.

b) Calcularemos el número de ciclos de cada instrucción. Como cada fase de ejecución dura exactamente un ciclo, el número de fases de cada instrucción es el número de ciclos que tarda en ejecutarse.

BR 3	1 ciclo FETCH, 1 ciclo DECO, 1 ciclo BRANCH. Se salta a la fase DECO de la siguiente instrucción: Total 3 ciclos
ADD R0, R0, R0	1 ciclo DECO, 1 ciclo ARIT. Se salta a la fase DECO de la siguiente instrucción: Total 2 ciclos
BGE 1	1 ciclo DECO. Se produce el salto: 1 ciclo BRANCH. Se salta a la fase DECO de la siguiente instrucción: Total 2 ciclos
SUB R0, R0, R0	1 ciclo DECO, 1 ciclo ARIT. Se salta a la fase DECO de la siguiente instrucción: Total 2 ciclos
BR 5	1 ciclo DECO, 1 ciclo BRANCH: Total 2 ciclos

En total, el programa tarda 11 ciclos en ejecutarse.

c) Los valores del PC para cada fase son los siguientes (en la tabla se indican los valores del PC durante la ejecución de la fase indicada. El PC puede actualizarse al final de cada fase):

	FETCH	DECO	ARIT	BRANCH
BR 3	00H	01H		01H
ADD	-	04H	04H	
BGE 1	-	05H		05H
SUB	-	02H	02H	
BR5	-	03H		

d) Secuencia de direcciones correspondiente a la ejecución de instrucciones:
00h, 03h, 04h, 01h, 02h, 05h
Secuencia de direcciones almacenada en el PC:
00h, 01h, 04h, 05h, 02h, 03h
No se corresponde porque, en las instrucciones de salto, en el momento en que se produce el salto la dirección de la siguiente instrucción a ejecutar no se halla en el registro PC, sino en el R@.

6.28

a) El bucle comienza en la dirección 00h (comparación de entrada) y termina en la dirección 07h (salto incondicional a la comparación de entrada en el bucle). El bucle termina cuando R3 es menor que R2, por lo que se entra al mismo mientras R3 es mayor o igual que R2 (comparación efectuada en la dirección 00h).
Dentro del bucle, si R4 es igual que R2 se incrementa R2 en una unidad. De lo contrario, se le restan dos unidades a R4.

b) Como inicialmente R3=100_{10} y R2=49_{10}, el bucle se ejecuta 52 veces (100 - 49 +1)

6.29

a) Veamos el efecto de cada una de las señales:

- Con Ld_PC = 1 se permite que el registro PC se cargue con la dirección que se envía a memoria a través del bus M@, incrementada en uno.
- Con Ld_IR = 1 se permite que el registro IR se cargue con la instrucción que se envía de la memoria a través del bus Mout.
- Con PC/@ = 0 se permite que la dirección que recibe la memoria por el bus M@ sea la que se almacena en el registro PC.

Por tanto, el efecto de activar las tres señales simultáneamente es realizar el fetch de una instrucción usando el contenido del registro PC. Esto se lleva a cabo en el estado de FETCH o durante el estado ARIT del grafo simplificado.

b) Veamos el efecto de cada una de las señales:

- Con Ld_PC = 1 se permite que el registro PC se cargue con la dirección que se envía a memoria a través del bus M@, incrementada en uno.
- Con Ld_IR = 1 se permite que el registro IR se cargue con la instrucción que se envía de la memoria a través del bus Mout.
- Con PC/@ = 1 se permite que la dirección que recibe la memoria por el bus M@ sea la que se almacena en los ocho bits de menor peso del registro IR.

El efecto de activar las tres señales simultáneamente es realizar el fetch de una instrucción usando la dirección de salto almacenada en la instrucción actual, que se lleva a cabo en el estado de BRANCH.

c) Veamos el efecto de cada una de las señales:

- Con Ld_RA = 1 se permite que el registro RA se cargue con el dato que procede del Banco de Registros.
- Con CRf = 1 se permite seleccionar del Banco de Registros un registro que hace la función de RF1 (registro fuente uno) o de Ri (registro índice).

Teniendo en cuenta el valor de las dos señales, el objetivo es almacenar en RA un primer operando fuente durante la realización del estado DECO.

d) El efecto de la señal ERd = 1 es permitir que un registro del Banco de Registros se escriba con el dato presente en el bus ALU (salida de la unidad ALU). Esta acción se realiza en dos estados, en ARIT para almacenar el resultado de una operación aritmético/lógica, y en LOAD para almacenar en el Banco de Registros el dato que se ha leído en memoria.

e) Veamos el efecto de cada una de las señales:

- Con Ld_RZ = 1 y Ld_RN = 1 se permite que los indicadores de condición se actualicen con los valores calculados por la ALU.
- Con ERd = 1 se permite que un registro del Banco de Registros se escriba con el dato presente en el bus ALU (salida de la unidad ALU).
- Con OPERAR = 1 se indica que en la salida de la ALU (bus ALU) ha de estar presente el resultado de realizar una operación aritmético/lógica.
- Con CRf = 2 se permite seleccionar del Banco de Registros un registro que hace la función de RF2 (registro fuente dos).

Dado el valor de las señales, se está llevando a cabo el estado ARIT, pero no se puede concretar si el segundo operando fuente es el que se lee del Banco de Registros o es un operando inmediato.

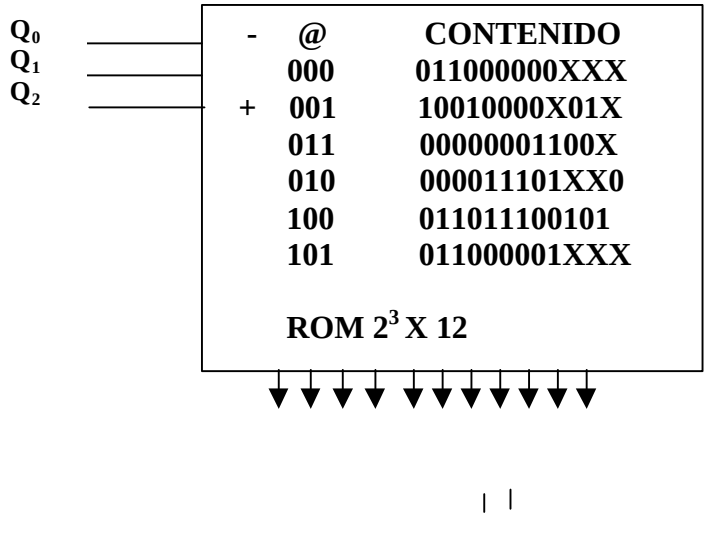
6.30

						T.Transición			T.Excitación					
Q_2	Q_1	Q_0	X_2	X_1	X_0	$Q+_2$	$Q+_1$	$Q+_0$	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	X	X	X	0	0	1	0	X	0	X	1	X
0	0	1	0	0	X	0	1	0	0	X	1	X	X	1
0	0	1	0	1	X	0	1	1	0	X	1	X	X	0
0	0	1	1	0	0	0	0	0	0	X	0	X	X	1
0	0	1	1	0	1	1	0	1	1	X	0	X	X	0
0	0	1	1	1	X	1	0	0	1	X	0	X	X	1
0	1	0	X	X	X	0	0	0	0	X	X	1	0	X
0	1	1	X	X	X	0	0	0	0	X	X	1	X	1
1	0	0	X	X	X	0	0	1	X	1	0	X	1	X
1	0	1	X	X	X	0	0	1	X	1	0	X	X	0

$J_{2=} \overline{Q}_1 Q_0 X_2 X_0 + \overline{Q}_1 Q_0 X_2 X_1$
$J_{1=} \overline{Q}_2 Q_0 X_2$
$J_{0=} \overline{Q}_1$
$K_2=1$
$K_1=1$
$K_{0=} = Q_1 + \overline{Q}_2 X_2 X_1 + \overline{Q}_2 X_2 \overline{X}_{0+} \overline{Q}_2 \overline{X}_2 \overline{X}_1$

Para diseñar la Unidad de Control de la MR como un sistema lógico secuencial cuya función de estado se realiza de forma mínima a tres niveles, primero realizaremos la tabla de transiciones que nos permitirá construir la tabla de excitación.

Función de salida con una ROM.



6.31

a) ¿Por qué se ha de realizar la decodificación de la instrucción?

Para que la Unidad de Control sepa que instrucción se está ejecutando, y por tanto qué secuencia de señales de control ha de generar para ejecutar la instrucción.

b) ¿Por qué se intenta reducir el número de estados de la Unidad de Control de la MR?

Cada estado del grafo de estados se lleva a cabo en un ciclo de reloj del procesador. Por tanto, el número de estados en que se divide la ejecución de una instrucción equivale al

número de ciclos en que tarda en ejecutarse la instrucción.
La duración del ciclo de reloj del procesador es la del estado que tarda más tiempo en realizarse. Por tanto, al diseñar el grafo de estados, no se ha de pensar tanto en reducir el número de estados aumentando el contenido semántico de algún estado (que podría incrementar el ciclo del procesador), sino en encontrar un equilibrio entre el número de estados y la duración del ciclo del procesador.

6.32

Ver solución al Problema 6.25.

6.33

a) Si la ALU de la MR, además de los indicadores de condición Z y N, también calculara los bits:
- V: desbordamiento para enteros
- C: desbordamiento para naturales

se podría considerar la incluisión de, al menos, cuatro instrucciones de salto más:

- BV Saltar si desbordamiento para enteros: se produce el salto si V=1.
- BNV Saltar si no desbordamiento para enteros: se produce el salto si V=0.
- BC Saltar si desbordamiento para naturales: se produce el salto si C=1.
- BNC Saltar si no desbordamiento para naturales: se produce el salto si C=0.

b) Conservando el formato actual de las instrucciones de salto, sólo se puede incluir una instrucción más, y tendría por código COND=100.

Supongamos que incluimos la instrucción BV, entonces el valor de la señal Cond se puede expresar con la función:

$$Cond = \overline{IR_{13}}..\overline{IR_{12}}.\overline{IR_{11}}. + (\overline{IR_{13}}..\overline{IR_{12}}.IR_{11}).Z + (\overline{IR_{13}}..IR_{12}.\overline{IR_{11}}).N + (\overline{IR_{13}}..IR_{12}.IR_{11}).(N+Z) +$$
$$+ (IR_{13}.\overline{IR_{12}}.IR_{11}).\overline{Z} + (IR_{13}..IR_{12}.\overline{IR_{11}}).\overline{N} + (IR_{13}.IR_{12}.IR_{11})(\overline{N+Z}.)$$

Y simplificando por Karnaugh a tres niveles obtenemos la expresión:

$$Cond = .\overline{IR_{12}}.\overline{IR_{11}}. + \overline{IR_{13}}..IR_{11}.Z + \overline{IR_{13}}..IR_{12}.N + IR_{13}.\overline{IR_{12}}.\overline{Z} + IR_{13}..\overline{IR_{11}}.\overline{N} + IR_{13}.\overline{N.Z}$$

Para incluir las cuatro instrucciones sería necesario cambiar el formato de las instrucciones de salto. Con objeto de no modificar ninguna instrucción, propondremos un conjunto de 8 instrucciones de salto que permitan manejar convenientemente los cuatro indicadores de condición. Las instrucciones BZ, BNZ, BL y BG permiten realizar todas las operaciones que realizaban las siete instrucciones de salto que antes tenía la MR. Por ejemplo, para saltar si menor o igual se puede usar la instrucción BG intercambiando los términos de la comparación:

- BZ Saltar si igual: se produce el salto si Z=1.
- BNZ Saltar si distinto: se produce el salto si Z=0.
- BL Saltar si menor: se produce el salto si N=1.
- BG Saltar si mayor: se produce el salto si N=0 y Z=0.
- BV Saltar si desbordamiento para enteros: se produce el salto si V=1.
- BNV Saltar si no desbordamiento para enteros: se produce el salto si V=0.
- BC Saltar si desbordamiento para naturales: se produce el salto si C=1.
- BNC Saltar si no desbordamiento para naturales: se produce el salto si C=0.

6.34

Iniciaremos el programa con la declaración de las variables "a" y "b" mediante la utilización de dos directivas de reserva de memoria:

```
a:    .RW 1
b:    .RW 1
```

A continuación se iniciará el código en lenguaje ensamblador (utilizando la directiva de inicio de programa) y la inicialización de las variables. Para ello, supondremos que las variables "a" y "b" se almacenarán temporalmente en los registros R1 y R2 respectivamente. La constante +16 supera el rango de los valores inmediatos que se pueden representar (-16,...,0,...15), por lo que deberá inicializarse con dos instrucciones de suma o bien mediante una de resta (usando el hecho de que el rango en los números negativos es mayor en la representación en complemento a dos).

```
        .BEGIN inicio
inicio: ADDI R0, #13, R1
        SUBI R0, #-16, R2
```

El bucle *mientras* se construirá con dos instrucciones. La primera calculará la condición del salto y la segunda decidirá si se debe saltar o no:

```
mientras:   SUBI R1, #10, R0
            BLE fmientras
```

Las siguientes instrucciones implementarán el cuerpo del bucle *mientras*. Se utilizará una instrucción de salto incondicional para retornar a la instrucción del calculo de la condición del bucle:

```
            SUBI R1, #1, R1
            ADDI R2, #2, R2
            BR mientras
```

A continuación indicaremos que se ha finalizado la ejecución del bucle *mientras* y que se inicia la ejecución de la sentencia condicional *si*. Para ello, se situarán las etiquetas correspondientes y las instrucciones necesarias para el cálculo de la condición:

```
fmientras:  SUB R1, R2, R0
            BGE sino
```

Si la condición de la instrucción *si* se cumple, hay que ejecutar la instrucción SWAP (implementada con tres instrucciones aritméticas), mientras que en caso contrario se ejecutará la instrucción del *sino*. En ambos casos, después de ejecutar las instrucciones correspondientes se debe llegar al final del programa:

```
si:     SUB R1, R2, R2
        SUB R1, R2, R1
        ADD R1, R2, R2
        BR fsi
sino:   SUBI R1, #1, R2
fsi:
```

Finalmente, el resultado almacenado en los registros debe guardarse en la memoria, que es donde residen las variables:

```
        STORE R1, a(R0)
        STORE R2, b(R0)
        .END
```

A continuación se presenta el programa descrito:

```
a:          .RW 1
b:          .RW 1
            .BEGIN inicio
inicio:     ADDI R0, #13, R1
            SUBI R0, #-16, R2
mientras:   SUBI R1, #10, R0
            BLE fmientras
            SUBI R1, #1, R1
            ADDI R2, #2, R2
            BR mientras
fmientras:  SUB R1, R2, R0
            BGE sino
si:         ADD R1, R2, R2
            SUB R1, R2, R2
            SUB R1, R2, R1
            BR fsi
sino:       SUBI R1, #1, R2
fsi:        STORE R1, a(R0)
            STORE R2, b(R0)
            .END
```

Una vez completado el programa en lenguaje ensamblador, presentamos a continuación su traducción a lenguaje máquina y la tabla de símbolos generada por el programa ensamblador:

L. ensamblador	Dirección	L. Máquina	Hexadecimal
ADDI R0, #13, R1	02h:	11 001 000 01101 000	C868h
SUBI R0, #-16, R2	03h:	11 010 000 10000 001	D081h
SUBI R1, #10, R0	04h:	11 000 001 01010 001	C151h
BLE fmientras	05h:	10 011 000 00001001	9809h
SUBI R1, #1, R1	06h:	11 001 001 00001 001	C909h
ADDI R2, #2, R2	07h:	11 010 010 00010 000	D210h
BR mientras	08h:	10 000 000 00000100	8004h

SUB R1, R2, R0	09h:	11 000 001 010 00 101	C145h
BGE sino	0Ah:	10 110 000 00001111	B00Fh
ADD R1, R2, R2	0Dh:	11 010 001 010 00 100	D144h
SUB R1, R2, R2	0Bh:	11 010 001 010 00 101	D145h
SUB R1, R2, R1	0Ch:	11 001 001 010 00 101	C945h
BR f_si	0Eh:	10 000 000 00010000	8010h
SUBI R1, #1, R2	0Fh:	11 010 001 00001 001	D109h
STORE R1, A(R0)	10h:	01 001 000 00000000	6800h
STORE R2, B(R0)	11h:	01 010 000 00000001	6001h

Tabla de Símbolos:

Símbolo	Valor
a:	00h
b:	01h
inicio:	02h
mientras:	04h
fmientras:	09h
sino:	0Fh
fsi:	10h

6.35

```
a:          .RW 1;                  a -> R1
b:          .RW 1;                  b -> R2
c:          .RW 1;                  c -> R3
            .BEGIN inicio;    R4 será una variable auxiliar
inicio:     ADDI R0, #13, R1;       a
            ADDI R0, #1, R2;        b
            ADD R0, R0, R3;         c
mientras:   SUBI R1, #10, R0
            BG cuerpo
            SUBI R2, #10, R4
```

```
            SUBI R4, #10, R0
            BGE fmientras
cuerpo:     SUBI R1, #11, R0
            BGE siguiente
            SUBI R2, #10, R4
            SUBI R4, #8, R0
            BLE siguiente
            ADDI R3, #3, R3
siguiente:  SUB R1, R2, R1
            BR mientras
fmientras:  STORE R1, a(R0)
            STORE R2, b(R0)
            STORE R3, c(R0)
            .END
```

ADDI R0, #13, R1	03h:	11 001 000 01101 000	C868h
ADDI R0, #1, R2	04h:	11 010 000 00001 000	D008h
ADD R0, R0, R3	05h:	11 011 000 000 00 100	D804h
SUBI R1, #10, R0	06h:	11 000 001 01010 001	C151h
BG cuerpo	07h:	10 111 000 0000 1011	B80Bh
SUBI R2, #10, R4	08h:	11 100 010 01010 001	E251h
SUBI R4, #10, R0	09h:	11 000 100 01010 001	C451h
BGE fmientras	0Ah:	10 110 000 0001 0011	B813h
SUBI R1, #11, R0	0Bh:	11 000 001 01011 001	C159h
BGE siguiente	0Ch:	10 110 000 0001 0001	B011h
SUBI R2, #10, R4	0Dh:	11 100 010 01010 001	E251h
SUBI R4, #8, R0	0Eh:	11 000 100 01000 001	C441h
BLE siguiente	0Fh:	10 011 000 0001 0001	9811h
ADDI R3, #3, R3	10h:	11 011 011 00011 000	DB18h
SUB R1, R2, R1	11h:	11 001 001 01000 101	C945h
BR mientras	12h:	10 000 000 0000 0110	8006h
STORE R1, A(R0)	13h:	01 001 000 0000 0000	4800h
STORE R2, B(R0)	14h:	01 010 000 0000 0001	5001h
STORE R3, C(R0)	15h:	01 011 000 0000 0010	5802h

Tabla de Símbolos

Símbolo	Valor
a:	00h
b:	01h
c:	02h
inicio:	03h
mientras:	06h
cuerpo:	0Bh
siguiente:	11h
fmientras:	13h

6.36

a)

```
a:          .DW 10
b:          .DW 15
mcd:        .RW 1
            .BEGIN inicio
inicio:     LOAD a(R0), R1; variable a -> R1
            LOAD b(R0), R2; variable b -> R2
mientras:   SUB R1, R2, R0
            BEQ fmientras
            BLE sino
            SUB R1, R2, R1
            BR mientras
sino:       SUB R2, R1, R2
            BR mientras
fmientras:  STORE R1, mcd(R0); "a" y "b" no se escriben en memoria
            .END
```

b)

LOAD a(R0), R1 12h: 00 001 000 0000 1111 080Fh

LOAD b(R0), R2	13h:	00 010 000 0001 0000	1010h
SUB R1, R2, R0	14h:	11 000 001 01000 101	C145h
BEQ fmientras	15h:	10 001 000 0001 1011	881Bh
BLE sino	16h:	10 011 000 0001 1001	9819h
SUB R1, R2, R1	17h:	11 001 001 010 00 101	C945h
BR mientras	18h:	10 000 000 0001 0100	8014h
SUB R2, R1, R2	19h:	11 010 010 001 00 101	D225h
BR mientras	1Ah:	10 000 000 0001 0100	8014h
STORE R1, mcd(R0)	1Bh:	01 001 000 0001 0001	4811h

Tabla de Símbolos

Símbolo	Valor
a:	0Fh
b:	10h
mcd:	11h
inicio:	12h
mientras:	14h
sino:	19h
fmientras:	1Bh

6.37

a)

Tabla de Símbolos

Símbolo	Valor
adrA:	10h
adrB:	14h
adrV:	17h

b)

LOAD adrA+1(R0), R1 R1 <- Y = 12	00 001 000 0001 0001	0811h
LOAD adrB-4(R0), R2 R2 <- X = 8	00 010 000 0001 0000	1010h
ADD R1, R2, R3	11 011 001 010 00 100	D944h

R3 <- R1 + R2 = 20

STORE R3, adrV+N(R2) 01 011 010 0001 1101 5A1Dh
M[23+6+8] = M[37] <- 20

c)

- adrV[14] = N

```
ADDI R0, #N, R1
ADDI R0, #14, R2
STORE R1, adrV(R2)
```

- adrV[X+Y] = adrV[10] + M[2*N]

```
ADDI R0, #10, R2
LOAD adrV(R2), R3
ADDI R0, #N, R2
ADD R2, R2, R2
LOAD 0(R2), R2
ADD R2, R3, R2
ADDI R0, #X, R3
ADDI R3, #Y, R3
STORE R2, adrV(R3)
```

6.38

```
programa cuadrado;
var a, a2, i: entero;
i := 0;
a2 := 0;
mientras (a < i) hacer
        a2 := a2 + a;
        i := i + 1;
```

fmientras
fprograma

```
a:          .RW 1;              variable a -> R1
a2:         .RW 1;              variable a2 -> R2
            .BEGIN inicio;      variable i -> R3
inicio:     LOAD a(R0), R1
            ADD R0, R0, R2
            ADD R0, R0, R2
mientras:   SUB R1, R3, R0
            BGE fmientras
            ADD R1, R2, R2
            ADDI R3, #1, R3
            BR mientras
fmientras:  STORE R2, a2(R0)
            .END
```

LOAD A(R0), R1	02h:	00 001 000 0000 0000	0800h
ADD R0, R0, R2	03h:	11 010 000 000 00 100	D004h
ADD R0, R0, R3	04h:	11 011 000 000 00 100	D804h
SUB R1, R3, R0	05h:	11 000 001 011 00 101	C165h
BGE fmientras	06h:	10 110 000 0000 1010	B00Ah
ADD R1, R2, R2	07h:	11 010 001 010 00 100	D144h
ADDI R3, #1, R3	08h:	11 011 011 00001 000	DB08h
BR mientras	09h:	10 000 000 0000 0101	8005h
STORE R2, a2(R0)	0Ah:	01 010 000 0000 0001	5001h

Tabla de Símbolos:

Símbolo	Valor
a:	00h
a2:	01h
inicio:	02h
mientras:	05h
fmientras:	0Ah

6.40

```
programa letras_a;
var a, punto, numero, i : entero;
var frase[0.. 124]: entero;
i := 0;
a := 61h;
punto := 2Eh;
numero := 0;
mientras (frase[i] != punto) hacer
        si (frase[i] == a) hacer numero := numero +1 fsi;
        i := i + 1;
fmientras
fprograma
```

```
a:          .DW 61h;          a      ->     R1
punto:      .DW 2Eh;          punto  ->     R2
numero:     .RW 1;            frase[i]->    R3
frase:      .RW 125;          numero ->     R4
            .BEGIN inicio;    i      ->     R5
inicio:     LOAD a(R0), R1
            LOAD punto(R0), R2
            ADD R0, R0, R5
            ADD R0, R0, R4
mientras:   LOAD frase(R5), R3
            SUB R3, R2, R0
            BEQ fmientras
            SUB R3, R1, R0
            BNE fsi
            ADDI R4, #1, R4
fsi:        ADDI R5, #1, R5
            BR mientras
fmientras:  STORE R4, numero(R0)
            .END
```

```
LOAD a(R0), R1          81h:          00 001 000 0000 0000          0800h
```

LOAD punto(R0), R2	82h:	00 010 000 0000 0001	1001h
ADD R0, R0, R5	83h:	11 101 000 000 00 100	E806h
ADD R0, R0, R4	84h:	11 100 000 000 00 100	E006h
LOAD frase(R5), R3	85h:	00 011 101 0000 0011	1D03h
SUB R3, R2, R0	86h:	11 000 011 010 00 101	C365h
BEQ fmientras	87h:	10 001 000 10000 1101	888D
SUB R3, R1, R0	88h:	11 000 011 001 00 101	C325h
BNE fsi	89h:	10 101 000 1000 1011	A88Bh
ADDI R4, #1, R4	8Ah:	11 100 100 00001 000	E608h
ADDI R5, #1, R5	8Bh:	11 101 101 00001 000	ED08h
BR mientras	8Ch:	10 000 000 1000 0101	8085h
STORE R4, num(R0)	8Dh:	01 100 000 0000 0010	6002h

Tabla de Símbolos:

Símbolo	Valor
a:	00h
punto:	01h
numero:	02h
frase:	03h
inicio:	81h
mientras:	85h
fsi:	8Bh
fmientras:	8Dh

6.41

programa vector;
const n=3

```
var suma, max, min, numero, i : entero;
var vector[0.. n]: entero;
i := 1;
suma := vector[0];
max := vector[0];
min := vector[0];
mientras (n > i) hacer
        numero := vector[i];
        i := i + 1;
       suma := suma + numero;
       si (numero > max) entonces max := numero; fsi
       si (numero < min) entonces min := numero; fsi
fmientras
fprograma
```

```
suma:       .RW 1
max:        .RW 1
min:        .RW 1
            N=3
n:          .DW N
vector:     .RW N
            .BEGIN inicio
inicio:     LOAD n(R0), R2;         n ->R2
            ADDI R0, #1, R1;        i -> R1
            LOAD vector(R0), R6;    numero -> R6
            ADD R6, R0, R3;         suma -> R3
            ADD R6, R0, R4;         max -> R4
            ADD R6, R0, R5;         min -> R5
mientras:   SUB R2, R1, R0
            BLE fmientras
            LOAD vector(R1), R6
            ADDI R1, #1, R1
            ADD R3, R6, R3



amax:       SUB R6, R4, R0
```

```
            BLE amin
            ADD R6, R0, R4
amin:       SUB R6, R5 R0
            BGE mientras
            ADD R6, R0, R5
            BR mientras
fmientras:  STORE R3, suma(R0)
            STORE R4, max(R0)
            STORE R5, min(R0)
            .END
```

LOAD n(R0), R2	3Ah:	00 010 000 0000 0011	1003h
ADDI R0, #1, R1	3Bh:	11 001 000 00001 000	C808h
LOAD vector(R0), R6	3Ch:	00 110 000 0000 0100	3004h
ADD R6, R0, R3	3Dh:	11 011 110 00000 100	DE04h
ADD R6, R0, R4	3Eh:	11 100 110 00000 100	E604h
ADD R6, R0, R5	3Fh:	11 101 110 00000 100	EE04h
SUB R2, R1, R0	40h:	11 000 010 001 00 101	C225h
BLE fmientras	41h:	10 011 000 0100 1100	984Ch
LOAD vector(R1), R6	42h:	00 110 001 00000100	3104h
ADDI R1, #1, R1	43h:	11 001 001 00001 000	C908h
ADD R3, R6, R3	44h:	11 011 011 11000 100	DBC4h
SUB R6, R4, R0	45h:	11 000 110 100 00 101	C685h
BLE amin	46h:	10 011 000 0100 1000	9848h
ADD R6, R0, R4	47h:	11 100 110 00000 100	E604h
SUB R6, R5 R0	48h:	11 000 110 10100 101	C6A5h
BGE mientras	49h:	10 110 000 0100 0000	B040h
ADD R6, R0, R5	4Ah:	11 101 110 00000 100	EE04h
BR mientras	4Bh:	10 000 000 0100 0000	8040h
STORE R3, suma(R0)	4Ch:	01 011 000 0000 0000	5800h
STORE R4, max(R0)	4Dh:	01 100 000 0000 0001	6001h
STORE R5, min(R0)	4Eh:	01 101 000 0000 0010	6802h

Tabla de Símbolos

Símbolo	Valor
suma:	00h
max:	01h
min:	02h
n:	03h
vector:	04h
inicio:	3Ah
mientras:	40h
amax:	45h
amin:	48h
fmientras:	4Ch

6.42

```
programa suma_vector;
var a[0..9], b[0..9], c[0..9] : entero;
var i : entero;
i := 0;
mientras (i < 10) hacer
        c[i] := a[i] + b[i];
        i := i + 1;
fmientras
fprograma
```

```
a:          .RW 10 ;  a[i] -> R2
b:          .RW 10 ;  b[i] -> R3
c:          .RW 10
            .BEGIN inicio
inicio:     ADD R0, R0, R1; i -> R1
mientras:   SUB R1, #10, R0
            BGE fmientras
            LOAD a(R1), R2
            LOAD b(R1), R3
            ADD R2, R3, R3
            STORE R3, c(R1)
            ADDI R1, #1, R1
```

```
            BR mientras
fmientras:  .END
```

ADD R0, R0, R1	80h:	11 001 000 000 00 100	C804h
SUB R1, #10, R0	81h:	11 000 001 01010 001	C151h
BGE fmientras	82h:	10 110 000 1000 1001	B089h
LOAD a(R1), R2	83h:	00 010 001 0000 0000	1100h
LOAD b(R1), R3	84h:	00 011 001 0000 1010	190Ah
ADD R2, R3, R3	85h:	11 011 010 011 00 100	DA64h
STORE R3, c(R1)	86h:	01 011 001 0001 0100	5914h
ADDI R1, #1, R1	87h:	11 001 001 00001 000	C908h
BR mientras	88h:	10 000 000 1000 0001	8081h

Tabla de Símbolos

Símbolo	Valor
a:	80h
b:	0Ah
c:	14h
inicio:	80h
mientras:	81h
fmientras:	89h

6.43

a) CLR Rd

```
.def CLR $1
ADD R0, R0, $1
.enddef
```

b) CLR A(Ri)

```
.def CLR $d1
STORE R0, $d1
.enddef
```

c) INC Rd

```
.def INC $1
ADDI $1, #1, $1
.enddef
```

d) DEC Rd

```
.def DEC $1
SUBI $1, #1, $1
.enddef
```

e) ADD3 Rf1, Rf2, Rd

```
.def AND3 $1, $2, $3
ADD $1, $3, $3
ADD $2, $3, $3
.enddef
```

f) ASL Rd

```
.def ASL $1
ADD $1, $1, $1
.enddef
```

g) MOV Rf, Rd

```
.def MOV $1, $2
ADD $1, R0, $2
.enddef
```

h) MOV Rf, A(Ri)

```
.def MOV $1, $d1
STORE $1, $d1
.enddef
```

i) MOV A(Ri), B(Rj)

```
.def MOV $d1, $d2
LOAD $d1, R7
STORE R7, $d1
.enddef
```

j) SWAP Rf1, Rf2

```
.def SWAP $1, $2
ADD R0, $1, R7
```

```
                                ADD $2, R0, $1
                                ADD R7, R0, $2
                                .enddef
```

k) SWAP A(Ri), B(Rj)

```
                                .def SWAP $d1, $d2
                                LOAD $d1, R6
                                LOAD $d2, R7
                                STORE R7, $d1
                                STORE R6, $d2
                                .enddef
```

l) ASRN #n, Rd

```
.def ASRN $n1, $1
        ADD R0, $n1, R7
basr:    BEQ fasr
        ASR $1, $1
        SUBI R7, #1, R7
        BR basr
.enddef
```

m) ASLN #n, Rd

```
.def ASLN $n1, $1
        ADD R0, $n1, R7
basr:   BEQ fasl
        ADD $1, $1, $1
        SUBI R7, #1, R7
        BR basl
.enddef
```

n) INC A(Ri)

```
.def INC $d1
LOAD $d1, R7
ADD R7, #1, R7
STORE R7, $d1
.enddef
```

6.44

```
programa suma_vector;
var suma, max, min, n : entero;
var vector[0..N] : entero;
```

```
var i : entero;
i := 0;
suma := 0;
mientras (i < n) hacer
        si (vector[i] < max) y (vector[i] > min) entonces
                suma := suma + vector[i];
        fsi;
        i := i + 1;
fmientras;
fprograma
```

```
suma:     .RW 1                 ; n -> R5
max:      .RW 1                 ; i -> R6
min:      .RW 1
n:        .RW 1
vector:   .RW N
          .BEGIN inicio
inicio:   LOAD max(R0), R3;   max -> R3
          LOAD min(R0), R4;    min -> R4
          LOAD n(R0), R5
          ADD R0, R0, R1;       suma -> R1
          ADD R0, R0, R6
mientras: SUB R6, R5, R0
          BGE fmientras
          LOAD vector(R6), R2; elemento del vector -> R2
          SUB R2, R3, R0
          BGE fsi
          SUB R2, R4, R0
          BLE fsi
          ADD R1, R2, R1
fsi:      ADDI R6, #1, R6
          BR mientras
mientras: STORE R1, suma(R0)
          .END
```

LOAD max(R0), R3	3Ah:	00 011 000 0000 0001	1801h
LOAD min(R0), R4	3Bh:	00 100 000 00000010	2002h
LOAD n(R0), R5	3Ch:	00 101 000 0000 0011	2803h
ADD R0, R0, R1	3Dh:	11 001 000 000 00 100	C804h
ADD R0, R0, R6	3Eh:	11 110 000 000 00 100	F004h
SUB R6, R5, R0	3Fh:	11 000 110 101 00 101	C6A5h
BGE fmientras	40h:	10 110 000 0011 1111	D03Fh
LOAD vector(R6), R2	41h:	00 010 110 0000 0100	1604h
SUB R2, R3, R0	42h:	11 000 010 011 00 101	C265h
BGE fsi	43h:	10 110 000 0100 0111	D047h
SUB R2, R4, R0	44h:	11 000 010 100 00 101	C285h
BLE fsi	45h:	10 011 000 0100 0111	9847h
ADD R1, R2, R1	46h:	11 001 001 010 00 100	C944h
ADDI R6, #1, R6	47h:	11 110 110 00001 000	F608h
BR mientras	48h:	10 000 000 0011 1111	803Fh
STORE R1, suma(R0)	49h:	01 001 000 0000 0000	4800h

Tabla de Símbolos

Símbolo	Valor
suma:	00h
max:	01h
min:	02h
n:	03h
vector:	04h
inicio:	3Ah
mientras:	3Fh
fsi:	47h
fmientras:	49h