Memoria práctica Estructura de Computadores I

Asignatura: Estructura de Computadores I

Curso: 1ero Ingeniería informática 2021/2022

Profesores: Alberto Ortiz Rodríguez y Francisco Bonnin Pascual

Proyecto realizado por:

- Daniel Salanova Dmitriyev DNI:49610682G Correo: dasadmx@gmail.com
- Hugo Valls Sabater DNI:43474568Z Correo: vs824@id.uib.cat

Fecha: 30 de mayo de 2022

Índice general

Capítulo 1

Introducción

1.1. Objetivos

El objetivo de esta práctica consiste en el desarrollo de un emulador. Un emulador escrito en lenguaje ensamblador del 68K que permita la ejecución de programas escritos para una máquina elemental dada.

Este emulador se titula Just A Rather Very Intelligent System (JARVIS), este contiene ocho registros: B0, B1, R2, R3, R4 y R5 se utilizan en operaciones de tipo ALU tanto de operando fuente como destino. Por otro lado también cuenta con los registros T6 y T7 que se utilizan como interfaz de la memoria, además de utilizarse también en operaciones de tipo ALU.

1.2 Estructura del informe

La memoria se estructura en diversas partes, incluyendo este primer capítulo introductorio con una breve descripción de la práctica. Una explicación general al trabajo realizado para cada una de las fases del emulador. Una descripción de la rutina de decodificación. Varias tablas, para las subrutinas utilizadas, y otra para los registros del 68k empleados. A continuación se encuentra el conjunto de pruebas para comprobar el correcto funcionamiento. Finalmente encontramos el apartado de las conclusiones donde se evaluarán los resultados obtenidos y el código fuente

Capítulo 2

Diseño Descendente

2.1 Estructura del programa

El programa se divide en tres partes principales, la primera, la fase de FETCH, fase donde se recorre el vector EMEM que contiene las instrucciones codificadas, se obtiene una de ellas (de principio a final) y se incrementa en uno el registro EPC para así apuntar a la siguiente instrucción al volver a la fase de FETCH.

La segunda es la fase de decodificación, en esta nos traemos dicha instrucción obtenida de EMEM y la decodificamos mediante una serie de instrucciones repetitivas que se mostrarán posteriormente. Una vez se conoce la instrucción se trata, y se obtiene el identificador que le corresponde para así posteriormente poder ejecutarla.

Por último en la fase de ejecución mediante el identificador mencionado anteriormente se salta la instrucción correcta a ejecutar y se ejecutan seguidamente se actualizan los flags si fuese necesario, una vez hecho esto se vuelve a saltar al FETCH para así continuar ejecutando el resto de instrucciones, hasta llegar a la instrucción STP, donde acabaría la ejecución del programa.

Ahora se procederá a explicar cada una de las fases y cómo hemos solucionado cada uno de los problemas encontrados a lo largo de la práctica.

2.2 Fase de Fetch

Esta primera fase consta de traer cada una de las instrucciones codificadas que se encuentran almacenadas dentro de la EMEM de nuestra máquina.

La dificultad de esta fase radica en que las posiciones de la memoria de la máquina JARVIS empiezan en el 0, y se deben recoger correctamente cada una de ellas teniendo en cuenta el valor del PC. Viendo como están almacenadas dentro del 68K podemos encontrar una equivalencia para poder traer de la memoria cada una de las instrucciones.

De esta manera:

EPC	@Mem en 68K
0	@EMEM + 0
1	@EMEM + 2
2	@EMEM + 4
...	

Viendo esta relación podemos observar que el valor de la suma de la posición que debemos hallar es: **POS = EMEM + PC*2**

Por tanto eso lo traducimos en 68K como el siguiente conjunto de operaciones:

```
;--- 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

MOVE.W  EPC,D7
MULS.W  #2,D7
MOVE.W  D7,A0
MOVE.W  EMEM(A0),EIR
ADD.W   #1,EPC

;--- FFETCH: FIN FETCH
```

Con este código obtendremos la instrucción codificada que se requiere y la almacenaremos en el registro de programa (EIR), además incrementaremos en 1 el EPC.

2.3 Fase de Decodificación

Una vez obtenida la instrucción codificada y almacenada en EIR debemos descubrir que tipo de instrucción es de las 14 que hay, es por eso que en esta fase trataremos de encontrar el ID (de 0-13) de la instrucción que se encuentra en el EIR.

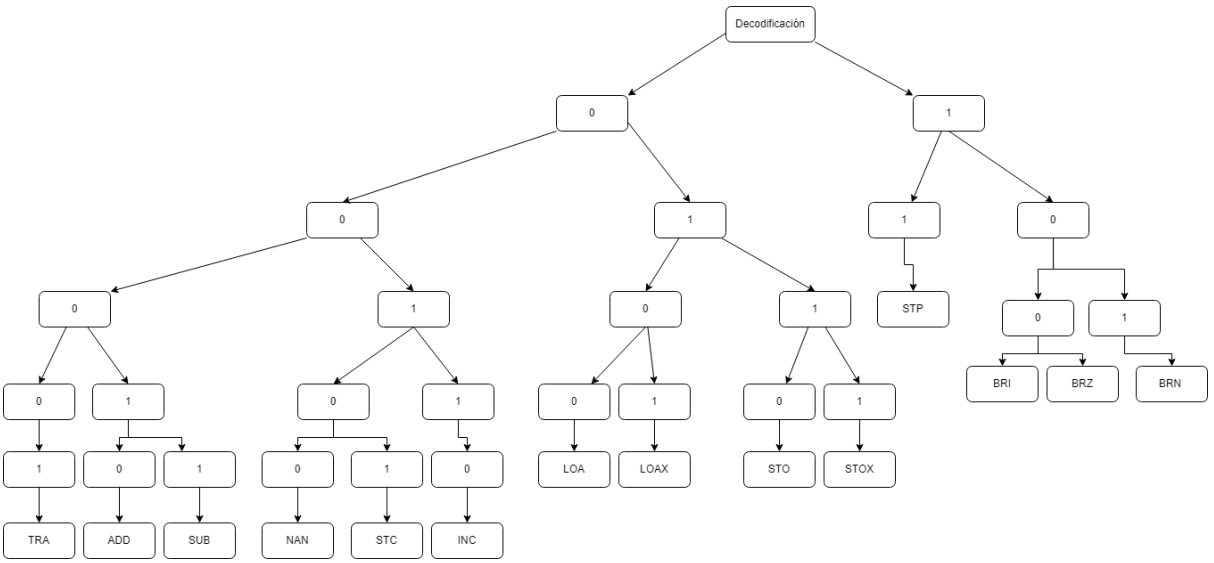
A la hora de implementar esta fase, hemos decidido implementarla como una subrutina de librería. Es por ello que antes de saltar a la subrutina, guardamos un espacio para almacenar el ID de la instrucción y otro para almacenar el EIR:

```
;Preparamos la pila
MOVE.W  #0,-(A7)
MOVE.W EIR,-(A7)

JSR DECOD
```

Una vez hemos llegado a la subrutina buscaremos cúal es el identificador de la instrucción del EIR mediante el análisis secuenciado de los bits de la instrucción codificada. La manera en que lo hemos hecho ha sido usando la instrucción BTST que nos proporciona 68K.

Cómo es demasiado extenso para ser explicado en este documento, hemos decidido adjuntar una imagen donde se muestra el árbol de secuenciación de bits que hemos usado para implementar esta subrutina.



Al terminar la subrutina de DECOD recuperaremos el EIR que almacenamos antes de entrar en la subrutina, y el ID de nuestra instrucción codificada.

```
MOVE.W   (A7)+,EIR
MOVE.W   (A7)+,D1
```

De esta manera también dejaremos la pila como estaba anteriormente y con el puntero de pila en el sitio correspondiente,

Esta fase terminará moviendo el contenido del ID dentro del registro D1 del 68K para que funcione el código aportado por los profesores para la búsqueda de la etiqueta a la que saltar según el ID y así se ejecute correctamente la instrucción.

2.4 Fase de Ejecución

Esta fase no tiene mucho misterio, para cada una de las instrucciones que se nos plantean para JARVIS, lo que hemos hecho ha sido buscar maneras para implementarlas con las instrucciones del 68K.

En esta fase hemos usado 2 subrutinas, por una parte ‘CREG’ y por otra ‘FLAGS’.

Lo que hacemos con CREG es buscar en qué posición se encuentra y qué contenido tiene uno de los operandos de la instrucción que se encuentra almacenado en D3. Para descubrir a qué registro hace referencia el código del operando, lo que hacemos es una búsqueda repetitiva de los bits hasta saber que registro es. De esta manera una vez descubierto, se incorpora dentro de D3 y A3, por una parte el contenido del registro y por otra la dirección.

Por último la subrutina de FLAGS lo que hacemos es coger el contenido de los flags e incorporarlo dentro del registro ESR según convenga.

Capítulo 3

Tablas

3.1 Tabla de subrutinas

DECOD	DECOD es una subrutina de librería, en esta decodificamos y obtenemos el identificador de cada instrucción para ejecutarla posteriormente. Al ser de librería hay que pasarle los parámetros necesarios, así como guardar el contenido de los registros de datos antes de ser utilizados. Por ello antes de saltar reservamos un espacio (al principio de la pila), word donde guardaremos el identificador, un word por

	encima movemos el contenido de EIR donde pondremos la cadena de bits a decodificar. Una vez saltamos a la subrutina guardamos el contenido de los registros D0 y D6, ya que serán utilizados (su contenido es explicado en la siguiente tabla). Posteriormente seguimos una decodificación exhaustiva mirando bit a bit hasta encontrar de qué instrucción se trata mediante BTST y diferentes saltos según el valor de estos. Una vez acabada la decodificación movemos el id a la primera posición de la pila. Devolvemos los valores iniciales de D0 y D6, devolvemos el valor de EIR y movemos el id a D1.
CREG	Esta subrutina de usuario se utiliza para saber el registro del operando fuente/destino, así como la dirección del operando destino para saber donde guardar el resultado de la operación a ejecutar. En esta se encuentra un proceso de decodificación similar al de DECOD donde se van mirando bit a bit de qué registro se trata, una vez se decodifica movemos el contenido del registro correspondiente a D3 y su dirección a A3. A esta subrutina hay que pasarle como parámetro el código del operando, estando este almacenado en D3.
FLAGS	En esta subrutina de usuario nos encargamos de actualizar los flags de nuestro emulador, para ello antes de saltar movemos SR (status register) a D4, dentro movemos D4 a D5 para no destruir el contenido, y realizamos sucesivas AND y OR con la dirección de memoria ESR donde están los flags de nuestro emulador para así obtener los valores de los flags que nos interesan, siendo estos el flag N, C y Z.

3.2 Tabla de registros del 68K

3.2.1 Tabla de registros de datos

D0	Usado en la subrutina DECOD para hacer el tratamiento del EIR para descubrir qué operación realiza.
D1	Usado durante la fase de decodificación, en este registro se incorporará el identificador de la instrucción una vez se vuelva de la subrutina DECOD.
D2	Registro usado durante la fase de ejecución en cada una de la einstrucciones. Sirve como registro auxiliar para almacenar el contenido del operando src o del valor de los bits de memoria.
D3	Registro usado en cada una de las einstrucciones para almacenar el contenido del operando dst/src. Usado también en la subrutina CREG para devolver el contenido del operando.
D4	En D4 movemos el SR (status register) los flags del 68k y los tratamos de manera en que podamos actualizar los flags de nuestra máquina en el word ESR.
D5	Este registro se usa para las instrucciones de JMP en la fase de ejecución, movemos el contenido de EIR a D5 y extraemos la dirección de memoria a la cual saltaremos o no dependiendo de la condición del JMP.
D6	Usado durante la fase de decodificación, en este registro insertamos el valor del identificador de la einstrucción que hemos estado decodificando.
D7	Este registro se utiliza Únicamente en la fase de FETCH, para guardar la dirección de memoria que nos traemos del EPC y luego lo multiplicamos por dos y movemos su contenido a un registro de direcciones para así recorrer EPROG de manera correcta.

3.2.1 Tabla de registros de direcciones

A0	Este registro de dirección se utiliza para recorrer el vector EMEM usando el modo indexado básico EMEM(A0), en la fase de FETCH.
A1	Este registro se usa para guardar la dirección del identificador de la instrucción a ejecutar, una vez guardada saltamos a la instrucción correspondiente mediante el modo indirecto por registro (A1)
A3	A3 es muy importante debido a su uso en las subrutinas CREG donde una vez obtenemos de qué registro se trata el operando. Cargamos con la instrucción LEA la dirección de memoria del operando, donde hemos de mover el resultado de la operación realizada en cada instrucción
A2	Estos registros de direcciones son registros auxiliares, utilizados en instrucciones como LOA o STO cuando se ha necesitado guardar ciertas direcciones de memoria.

Capítulo 4

Juego de pruebas

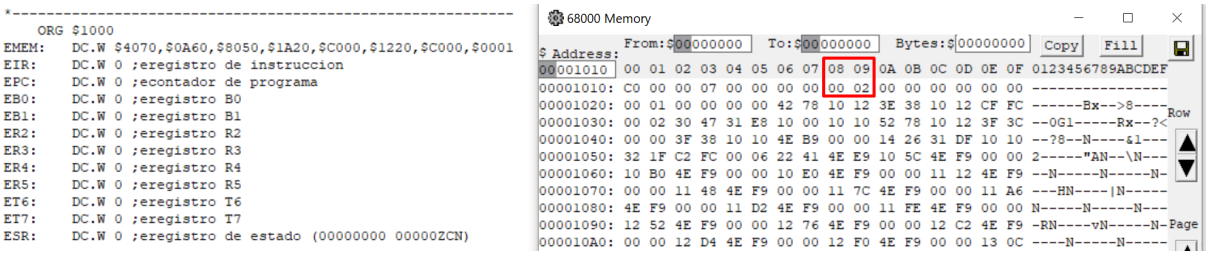
A continuación se mostrarán las pruebas que hemos realizado sobre el proyecto desarrollado con el fin de demostrar el correcto funcionamiento de este.

4.1 Prueba 1

En esta primera prueba, hemos decidido ejecutar el programa mínimo para que la práctica pueda ser evaluada, este programa es aportado por el profesorado en el documento de la práctica. El programa es el siguiente:

```
    ORG $1000
EMEM:   DC.W $4070,$0A60,$8050,$1A20,$C000,$1220,$C000,$0001
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)
```

Este programa debe dar como resultado en la posición @1018Hex el valor 0002Hex. El resultado que obtenemos de la ejecución de este conjunto de instrucciones en nuestra implementación es la siguiente:



4.1 Prueba 2

En esta prueba hemos decidido ejecutar el programa de la propia práctica. La cabecera es la siguiente

```
    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)
```

El resultado que debe dar como resultado en las posiciones @1028, @102A y @102C el valor 0005Hex, estas posiciones hacen referencia al vector C. Por tanto una vez ejecutado el programa el resultado que da en nuestra implementación es la siguiente:

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

```
68000 Memory
From:$00000000  To:$00000000  Bytes:$00000000  Copy  Fill
$ Address:
00001020  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
00001020: 00 01 00 03 00 02 00 04 00 05 00 05 00 05 C0 00 ----------------
00001030: 00 0E 00 03 00 00 00 00 00 01 00 05 00 00 00 05 ----------------
00001040: 00 04 00 04 42 78 10 30 3E 38 10 30 CF FC 00 02 ----Bx-0>8-0----
00001050: 30 47 31 E8 10 00 10 2E 52 78 10 30 3F 3C 00 00 0G1-----.Rx-0?<--
00001060: 3F 38 10 2E 4E B9 00 00 14 44 31 DF 10 2E 32 1F ?8-.N----D1--.2-
00001070: C2 FC 00 06 22 41 4E E9 10 7A 4E F9 00 00 10 CE ----"AN--zN-----
00001080: 4E F9 00 00 10 FE 4E F9 00 00 11 30 4E F9 00 00 N-----N----0N---
00001090: 11 66 4E F9 00 00 11 9A 4E F9 00 00 11 C4 4E F9 -fN-----N-----N-
000010A0: 00 00 11 F0 4E F9 00 00 12 1C 4E F9 00 00 12 70 ----N-----N----p
000010B0: 4E F9 00 00 12 94 4E F9 00 00 12 E0 4E F9 00 00 N-----N-----N---
```

Capítulo 5

Conclusiones

En esta práctica hemos aprendido y hemos podido aplicar toda la teoría enseñada durante este curso para el diseño e implementación de un programa en ensamblador de cierta complejidad. Para ello ha sido necesario realizar de manera correcta las tres fases que todo máquina según la arquitectura de Von Neuman implementa, la fase de fetch donde nos traemos la instrucción a ejecutar e incrementamos en uno el pc, la fase de decodificación donde decodificamos la instrucción que nos traemos para así saber cual hay que ejecutar y finalmente en la fase de ejecución donde ejecutamos la instrucción y actualizamos los flags aplicando así los conceptos de la arquitectura de Von Neuman y la programación en ensamblador vistos en clase.

Durante la realización de esta práctica, nos han ido surgiendo diferentes dudas y problemas que hemos podido solucionar. Algunos de estos han sido, en la fase de ejecución mayoritariamente, las instrucciones de load y store al saber extraer y tratar la dirección de memoria con la cual guardar algún dato en dicha dirección de memoria o cargar el contenido de dicha dirección en otra. Otro gran problema que nos surgió fue el saber cómo decodificar y comprobar de qué registro se trataba cada operando una vez traída la instrucción, así como la actualización de los flags.

Capítulo 6

Código fuente

```
*-----------------------------------------------------------
* Title      : PRAFIN22
```

```
* Written by : Daniel Salanova Dmitriyev y Hugo Valls Sabater
* 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

      MOVE.W     EPC,D7
      MULS.W     #2,D7
      MOVE.W     D7,A0
      MOVE.W     EMEM(A0),EIR
      ADD.W #1,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,-(A7)
     MOVE.W EIR,-(A7)

     JSR DECOD

    ;Recogemos los datos incorporados que se encontraban final de
la pila
     MOVE.W     (A7)+,EIR
     MOVE.W     (A7)+,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:
     MOVE.W     EIR,D3
     ;Obtenemos A
    AND.W  #%0000000001110000,D3
    LSR.L  #4,D3

    JSR CREG ;Hayamos posicion de A y su contenido
    MOVE.W D3,D2 ;Contenido de A movido a D2

    ;Guardamos los flags de la operacion y actualizamos los de
JARVIS
    MOVE.w SR, D4
    JSR FLAGS

    MOVE.W      EIR,D3
    ;Obtenemos B
    AND.W  #%0000011100000000,D3
    LSR.L  #8,D3
    JSR CREG

    ;Copiamos el contenido de A a la poscion de B
    MOVE.W D2, (A3)

     JMP FETCH
EADD:
     MOVE.W     EIR,D3
     ;Obtenemos A
    AND.W  #%0000000001110000,D3
    LSR.L  #4,D3

    JSR   CREG ;Hayamos posicion de A y su contenido

    MOVE.W      D3,D2 ;Contenido movido de A a D2

    MOVE.W      EIR,D3
    ;Obtenemos B
    AND.W  #%0000011100000000,D3
    LSR.L  #8,D3

    JSR CREG ;Hayamos posicion de B y su contenido

    ;Sumamos B + A
    ADD.W D2,D3
```

```
    ;Guardamos los flags de la operacion y actualizamos los de
JARVIS
    MOVE.W SR,D4
    JSR FLAGS

    ;Copiamos el resultado a B
    MOVE.W D3, (A3)
     JMP FETCH
ESUB:
     MOVE.W EIR,D3
    AND.W #%0000000001110000,D3  ;Obtenemos A
    LSR.L #4,D3

    JSR CREG ;Hayamos posicion de A y su contenido

    MOVE.W D3,D2 ;Contenido movido de A a D2

    MOVE.W EIR,D3
    AND.W #%0000011100000000,D3 ;Obtenemos B
    LSR.L #8,D3
    JSR CREG ;Hayamos posicion de B y su contenido

    ;Se ejecuta la operacion A + (NO_B1 + 1)
    NOT D3
    ADD.W #1,D3
    ADD.W D2,D3

    ;Guardamos los flags de la operacion y actualizamos los de
JARVIS
    MOVE.W SR,D4
    JSR FLAGS

    ;Copiamos el resultado a B
    MOVE.W D3, (A3)
     JMP FETCH
ENAN:
     MOVE.W EIR,D3
     ;Obtenemos A
    AND.W #%0000000001110000,D3
    LSR.L #4,D3

    JSR CREG ;Hayamos posicion de A y su contenido
    MOVE.W D3,D2 ;Contenido movido de A a D2

    MOVE.W EIR,D3
    AND.W #%0000011100000000,D3 ;Obtenemos B
    LSR.L #8,D3
    JSR CREG ;Hayamos posicion de B y su contenido
```

```
    ;Hacemos la AND de D3 y D2 para luego negarla,
    ;que es lo equivalente a NAND
    AND.W D2, D3
    NOT.W D3

    ;Guardamos los flags de la operacion y actualizamos los de
JARVIS
    MOVE.W SR,D4
    JSR FLAGS

    ;Copiamos el resultado a B
    MOVE.W D3, (A3)
     JMP FETCH
ESTC:
     MOVE.W EIR,D3
     AND.W #%0000000011111111,D3
     ;Extendemos K para hacer las operaciones correctamente
    EXT.W D3

     MOVE.W D3, D2

     MOVE.W EIR, D3
     AND.W #%0000011100000000,D3
     LSR.L #8,D3
     JSR CREG ;Hayamos posicion de B y su contenido

     MOVE.W D2, (A3) ; Almacenamos K en posicion de B

    ;Guardamos los flags de la operacion y actualizamos los de
JARVIS
    MOVE.W SR,D4
     JSR FLAGS

     JMP FETCH
EINC:
     MOVE.W EIR,D3
     AND.W #%0000000011111111,D3
     ;Extendemos K para hacer las operaciones correctamente
    EXT.W D3

    MOVE.W D3, D2

     MOVE.W EIR, D3
     AND.W #%0000011100000000,D3
     LSR.L #8,D3
     JSR CREG
```

```
      ADD.W D2, D3

      MOVE.W SR,D4
      JSR FLAGS

    MOVE.W D3, (A3)
      JMP FETCH
ELOA:
      MOVE.W EIR, D3
      AND.W #%0000111111110000,D3
      LSR.L #4, D3

    ;Almacenamos M en D2
      MOVE D3, D2

      MOVE.W #%0000000000000110, D3 ;Tenemos T6
      JSR CREG ;Tenemos poscion y contenido de T6

      ;Multiplicamos por 2 para hayar correctamente el indice
      ;de la posicion que buscamos dentro de la memoria
      MULU.W #2,D2
    MOVE.W D2, A2

      MOVE.W EMEM(A2),(A3)

      ;Actualizamos flags
    MOVE.W SR,D4
      JSR FLAGS

      JMP FETCH
ELOAX:
      MOVE.W EIR, D3
      AND.W #%0000111111110000,D3
      LSR.L #4, D3

    ;LEA.L EMEM,A2 ;Hayamos la direccion de EMEM
    MOVE.W D3,D2  ;Guardamos M

      MOVE.W EIR, D3
      AND.W #%0000000000001000,D3
      ;Hayamos i y limipiamos los bits 1 y 2 ya que deben
      ;ser 00 para ser B0 o B1
    LSR.L #3, D3
      BCLR.L #1,D3
      BCLR.L #2,D3 ;Ahora tendremos en D3 00X -> X = 1 o 0
      JSR CREG ; Hayamos contenido de Bi
```

```
      ;Ahora tenemos contenido Bi + M
      ADD.W D3, D2
     ;Multiplicamos por 2 para hayar correctamente el indice
      ;de la posicion que buscamos dentro de la memoria
      MULU.W #2,D2

     MOVE.W EIR, D3
      AND.W #%0000000000000100,D3 ;Mascara
      LSR.L #2, D3 ;Tenemos j
      BSET.L #1,D3
      BSET.L #2,D3 ;Ahora tendremos en D3 11X -> X = 1 o 0
      JSR CREG ; Hayamos contenido de Tj y direccion

      MOVE.W D2, A2
      MOVE.W EMEM(A2),(A3)

     MOVE.W SR,D4 ;Actualizamos flags
      JSR FLAGS

      JMP FETCH
ESTO:
      MOVE.W EIR, D3
      AND.W #%0000111111110000,D3
      LSR.L #4, D3

      MOVE D3, D2 ; Almacenamos M en D2

      MOVE.W #%0000000000000110, D3 ;Tenemos T6
      JSR CREG ;Tenemos poscion y contenido de T6

      ;Multiplicamos por 2 para hayar correctamente el indice
      ;de la posicion que buscamos dentro de la memoria
      MULU.W #2,D2
      MOVE.W D2, A2

      MOVE.W D3,EMEM(A2)

      JMP FETCH
ESTOX:
      MOVE.W EIR, D3
      AND.W #%0000111111110000,D3
      LSR.L #4, D3

     MOVE.W D3,D2  ;Guardamos M

      MOVE.W EIR, D3
      AND.W #%0000000000001000,D3
      LSR.L #3, D3 ;Tenemos i
```

```
      BCLR.L #1,D3
      BCLR.L #2,D3 ;Ahora tendremos en D3 00X -> X = 1 o 0
      JSR CREG ; Hayamos contenido de Bi

      ADD.W D3, D2

    ;Multiplicamos por 2 para hayar correctamente el indice
      ;de la posicion que buscamos dentro de la memoria
      MULU.W #2,D2
      MOVE.W D2, A2

    MOVE.W EIR, D3
      AND.W #%0000000000000100,D3 ;Mascara
      LSR.L #2, D3 ;Tenemos j
      BSET.L #1,D3
      BSET.L #2,D3 ;Ahora tendremos en D3 11X -> X = 1 o 0
      JSR CREG ; Hayamos contenido de Tj y direccion


      MOVE.W D3,EMEM(A2) ;Movemos [M + [Bi]] -> A3

      JMP FETCH
EBRI:
      MOVE.W EIR,D5
    AND.W #$0FF0,D5
    LSR.L  #4,D5

    ;Este saltos es incondicional por tantno ponemos
    ;la direccion de memoria al EPC
    MOVE.W D5,EPC
      JMP FETCH
EBRZ:
      MOVE.W EIR,D5
    AND.W #$0FF0,D5
    LSR.L  #4,D5


    AND.W #%0000000000000100,ESR ;Mascara del flag Z
    ;Si Z=0 volvemos al fetch, si Z=1 movemos la
    ;direccion de memorai a EPC
    BEQ FETCH
    MOVE.W D5,EPC

      JMP FETCH
EBRN:
      MOVE.W EIR,D5
    AND.W #$0FF0,D5
    LSR.L  #4,D5
```

```
    AND.W #%0000000000000001,ESR ;Mascara del flag N

    ;Si N=0 volvemos al fetch, si N=1 movemos la
    ;direccion de memorai a EPC
    BEQ FETCH
    MOVE.W D5,EPC

    JMP FETCH


ESTP:

     SIMHALT
    ;--- FEXEC: FIN EJECUCION


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



FLAGS:        ;(00000000 000XNZVC)--(00000000 00000ZCN)
   ;FLAG N
    MOVE.W D4,D5
    AND.W #%0000000000001000,D5  ;HACEMOS UNA MASCARA DEL BIT 3
    BEQ N0
    JMP N1

N0:         ;SI EL BIT 3 DE SR ES 0 CAMBIAMOS EL VALOR DE EL BIT 2
DE ESR A 0
    AND.W #%1111111111111110,ESR
    JMP FLAGC

N1:         ;SI EL BIT 3 DE SR ES 1 CAMBIAMOS EL VALOR DE EL BIT 2
DE ESR A 1
    OR.W #%0000000000000001,ESR

FLAGC:
    ;FLAG C     ;(00000000 000XNZVC)--(00000000 00000ZCN)
    MOVE.W D4,D5
    AND.W #%0000000000000001,D5 ;HACEMOS UNA MASCARA DEL BIT 0
    BEQ C0
    JMP C1
```

```
C0:         ;SI EL BIT 0 DE SR ES 0 CAMBIAMOS EL VALOR DE EL BIT 1
DE ESR A 0
    AND.W #%1111111111111101,ESR
    JMP FLAGZ

C1:         ;SI EL BIT 0 DE SR ES 1 CAMBIAMOS EL VALOR DE EL BIT 1
DE ESR A 1
    OR.W #%0000000000000010,ESR


FLAGZ:  ;(00000000 000XNZVC)--(00000000 00000ZCN)
    ;FLAG Z
    MOVE.W D4,D5
    AND.W #%0000000000000100,D5 ;HACEMOS UNA MASCARA DEL BIT 2
    BEQ FN0
    JMP FN1

FN0:        ;SI EL BIT 2 DE SR ES 0 CAMBIAMOS EL VALOR DE EL BIT 0
DE ESR A 0
    AND.W #%1111111111111011,ESR
    RTS

FN1:        ;SI EL BIT 2 DE SR ES 1 CAMBIAMOS EL VALOR DE EL BIT 0
DE ESR A 1
    OR.W #%0000000000000100,ESR
    RTS


;//////// Inicio de la subrutina CREG
CREG:
    BTST.L #2,D3
    BEQ REG0
    JMP REG1
REG1:
    BTST.L #1,D3
    BEQ REG10
    JMP REG11

REG0:
    BTST.L #1,D3
    BEQ REG00
    JMP REG01
REG01:
    BTST.L #0,D3
    BEQ R2
    JMP R3

REG10:
```

```
    BTST.L #0,D3
    BEQ R4
    JMP R5
REG11:
    BTST.L #0,D3
    BEQ T6
    JMP T7

REG00:
    BTST.L #0,D3
    BEQ B0
    JMP B1


B0:
    MOVE.W EB0,D3 ;SABEMOS QUE ES B0
    LEA EB0, A3
    RTS
B1:
    MOVE.W EB1,D3 ;SABEMOS QUE ES B1
    LEA EB1, A3
    RTS
R2:
    MOVE.W ER2,D3 ;SABEMOS QUE ES R2
    LEA ER2, A3
    RTS
R3:
    MOVE.W ER3,D3 ;SABEMOS QUE ES R3
    LEA ER3, A3
    RTS
R4:
    MOVE.W ER4,D3 ;SABEMOS QUE ES R4
    LEA ER4, A3
    RTS
R5:
    MOVE.W ER5,D3 ;SABEMOS QUE ES R5
    LEA ER5, A3
    RTS
T6:
    MOVE.W ET6,D3 ;SABEMOS QUE ES T6
    LEA ET6, A3
    RTS
T7:
    MOVE.W ET7,D3 ;SABEMOS QUE ES T7
    LEA ET7, A3
    RTS

          ; ESCRIBID VUESTRO CODIGO AQUI
```

```
    ;--- FSUBR: FIN SUBRUTINAS


    ;--- 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,-(A7) ;GUARDAMOS LOS VALORES DE ESTOS REGISTROS
PORQUE LOS
                     ;USAMOS EN DECOD Y ASI NO LOS MODIFICAMOS
    MOVE.W D6,-(A7)
    MOVE.W 8(A7),D0 ;METEMOS EL EIR DE LA PILA EN D0


     ; ESCRIBID VUESTRO CODIGO AQUI
    BTST.L #15,D0
    BEQ O ;SALTAMOS SI EL BIT QUE COMPROBAMOS ES 0 , EN CASO
NEGATIVO

    JMP I ;LEEMOS LA SIGUIENTE INSTRUCCION JMP YA TENIENDO EN
CUENTA QUE ES 1
I:
    BTST.L #14,D0
    BEQ IO
    JMP II

O:  ;
    BTST.L #14,D0
    BEQ OO
    JMP OI

OO:
     BTST.L #13,D0
     BEQ OOO
     JMP OOI

OOO:
     BTST.L #12,D0
     BEQ  OOOO
     JMP OOOI

OOI:
     BTST.L #12,D0
```

```
      BEQ  OOIO
      JMP OOII

OOOO:
      BTST.L #11,D0
      BEQ  ERROR ;ERROR
      JMP OOOOI

OOOI:
      BTST.L #11,D0
      BEQ  OOOIO
      JMP OOOII

OOOOI:;TRA
      MOVE.W #0,D6
      JMP VEC
OOOIO: ;ADD
      MOVE.W #1,D6
      JMP VEC
OOOII: ; SUB
      MOVE.W #2,D6
      JMP VEC
OOII:
      BTST.L #11,D0
      BEQ  OOIIO
      JMP ERROR ; ERROR

OOIO:
      BTST.L #11,D0
      BEQ  OOIOO
      JMP OOIOI
OOIOO: ; NAN
      MOVE.W #3,D6
      JMP VEC
OOIOI: ; STC
      MOVE.W #4,D6
      JMP VEC
OOIIO: ; INC
      MOVE.W #5,D6
      JMP VEC




OI:
    BTST.L #13,D0
    BEQ OIO
```

```
    JMP OII

OIO:
     BTST.L #12,D0
    BEQ OIOO
    JMP OIOI
OII:
     BTST.L #12,D0
    BEQ OIIO
    JMP OIII
OIOO: ;LOA M
     MOVE.W #6,D6
     JMP VEC
OIOI: ;LOAX     M(Bi),Tj
     MOVE.W #7,D6
     JMP VEC
OIIO: ;STO M
     MOVE.W #8,D6
     JMP VEC
OIII: ;STOX Tj,M(Bi)
     MOVE.W #9,D6
     JMP VEC
II:   ; STP


     MOVE.W #13,D6
     JMP VEC


IO:
    BTST.L #13,D0
    BEQ IOO
    JMP IOI
IOO:
    BTST.L #12,D0
    BEQ IOOO
    JMP IOOI

IOOI: ;BRZ M
     MOVE.W #11,D6
     JMP VEC
IOOO: ;BRI M
    MOVE.W #10,D6
     JMP VEC
IOI:
    BTST.L #12 ,D0
    BEQ IOIO
    JMP ERROR ; error
```

```
IOIO: ;BRN M
    MOVE.W #12,D6
      JMP VEC




ERROR:
     SIMHALT
VEC:
    MOVE.W D6,10(A7) ;AQUI ES DONDE PONEMOS EL ID AL FINAL DE LA
PILA
    CLR D6
    JMP FINAL

FINAL:
    MOVE.W (A7)+,D6  ;RECUPERAMOS LOS VALORES DE LOS REGISTROS
                     ;PARA DEJARLOS COMO ESTABAN ANTES
    MOVE.W (A7)+,D0  ;RTS TENEMOS LA DIRECCION PC Y LUEGO CUANDO
                     ;VOLVEMOS AL PROG PRINCIPAL RECUPERAMOS EIR E
ID
    RTS


    ;--- FDECOD: FIN DECOD
    END    START
```