



PRÁCTICA FINAL 2022


ESTRUCTURA DE COMPUTADORES I

1º CURSO EN INGENIERÍA INFORMÁTICA

JOAN MARTORELL COLL

43233750Y

joan.martorell4@estudiant.uib.cat



ÍNDICE

ÍNDICE	1
INTRODUCCIÓN	2
EXPLICACIÓN GENERAL	3
ÁRBOL DE DECODIFICACIÓN	4
SUBROUTINAS.....	5
REGISTROS DEL 68K.....	5
PRUEBAS	6
CONCLUSIONES	7
CÓDIGO FUENTE	8

INTRODUCCIÓN

Un emulador permite que un ordenador pueda ejecutar un programa escrito para una máquina diferente. En esta práctica final se implementará, en lenguaje ensamblador del 68K, un programa que emule la ejecución de programas escritos para una máquina elemental dada.

Estos programas deberán estar escritos usando el conjunto de instrucciones de la máquina en cuestión, y el emulador deberá funcionar para cualquier programa que respete dicho conjunto. Para llevar a cabo esta emulación, todas las partes de la máquina elemental se definirán en la memoria del 68K. Por un lado, el programa debe ser capaz de leer de la memoria del 68K una secuencia de instrucciones codificadas como words, de acuerdo al conjunto de instrucciones de la propia máquina elemental. Para cada una de estas instrucciones, el programa aplicará un proceso de decodificación para determinar de qué instrucción del conjunto se trata y, a continuación, emulará su ejecución. Debido a que la máquina elemental dada está diseñada siguiendo una arquitectura Von Neumann, junto con las instrucciones que forman el programa también se almacenarán los datos. Por otro lado, además de la memoria para el programa y los datos, el emulador también reservará una serie de posiciones de memoria en el 68K para representar todos los registros de la máquina elemental a emular, así como un registro de estado que contendrá los flags.

La máquina que se emulará en esta práctica se llama JARVIS (Just A Rather Very Intelligent System). Tanto los registros como su conjunto de instrucciones son de 16 bits. La JARVIS posee los siguientes registros:

- B0 y B1, registros de direcciones, que se utilizan en algunas instrucciones para acceder a memoria usando un modo de direccionamiento indexado;
- R2, R3, R4 y R5, que son de propósito general y se utilizan en operaciones de tipo ALU, ya sea como operando fuente o como operando destino;
- T6 y T7, que se utilizan como interfaz con la memoria, además de poder ser empleados en operaciones de tipo ALU como operando.

NOTA: para facilitar la distinción entre todo lo relativo a la JARVIS y lo relativo al 68K, se añadirá a partir de ahora el prefijo “e” a todo lo que pertenezca a la primera. Así, el registro Ri de la máquina emulada lo denotaremos por ERi, los programas de la máquina emulada los denotaremos por eprogramas, etc.

EXPLICACIÓN GENERAL

El programa emulador que se escribirá será un bucle que llevará a cabo los siguientes pasos para cada instrucción del eprograma indicado en EMEM:

- **FETCH**

Cogemos la primera componente del vector EMEM, la cual contiene un código en hexadecimal que representa la codificación de una instrucción. Este código incluye el Opcode (Operation Code) en sus bits más significativos (15, 14, 13, 12 o 11 dependiendo de la instrucción) y los operandos en los bits menos significativos de dicho código. Las direcciones de las componentes de este vector se guardan en el registro A0, y el contenido en el EIR. Cuando acaban las tres fases (Fetch, Decodificación y Ejecución) de una instrucción, utilizamos el registro EPC para apuntar a la siguiente instrucción del eprograma introducido a la JARVIS y se repite el proceso de realización de una instrucción. Antes de saltar a la fase de decodificación guardamos en la pila un word para el código de la instrucción y un word para el resultado de la decodificación.

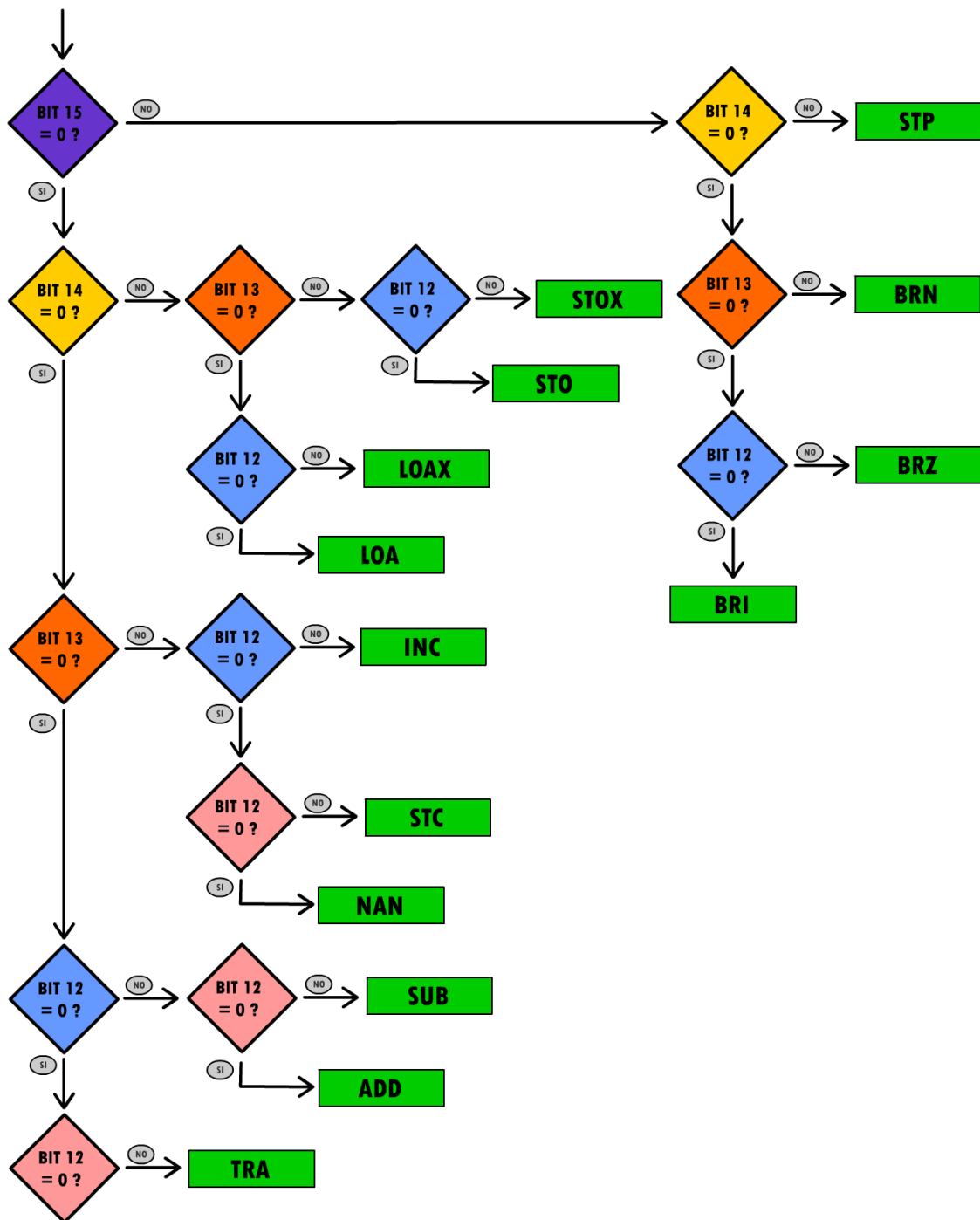
- **DECODIFICACIÓN**

Luego, cuando ya tenemos todo preparado en la pila en orden correcto, se realiza un JSR (Jump to Sub-Routine) a la fase de Decodificación, la cual nos permite ver ante que instrucción nos encontramos a través del “Árbol de Decodificación”. El código pasa por una rutina de decodificación que analiza los bits del Opcode para saber de qué instrucción se trata y luego, al identificar de que instrucción se trata, guardaremos el número que le corresponde en la pila en el word reservado previamente. El número guardado en la pila se utilizará al final de la fase de Decodificación para acceder a la JUMPLIST y hacer un salto a la Ejecución de la instrucción.

- **EJECUCIÓN**

Se analizan los bits menos significativos del código de la instrucción, los cuales indican los registros y/o valores utilizados en la instrucción. En esta fase, la máquina pasa por subrutinas para actualizar los Flags de la JARVIS y para saber qué registros utilizará la máquina dependiendo de la instrucción. El tipo de subrutinas utilizadas en esta máquina serán de usuario. Para concluir esta fase, al acabar las operaciones de una instrucción ejecutada, se realizará un Branch incondicional a la fase Fetch.

ÁRBOL DE DECODIFICACIÓN



SUBROUTINAS

SUBROUTINA	FUNCIÓN
DET_A	Subrutina de usuario que sirve para detectar el valor a en el EIR. Al acabar la subrutina, hace un salto incondicional a la subrutina DET_REG, la cual se encargará de detectar de que eregistro de la máquina JARVIS se trata.
DET_B	Subrutina de usuario que sirve para detectar el valor b en el EIR. Al acabar la subrutina, hace un salto incondicional a la subrutina DET_REG, la cual se encargará de detectar de que eregistro de la máquina JARVIS se trata.
DET_K	Subrutina de usuario que sirve para detectar el valor k en el EIR. Al acabar la subrutina, guarda el valor con extensión de signo en el registro D2.
DET_M	Subrutina de usuario que sirve para detectar el valor m en el EIR. Al acabar la subrutina, guarda el valor en el registro D2.
DET_REG	Subrutina de usuario que sirve para detectar de que eregistro se trata. Mira el registro D2, que anteriormente habrá sido usado como salida de DET_A o DET_B, y redirige a la subrutina pertinente del eregistro.
DET_Bi	Subrutina de usuario que sirve para detectar el valor de Bi en el EIR. Luego, redirige a la subrutina pertinente del eregistro.
DET_Tj	Subrutina de usuario que sirve para detectar el valor de Ti en el EIR. Luego, redirige a la subrutina pertinente del eregistro.
ES_XX	Cada una de estas subrutinas de usuario guarda la dirección de su eregistro pertinente en A2.
FLAG_X	Cada una de estas subrutinas de usuario copia su pertinente flag del 68k a la máquina JARVIS.
DECOD	Esta subrutina de librería se utiliza para la decodificación.

REGISTROS DEL 68K

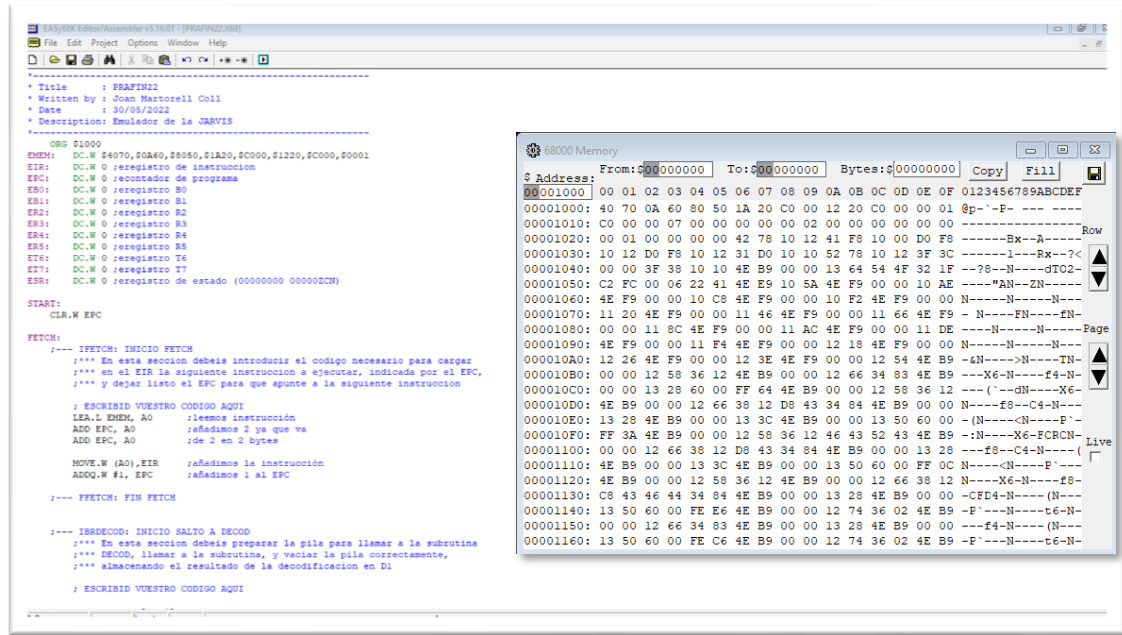
Registros de datos	FUNCIÓN
D0	Utilizado para modificar el EPC
D1	ID de la instrucción
D2	Para detectar el valor "a", "b", "k", "m"
D3 + D4 + D6	Para operaciones en la ejecución de las instrucciones

Registros de dirección	FUNCIÓN
A0	Componente del vector
A1	Para la JUMPLIST
A2	Dirección del eregistro
A3	Para operaciones en la ejecución de las instrucciones
A7	Es el Stack Pointer

PRUEBAS

PRUEBA 1:

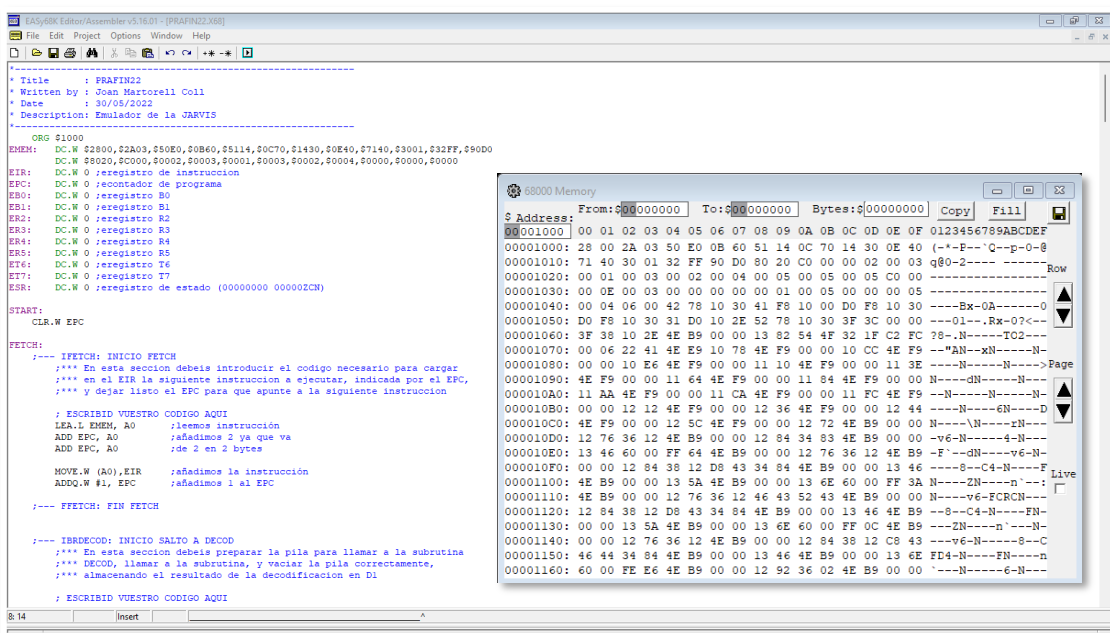
EMEM: \$4070,\$0A60,\$8050,\$1A20,\$C000,\$1220,\$C000,\$0001



Para que el programa se ejecute correctamente en la posición de memoria del 68K correspondiente a ER2 (en este caso, @1018Hex) debería contener el valor 2Dec = 0002Hex. ✓

PRUEBA 2:

EMEM: \$2800,\$2A03,\$50E0,\$0B60,\$5114,\$0C70,\$1430,\$0E40,\$7140,\$3001,\$32FF,\$90D0,\$8020,\$C000,\$0002,\$0003,\$0001,\$0003,\$0002,\$0004,\$0000,\$0000,\$0000



Para que el programa se ejecute correctamente en las posiciones de memoria del 68K @1028, @102A y @1020C deberían contener el valor 5Dec = 0005Hex. ✓

CONCLUSIONES

De lo que llevo de curso este es el trabajo más difícil al que me he enfrentado, le he dedicado muchas horas a entender como funciona cada instrucción en el lenguaje ensamblador.

Pero en el fondo, el hecho de tener que programar un emulador similar al 68k, me ha hecho entender aún más como funcionan estas máquinas. He resuelto muchas dudas durante el camino; por ejemplo, cómo funciona la pila para utilizarla en subrutinas de librería o el tratamiento de bits específicos, además de ir con cuidado al ejecutar una instrucción ya que se pueden actualizar los flags y surgir errores casi imposibles de detectar.

En conclusión, creo que el programa funciona a la perfección y estoy muy orgulloso del esfuerzo y trabajo hecho durante la práctica que seguro me servirá para mejorar en la asignatura.

CÓDIGO FUENTE

```

*-----
* Title       : PRAFIN22
* Written by  : Joan Martorell Coll
* Date       : 30/05/2022
* Description: Emulador de la JARVIS
*-----

    ORG $1000
EMEM:  DC.W $2800,$2A03,$50E0,$0B60,$5114,$0C70,$1430,$0E40,$7140,
$3001, $32FF,$90D0
        DC.W $8020,$C000,$0002,$0003,$0001,$0003,$0002,$0004,$0000,
$0000,$0000
EIR:    DC.W 0 ;eregistro de instruccion
EPC:    DC.W 0 ;econtador de programa
EB0:    DC.W 0 ;eregistro B0
EB1:    DC.W 0 ;eregistro B1
ER2:    DC.W 0 ;eregistro R2
ER3:    DC.W 0 ;eregistro R3
ER4:    DC.W 0 ;eregistro R4
ER5:    DC.W 0 ;eregistro R5
ET6:    DC.W 0 ;eregistro T6
ET7:    DC.W 0 ;eregistro T7
ESR:    DC.W 0 ;eregistro de estado (00000000 00000ZCN)

START:
    CLR.W EPC

FETCH:
    ;--- IFETCH: INICIO FETCH
        ;*** En esta seccion debeis introducir el codigo necesario
para cargar
        ;*** en el EIR la siguiente instruccion a ejecutar, indicada
por el EPC,
        ;*** y dejar listo el EPC para que apunte a la siguiente
instruccion

        ; ESCRIBID VUESTRO CODIGO AQUI
        LEA.L EMEM, A0      ;leemos instrucción
        ADD EPC, A0         ;añadimos 2 ya que va
        ADD EPC, A0         ;de 2 en 2 bytes

        MOVE.W (A0),EIR     ;añadimos la instrucción
        ADDQ.W #1, EPC      ;añadimos 1 al EPC

    ;--- FFETCH: FIN FETCH

    ;--- IBRDECOD: INICIO SALTO A DECOD
        ;*** En esta seccion debeis preparar la pila para llamar a la
subrutina
        ;*** DECOD, llamar a la subrutina, y vaciar la pila
correctamente,
        ;*** almacenando el resultado de la decodificacion en D1

        ; ESCRIBID VUESTRO CODIGO AQUI

        ;preparamos la pila
        MOVE.W #0, -(SP)
        MOVE.W EIR, -(SP)

```

```

        JSR DECOD          ;decodificamos

        ;sacamos lo que hay en la pila
        ADDQ.W #2, SP
        MOVE.W (SP)+, D1

;--- FBRDECOD: FIN SALTO A DECOD

;--- IBREXEC: INICIO SALTO A FASE DE EJECUCION
        ;*** Esta seccion se usa para saltar a la fase de ejecucion
        ;*** NO HACE FALTA MODIFICARLA
        MULU #6,D1
        MOVEA.L D1,A1
        JMP JMPLIST(A1)
JMPLIST:
        JMP ETRA
        JMP EADD
        JMP ESUB
        JMP ENAN
        JMP ESTC
        JMP EINC
        JMP ELOA
        JMP ELOAX
        JMP ESTO
        JMP ESTOX
        JMP EBRI
        JMP EBRZ
        JMP EBRN
        JMP ESTP
;--- FBREXEC: FIN SALTO A FASE DE EJECUCION

;--- IEXEC: INICIO EJECUCION
        ;*** En esta seccion debeis implementar la ejecucion de cada
einstr.

        ; ESCRIBID EN CADA ETIQUETA LA FASE DE EJECUCION DE CADA
INSTRUCCION

ETRA:      ;Xb <- [Xa]
        JSR DET_A          ;detectamos aaa
        MOVE.W (A2),D3      ;guardamos aaa
        JSR DET_B          ;detectamos bbb
        MOVE.W D3, (A2)     ;Xb <- [Xa]

        ;actualizamos flag z
        JSR FLAG_Z

        BRA FETCH

EADD:      ;Xb <- [Xb] + [Xa]

        JSR DET_A          ;detectamos aaa
        MOVE.W (A2),D3      ;guardamos aaa
        JSR DET_B          ;detectamos bbb
        MOVE.W (A2),D4      ;guardamos bbb
        ADD.W D3,D4         ;sumamos aaa y bbb
        MOVE.W D4, (A2)     ;Xb <- [Xb] + [Xa]

```

```

;actualizamos flags
JSR FLAG_Z
JSR FLAG_C
JSR FLAG_N

BRA FETCH

ESUB:      ;Xb <- [Xb] - [Xa] <==> A - B = A + (B' + 1)

JSR DET_A      ;detectamos aaa
MOVE.W (A2), D3 ;guardamos aaa
NOT D3          ;negamos aaa
ADDQ.W #1, D3   ;sumamos 1 a aaa

JSR DET_B      ;detectamos bbb
MOVE.W (A2), D4 ;guardamos bbb
ADD.W D3, D4    ;sumamos aaa'+1 y bbb
MOVE.W D4, (A2) ;Xb <- [Xb] - [Xa]

;actualizamos flags
JSR FLAG_Z
JSR FLAG_C
JSR FLAG_N

BRA FETCH

ENAN:      ;Xb <- [Xb] nand [Xa]

JSR DET_A      ;detectamos aaa
MOVE.W (A2), D3 ;guardamos aaa
JSR DET_B      ;detectamos bbb
MOVE.W (A2), D4 ;guardamos aaa

AND.W D3, D4    ;hacemos la operación NAND
NOT.W D4        ;negamos la solución

MOVE.W D4, (A2) ;Xb <- [Xb] nand [Xa]

;actualizamos flags
JSR FLAG_Z
JSR FLAG_N

BRA FETCH

ESTC:      ;Xb <- k (Ext. signo)

JSR DET_K      ;detectamos kkkkkkkk, ahora está en D2
MOVE.W D2, D3   ;guardamos kkkkkkkk
JSR DET_B      ;detectamos bbb
MOVE.W D3, (A2) ;Xb <- k

;actualizamos flags
JSR FLAG_Z
JSR FLAG_N

BRA FETCH

EINC:      ;Xb <- [Xb] + k (Ext. Signo)

```

```

JSR DET_K           ;detectamos kkkkkkkk, ahora está en D2
MOVE.W D2, D3       ;guardamos kkkkkkkk
JSR DET_B           ;detectamos bbb
ADD.W D3, (A2)       ;Xb <- [Xb] + k

;actualizamos flags
JSR FLAG_Z
JSR FLAG_C
JSR FLAG_N

BRA FETCH

ELOA:               ;T6 <- [M]

JSR DET_M           ;detectamos mmmmmmmm, ahora está en D2
MOVE.W D2, A3       ;lo guardamos en el EPC
ADD.W D2, A3        ;recordemos que la dirección va de dos en dos

MOVE.W EMEM(A3),ET6 ;T6 <- [M]

;actualizamos flags
JSR FLAG_Z
JSR FLAG_N

BRA FETCH

ELOAX:              ;Tj <- [M + [Bi]]

JSR DET_M           ;detectamos mmmmmmmm, ahora está en D2
MOVE.W D2, D3       ;guardamos mmmmmmmm en D3

JSR DET_Bi          ;detectamos Bi, está en A2
MOVE.W (A2),A3       ;guardamos Bi en D4
ADD.W (A2),A3        ;recordemos que la dirección va de dos en dos

ADD.W D3, A3         ;sumamos mmmmmmmm + [Bi]
ADD.W D3, A3         ;recordemos que la dirección va de dos en dos
MOVE.W EMEM(A3),D3   ;guardamos la variable introducida

JSR DET_Tj          ;detectamos Tj
MOVE.W D3, (A2)      ;Tj <- [M + [Bi]]

;actualizamos flags
JSR FLAG_Z
JSR FLAG_N

BRA FETCH

ESTO:               ;M <- [T6]

MOVE.W ET6, A2       ;guardamos dirección de T6
MOVE.W (A2),D3       ;guardamos lo que hay en la dirección

JSR DET_M           ;detectamos mmmmmmmm, ahora está en D2
MOVE.W D2, A3       ;lo guardamos en el EPC
ADD.W D2, A3        ;recordemos que la dirección va de dos en dos

MOVE.W D3, (A3)      ;M <- [T6]

```

```

    BRA FETCH

ESTOX:      ;M + [Bi] <- [Tj]

    JSR DET_Tj      ;detectamos Tj
    MOVE.W (A2),D3   ;guardamos Tj

    JSR DET_Bi      ;detectamos Bi
    MOVE.W (A2),A3   ;guardamos Bi en D4
    ADD.W (A2),A3    ;recordemos que la dirección va de dos en dos

    JSR DET_M       ;detectamos mmmmmmmmm, ahora está en D2
    ADD.W D2, A3     ;sumamos mmmmmmmmm + [Bi]
    ADD.W D2, A3     ;recordemos que la dirección va de dos en dos

    MOVE.W D3, EMEM(A3) ;M + [Bi] <- [Tj]

    BRA FETCH

EBRI:      ;PC <- M

    JSR DET_M       ;detectamos mmmmmmmmm, ahora está en D2
    MOVE.W D2, EPC   ;PC <- M

    BRA FETCH

EBRZ:      ;Si Z = 1, PC <- M

    JSR DET_M       ;detectamos mmmmmmmmm, ahora está en D2

    BTST #2, ESR     ;comprovamos el flag EZ
    BEQ ACT1         ;si es 0, saltamos

    MOVE.W D2, EPC   ;PC <- M

    ACT1:
        BRA FETCH

EBRN:      ;Si N = 1, PC <- M

    JSR DET_M       ;detectamos mmmmmmmmm, ahora está en D2

    BTST #0, ESR     ;comprovamos el flag EZ
    BEQ ACT1         ;si es 0, saltamos

    MOVE.W D2, EPC   ;PC <- M

    ACT2:
        BRA FETCH

ESTP:
    SIMHALT

    ;--- FEEXEC: FIN EJECUCION

    ;--- ISUBR: INICIO SUBROUTINAS
    ;*** Aquí debeis incluir las subrutinas que necesite vuestra
solucion
    ;*** SALVO DECOD, que va en la siguiente seccion

```

```

; ESCRIBID VUESTRO CODIGO AQUI

DET_A:
    MOVE.W EIR, D2      ;guarda EIR en D2
    AND.W #$0070, D2    ;detecta aaa
    LSR #4,D2           ;lo mueve a la derecha
    BRA DET_REG

DET_B:
    MOVE.W EIR, D2      ;guarda EIR en D2
    AND.W #$0700, D2    ;detecta bbb
    LSR #8, D2          ;lo mueve a la derecha
    BRA DET_REG

DET_K:
    MOVE.W EIR, D2      ;guarda EIR en D2
    AND.W #$00FF, D2    ;detecta kkkkkkkk
    EXT.W D2            ;extension de signo
    RTS

DET_M:
    MOVE.W EIR, D2      ;guarda EIR en D2
    AND.W #$0FF0, D2    ;detecta mmmmmmmm
    LSR #4,D2           ;lo mueve a la derecha
    RTS

DET_REG:                ;retorna el eregistro indicado por D2
    CMP.W #$0, D2
    BEQ ES_B0
    CMP.W #$1, D2
    BEQ ES_B1
    CMP.W #$2, D2
    BEQ ES_R2
    CMP.W #$3, D2
    BEQ ES_R3
    CMP.W #$4, D2
    BEQ ES_R4
    CMP.W #$5, D2
    BEQ ES_R5
    CMP.W #$6, D2
    BEQ ES_T6
    CMP.W #$7, D2
    BEQ ES_T7

DET_Bi:
    MOVE.W EIR, D2
    AND.W #$0008, D2    ;detectamos i
    LSR #3,D2           ;movemos 4 posiciones a la derecha

    CMP.W #1, D2        ;si D2 = 1, saltamos a B1
    BEQ ES_B1           ;si D2 = 0, saltamos a B0

    BRA ES_B0

DET_Tj:
    MOVE.W EIR, D2
    AND.W #$0004, D2    ;detectamos i
    LSR #2,D2           ;movemos 3posiciones a la derecha

    CMP.W #1, D2        ;si D2 = 1, saltamos a T7
    BEQ ES_T7           ;si D2 = 0, saltamos a T6

    BRA ES_T6

```

```

;funciones que devuelven la dirección de cada eregistro
ES_B0:
    LEA.L EB0, A2
    RTS
ES_B1:
    LEA.L EB1, A2
    RTS
ES_R2:
    LEA.L ER2, A2
    RTS
ES_R3:
    LEA.L ER3, A2
    RTS
ES_R4:
    LEA.L ER4, A2
    RTS
ES_R5:
    LEA.L ER5, A2
    RTS
ES_T6:
    LEA.L ET6, A2
    RTS
ES_T7:
    LEA.L ET7, A2
    RTS

FLAG_Z:
    ;MIRAMOS EL FLAG Z
    BEQ ES_Z1          ;SALTA A Z0 SI Z = 1

    BCLR #2, ESR        ;PONE EL BIT A 0
    RTS

    ES_Z1:
        BSET #2, ESR    ;PONE EL BIT A 1
        RTS

FLAG_C:
    ;MIRAMOS EL FLAG C
    BCS ES_C1          ;SALTA A C0 SI C = 1

    BCLR #1, ESR        ;PONE EL BIT A 0
    RTS

    ES_C1:
        BSET #1, ESR    ;PONE EL BIT A 1
        RTS

FLAG_N:
    ;MIRAMOS EL FLAG N
    BMI ES_N1          ;SALTA A N0 SI N = 1

    BCLR #0, ESR        ;PONE EL BIT A 0
    RTS

    ES_N1:
        BSET #0, ESR    ;PONE EL BIT A 1
        RTS

```

```

;--- FSUBR: FIN SUBROUTINAS

;--- IDECOD: INICIO DECOD
;*** Tras la etiqueta DECOD, debeis implementar la subrutina
de
;*** decodificacion, que debera ser de libreria, siguiendo la
interfaz
;*** especificada en el enunciado
DECOD:
; ESCRIBID VUESTRO CODIGO AQUI
MOVE.W D0, -(SP)
MOVE.W 6(SP), D0

BTST.L #15, D0
BEQ BITS_0      ;bits 0xxx

;bits 1xxx
BTST.L #14, D0
BEQ BITS_1_0    ;bits 10xxx

MOVE.W #13, 8(SP) ;bits 11xxx (STP)
BRA ACABADO
BITS_1_0:
BTST.L #13, D0
BEQ BITS_1_0_0  ;bits 100xx

MOVE.W #12, 8(SP) ;bits 101xx (BRN)
BRA ACABADO
BITS_1_0_0:
BTST.L #12, D0
BEQ BITS_1_0_0_0 ;bits 1000x

MOVE.W #11, 8(SP) ;bits 1001x (BRZ)
BRA ACABADO
BITS_1_0_0_0:
MOVE.W #10, 8(SP) ; (BRI)
BRA ACABADO

BITS_0:
BTST.L #14, D0
BEQ BITS_0_0    ;bits 00xxx

;bits 01xxx:
BTST.L #13, D0
BEQ BITS_0_1_0  ;bits 010xx

;bits 011xx:
BTST.L #12, D0
BEQ BITS_0_1_1_0 ;bits 0110x

MOVE.W #9, 8(SP) ;bits 0111x (STOX)
BRA ACABADO
BITS_0_1_1_0:
MOVE.W #8, 8(SP) ; (STO)
BRA ACABADO
BITS_0_1_0:
BTST.L #12, D0

```



```

        BEQ BITS_0_1_0_0      ;bits 0100x

        MOVE.W #7, 8(SP)      ;bits 0101x (LOAX)
        BRA ACABADO
BITS_0_1_0_0:
        MOVE.W #6, 8(SP)      ;(LOA)
        BRA ACABADO
BITS_0_0:
        BTST.L #13, D0
        BEQ BITS_0_0_0      ;bits 000xx
;bits 001xx:
        BTST.L #12, D0
        BEQ BITS_0_0_1_0      ;bits 0010x

        MOVE.W #5, 8(SP)      ;bits 0011x (INC)
        BRA ACABADO
BITS_0_0_1_0:
        BTST.L #11, D0
        BEQ BITS_0_0_1_0_0      ;bits 00100

        MOVE.W #4, 8(SP)      ;bits 00101 (STC)
        BRA ACABADO
BITS_0_0_1_0_0:
        MOVE.W #3, 8(SP)      ;(NAN)
        BRA ACABADO

BITS_0_0_0:
        BTST.L #12, D0
        BEQ BITS_0_0_0_0      ;bits 0000x
;bits 0001x:
        BTST.L #11, D0
        BEQ BITS_0_0_0_1_0      ;bits 00010

        MOVE.W #2, 8(SP)      ;bits 00011 (SUB)
        BRA ACABADO
BITS_0_0_0_1_0:
        MOVE.W #1, 8(SP)      ;(ADD)
        BRA ACABADO
BITS_0_0_0_0_0:
        MOVE.W #0, 8(SP)      ;bits 00001 (TRA)

ACABADO:
        MOVE.W (SP)+, D0
        RTS

;--- FDECOD: FIN DECOD
END      START

```