Cuaderno de Prácticas
Laboratorio de Fundamentos de Computadores

PARTE II:
Programación en ensamblador

Autor: Rafael Moreno Vozmediano

Facultad de Informática
Universidad Complutense de Madrid

La Máquina Rudimentaria: Arquitectura del repertorio de instrucciones

1. Registros y memoria visibles al programador

1.1. Registros de propósito general: R1-R7

Los registros R1-R7 son registros de 16 bits que se pueden utilizar como registros fuente y destino en operaciones aritmeticológicas.

Ejemplos:

Instrucción	**Operación realizada**
ADD R1,R2,R3	R3 ← R1 + R2
SUB R1,R2,R3	R3 ← R1 - R2

También se pueden utilizar como registro índice en operaciones de movimiento de datos con memoria:

Ejemplos:

Instrucción	**Operación realizada**
LOAD 4(R1),R2	R2 ← Mem(R1 + 4)
STORE R1,-10(R2)	Mem(R2 - 10) ← R1

1.2. El registro R0

El **registro R0** es un registro especial, también de 16 bits, que siempre contiene el valor 0. Este registro puede utilizarse como registro fuente (por ejemplo para inicializar otros registros a 0), pero no debe utilizarse para guardar el resultado de una operación.

Ejemplos:

Instrucción	**Operación realizada**
ADD R0,R0,R3	R3 ← 0
LOAD 16(R0),R2	R2 ← Mem(16)
STORE R1,200(R0)	Mem(200) ← R1

1.3. El registro de estado

El registro de estado contiene dos bits de estado (Z y N) que se actualizan después de cada operación:

Bit	Significado	Se actualiza si
Z	Cero	El resultado de la última operación es cero
N	Negativo	El resultado de la última operación es negativo

1.4. La memoria

La memoria contiene 256 palabras de 16 bits. Las direcciones de memoria son por tanto de 8 bits (de la 0 a la 255).

2. Modos de direccionamiento y tipos de operandos

2.1. Direccionamiento directo de registro: Ri

En este modo de direccionamiento, el operando está contenido en un registro de propósito general (Ri), que se puede utilizar como operando fuente o destino (excepto si se trata del registro R0)

Ejemplos:

Instrucción	Operación realizada
ADD R1,R2,R3	R3 ← R1 + R2
SUB R1,R2,R3	R3 ← R1 - R2

El operando almacenado en el registro es un valor de 16 bits expresado en C'2. El rango de valores que puede este operando es el siguiente:

Operando en registro (16 bits – C'2): Rango [-32768, +32767]

2.2. Direccionamiento inmediato: #valor$_5$

En este modo de direccionamiento, el operando está contenido en el propio código de instrucción máquina. Este modo de direccionamiento sólo puede utilizarse como operando fuente, nunca como operando destino.

Ejemplos:

Instrucción	**Operación realizada**
ADDI R1,#8,R3	R3 ← R1 + 8
SUBI R1,#10,R3	R3 ← R1 - 10

El operando inmediato es un valor de 5 bits expresado en C'2. El rango de valores que puede este operando es el siguiente:

Operando inmediato (5 bits – C'2): Rango [-16, 15]

2.3. Direccionamiento con dirección base y desplazamiento: Dir_base$_8$(Ri)

En este modo de direccionamiento, el operando está contenido memoria. La dirección de memoria del operando se calcula como la suma de la dirección base (valor de 8 bits) y el contenido del registro índice (Ri).

Este modo de direccionamiento se puede utilizar como operando fuente en operaciones de LOAD (movimiento de memoria a registro) y como operando destino en operaciones de STORE (movimiento de registro a memoria)

Ejemplos:

Instrucción	**Operación realizada**
LOAD 4(R1),R2	R2 ← Mem(R1 + 4)
STORE R1,10(R2)	Mem(R2 + 10) ← R1

2.4. Direccionamiento absoluto: Dir_absoluta$_8$

En este modo de direccionamiento el operando expresa una dirección de memoria de 8 bits. Este modo de direccionamiento se utiliza únicamente en las instrucciones de salto.

Ejemplos:

Instrucción	Operación realizada
BR 56	PC ← 56

3. Repertorio de instrucciones

3.1. Instrucciones aritmeticológicas

Notación en LE	Operación	Indicadores de Condición
ADDI Rf1, #n, Rd	Rd := Rf1 + n	Z := (Rf1 + n = 0) N := (Rf1 + n < 0)
SUBI Rf1, #n, Rd	Rd := Rf1 - n	Z := (Rf1 - n = 0) N := (Rf1 - n < 0)
ADD Rf1, Rf2, Rd	Rd := Rf1 + Rf2	Z := (Rf1 + Rf2 = 0) N := (Rf1 + Rf2 < 0)
SUB Rf1, Rf2, Rd	Rd := Rf1 - Rf2	Z := (Rf1 - Rf2 = 0) N := (Rf1 - Rf2 < 0)
ASR Rf2, Rd	Rd := Rf2 >> 1	Z := (Rf2 >> 1 = 0) N := $Rf2_{15}$
AND Rf1, Rf2, Rd	Rd := Rf1 ∧ Rf2	Z := (Rf1 ∧ Rf2 = 0) N := (Rf1 ∧ Rf2 < 0)

(NOTA: La instrucción ASR es equivalente a dividir por 2)

3.2. Instrucciones de acceso a memoria:

Notación en LE	Operación	Indicadores de Condición
LOAD dir_base(Ri), Rd	Rd := M[dir_base + Ri]	Z := (M[dir_base+Ri] = 0) N := (M[dir_base+Ri] < 0)
STORE Rf, dir_base(Ri)	M[dir_base + Ri] := Rf	Z y N no cambian.

3.3. Instrucciones de salto

Notación en LE	Condición	Comentarios
BR dir_destino	1	Salto incondicional
BEQ dir_destino	Z	Salta si igual
BL dir_destino	N	Salta si menor
BLE dir_destino	N v Z	Salta si menor o igual
	-	No usado
BNE dir_destino	$\overline{Z}$	Salta si no igual
BGE dir_destino	$\overline{N}$	Salta si mayor o igual
BG dir_destino	$\overline{N \text{ v } Z}$	Salta si mayor

El proceso de ensamblado

1. Directivas de ensamblador

Las directivas de ensamblador controlan acciones auxiliares que se realizan durante el proceso de ensamblado, tales como reservar posiciones de memoria, inicializar posiciones de memoria a un determinado valor, definir etiquetas, indicar el inicio y el final del programa, o definir macros. Las directivas no son traducibles a código máquina.

Directiva LE	Operación
.dw expr1 {, expr2,...,exprN}	Define **N** posiciones de memoria consecutivas con valores iniciales **expr1, ..., exprN**
.rw N	Reserva **N** posiciones de memoria consecutivas no inicializadas
identificador = expresión	Define un **identificador** con el valor asociado a la **expresión**
.begin etiqueta	Indica que la instrucción en la dirección **etiqueta** es la primera a ejecutar
.end	Indica el fin de la ejecución del programa
.org expresión	La siguiente instrucción o dato se almacenará en la dirección indicada por el valor de la **expresión**
.def nombre {parámetros}	Indica el inicio de una **macro** identificada por un **nombre** y con una lista de **parámetros**. Estos parámetros pueden ser de tipo registro ($n), inmediato ($in) o dirección de memoria ($dn)
.enddef	Indica el final de la última **macro iniciada**

2. El ensamblado de programas

El proceso de ensamblado consiste en la traducción de un programa escrito en ensamblador a lenguaje máquina. Esta traducción es directa e inmediata, ya que las instrucciones en ensamblador no son más que nemotécnicos de las instrucciones máquina que ejecuta directamente la CPU.

2.1. Ensamblado de programas sin macros

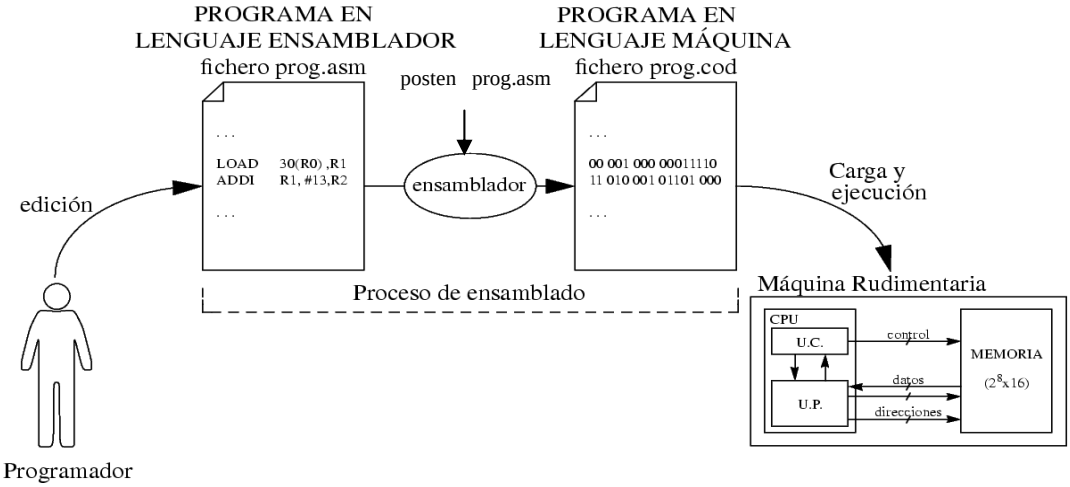
Si el programa no contiene macros, lo editaremos con un editor de texto (por ejemplo, **notepad**) con la extensión **.asm**, por ejemplo: **prog.asm**

Para ensamblar el programa, abrir una ventana MS-DOS y ejecutar la siguiente orden:

posten prog.asm

Esta orden genera un fichero con el mismo nombre, pero con extensión .cod (**prog.cod**). Este fichero contiene el programa binario en lenguaje máquina que se deberá simular con posterioridad en el simulador de la máquina rudimentaria.

El esquema del procedimiento sería el siguiente:



2.2. Ensamblado de programas con macros

Si el programa sí contiene macros, será necesario editar dos ficheros distintos (usando cualquier editor de texto, por ejemplo **notepad**), ambos deberán llamarse con la extensión **.mr**, por ejemplo:

- **prog.mr** que contendrá el código ensamblador del programa
- **macros.mr** que contendrá las macros utilizadas en el programa

Para ensamblar el programa, en primer lugar es necesario llamar al preensamblador, que se encarga de expandir las macros. Para preensamblar el programa, abrir una ventana MS-DOS y ejecutar la siguiente orden:

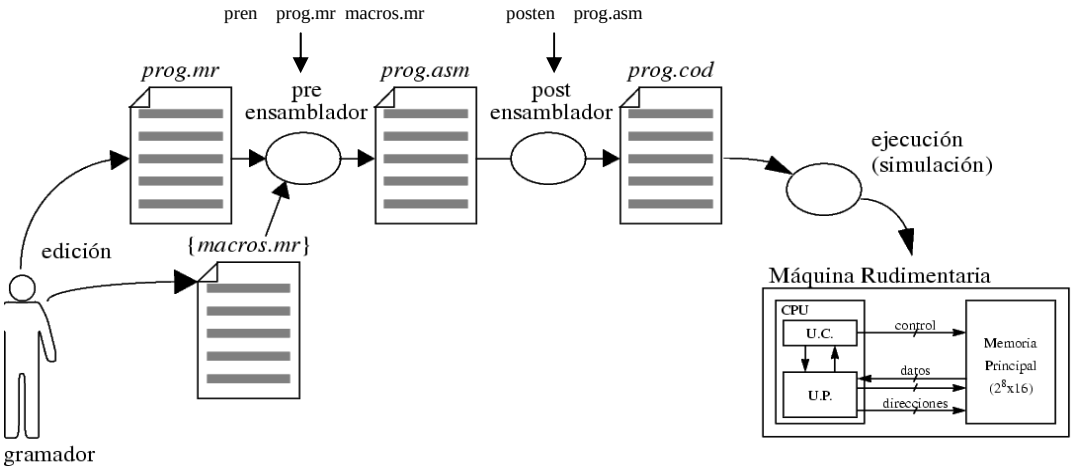
pren prog.mr macros.mr

Esta orden genera un fichero con el mismo nombre que el programa, pero con extensión **.asm** (**prog.asm**), que contiene el código ensamblador con las macros expandidas. A continuación es necesario llamar al programa ensamblador para convertir este programa a código máquina, mediante la siguiente orden

posten prog.asm

Esta orden genera un fichero con el mismo nombre, pero con extensión .cod (**prog.cod**), que contiene el programa binario en lenguaje máquina.

El procedimeinto completo sería el siguiente:

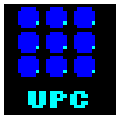


3. Simulador de la máquina rudimentaria

El simulador de la Máquina Rudimentaria ha sido desarrollado por el Departamento de Arquitectura de Computadores de la Universidad Politécnica de Cataluña y se puede descargar de la siguiente dirección:

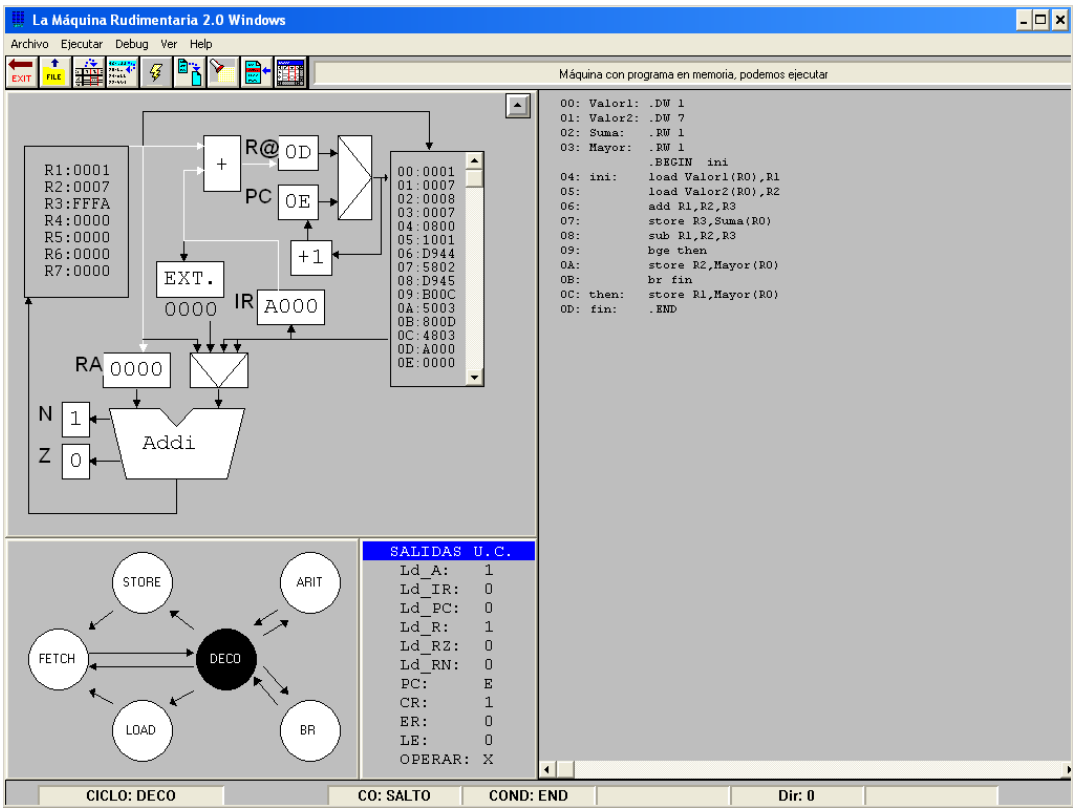
ftp://ftp.ac.upc.es/pub/archives/mr/fuentes

Para arrancar el simulador de la máquina rudimentaria, buscar la aplicación **Mr.exe** con el siguiente icono:



Mr.exe

Se abrirá la siguiente ventana:



3.1. Carga del programa

Utilizar la opción **Cargar** del menú **Archivo** y seleccionar el programa deseado con extensión **.cod**

3.2. Ejecución del programa

Ejecución del programa completo

Para ejecutar el programa completo, de principio a fin, ir al menú **Ejecutar** y seleccionar la opción **Run**. Opcionalmente también se puede utilizar la tecla **F9**

Ejecución instrucción a instrucción

Para realizar una ejecución detallada del programa, instrucción a instrucción, ir al menú **Ejecutar** y seleccionar la opción **Step**. Opcionalmente también se puede utilizar la tecla **F8.**

Esta opción puede ser muy útil para depurar el programa

Ejecución ciclo a ciclo

Para ejecutar realizar una ejecución aún más detallada del programa, distinguiendo entre los distintos ciclos de una instrucción, ir al menú **Ejecutar** y seleccionar la opción **Cicle**. Opcionalmente también se puede utilizar la tecla **F7**.

Nótese que esta opción es interesante únicamente si deseamos realizar un seguimiento del diagrama de estados de la máquina rudimentaria. Si lo que deseamos es depurar el programa, es preferible la ejecución instrucción a instrucción.

Reinicio del simulador (Reset)

Una vez que ha finalizado la ejecución de nuestro programa, si deseamos volver a ejecutarlo debemos reiniciar el simulador. Para ello ir al menú **Ejecutar** y seleccionar la opción **Reset**. Opcionalmente también se puede utilizar la tecla **F10.**

3.3. Depuración del programa: breakpoints

Si estamos depurando un programa que no funciona correctamente, la utilización de breakpoints puede resultar muy útil.

Al establecer un breakpoint en una instrucción del programa, la ejecución de éste se detiene al llegar a dicha instrucción. Posteriormente podemos continuar la ejecución normal del programa o continuar la ejecución instrucción a instrucción.

Para entrar en modo de establecimiento de breakpoints, ir al menú **Debug** y seleccionar la opción **BreakP**. Posteriormente pinchar sobre la instrucción o instrucciones donde se desea establecer un breakpoint.

Para salir del modo de breakpoints, ir de nuevo al menú **Debug** y seleccionar nuevamente la opción **BreakP**.

PRÁCTICA 0: Codificación y depuración de programas en ensamblador.

El objetivo de esta primera práctica es la toma de contacto con las herramientas de ensamblado y simulación de programas ensamblador de la máquina rudimentaria, que se utilizarán para la realización de las prácticas de que consta esta parte de la asignatura.

El desarrollo de un programa se compone de las siguientes etapas:

- Especificación del problema.
- Diseño de un diagrama de flujo orientado al lenguaje objetivo
- Escritura del programa en lenguaje ensamblador (codificación).
- **Edición** del programa fuente.
- **Ensamblado**
- **Simulación**.
- **Depuración** de errores.

En esta primera práctica se aprenderán las nociones básicas necesarias para la realización de las cinco últimas fases, que son las más directamente relacionadas con el trabajo a efectuar en el laboratorio.

En lo que sigue, utilizaremos un problema ejemplo sobre el que basaremos esta práctica. Este ejemplo consiste en la siguiente especificación:

Dados dos números X e Y, calcular:

1.- su suma
2.- el mayor de ellos.

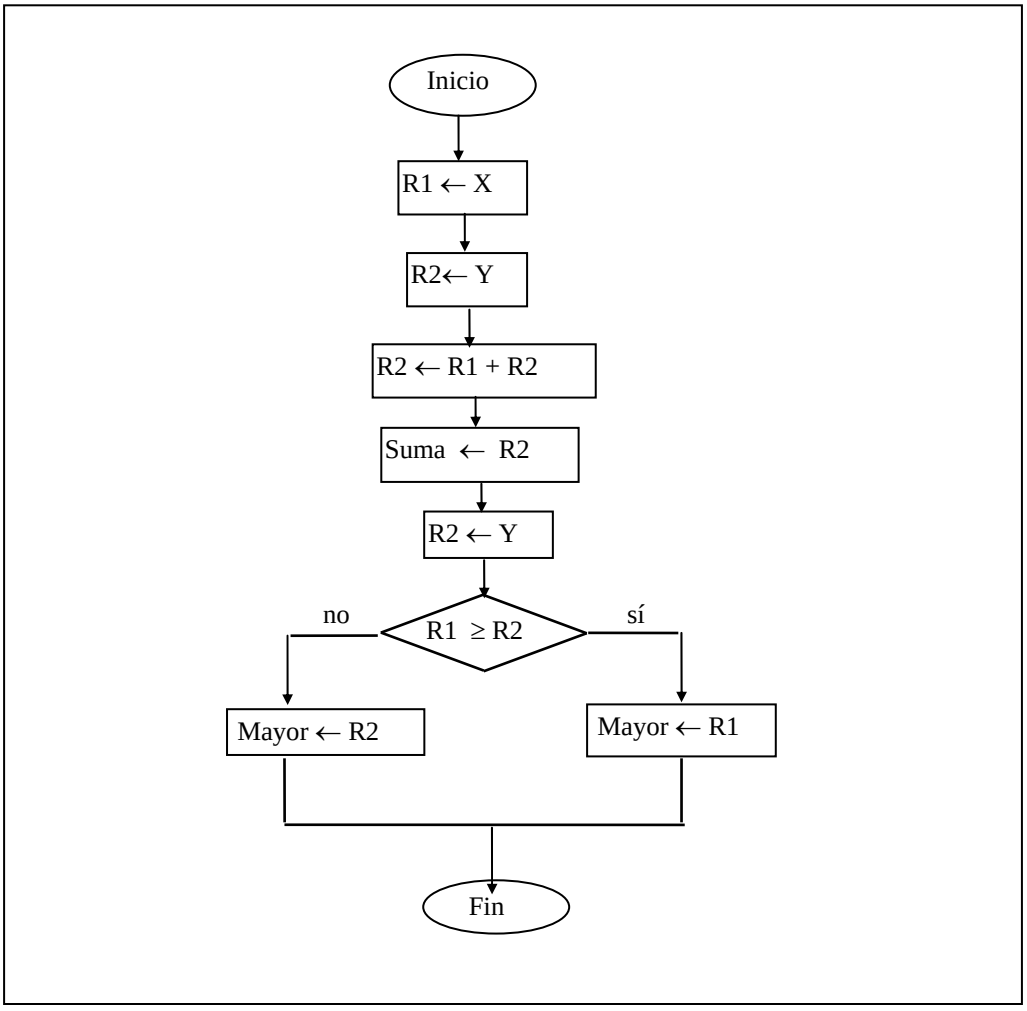
Una especificación más formal del problema es la siguiente:

Dados dos números naturales X e Y, se pide calcular los resultados Suma y Mayor:

$$Suma = X + Y$$

$$Mayor = \begin{cases} X, si\ X \geq Y \\ Y, si\ X < Y \end{cases}$$

Como primera aproximación, realizaremos un diagrama de flujo de las operaciones, en donde detallamos los registros que necesitamos y la colocación espacial de los bloques. Debemos tener en cuenta que un programa en ensamblador no se compone únicamente de instrucciones que expresan de forma abstracta el algoritmo que implementa, sino que, al contrario de lo que ocurre en alto nivel, el programador necesita sopesar las distintas opciones que la arquitectura final ofrece: dónde y cómo almacenar variables, cómo manejar datos y control, etc.



Tras el diagrama de flujo en donde, como vemos, hemos reflejado las etiquetas, las variables intermedias, etc, editaremos el código sustituyendo las expresiones por instrucciones en ensamblador, resultando el siguiente listado, que corresponde al programa fuente:

```
        .BEGIN ini    ; Etiqueta de inicio de programa

Valor1:.DW 2     ; Reserva palabra para Valor1 inicializada a 2
Valor2:.DW 7     ; Reserva palabra para Valor2 inicializada a 7
Suma:  .RW 1     ; Reserva palabra para guardar resultado de Suma
Mayor: .RW 1     ; Reserva palabra para guardar resultado de Mayor


ini:    load Valor1(R0),R1      ; Carga Valor1 en R1
        load Valor2(R0),R2      ; Carga Valor2 en R2
        add R1,R2,R3            ; Suma R1 + R2 y guarda resultado en R3
        store R3,Suma(R0)       ; Guarda en memoria resultado de la suma
        sub R1,R2,R3            ; Realiza resta para comparar valores
        bge then                ; Si R1 >= R2 salta a etiqueta "then"
        store R2,Mayor(R0)      ; R2 > R1 ➔ Guarda R2 en Mayor
        br fin                  ; Salta a fin del programa
then:   store R1,Mayor(R0)      ; R1 >= R2 ➔ Guarda R1 en Mayor
fin:
        .END                    ; Fin de programa
```

NOTAS IMPORTANTES:

1) Las directivas (.BEGIN, .END, .DW, .RW) deben escribirse en **MAYÚSCULA**
2) Al editar el programa, introducir **una línea extra en blanco** al final del código fuente (detrás de la directiva .END)

PRACTICA 1: Programa Simple. Diseño de macros.

OBJETIVOS:

1) Desarrollar un programa simple en ensamblador y verificar su corrección mediante la simulación.
2) Aprender a realizar macros

ESPECIFICACIONES:

Parte A:

Realizar un programa que multiplique dos números naturales de longitud WORD usando la instrucción de suma (ADD).

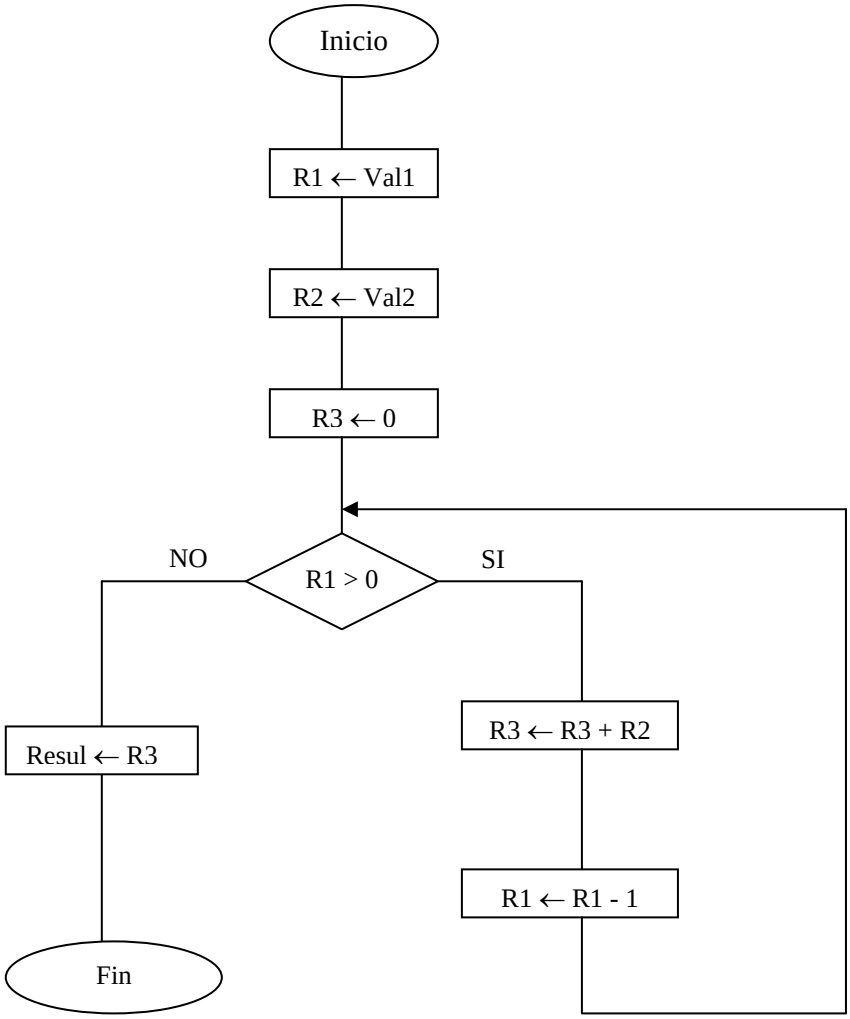
El algoritmo es sencillo y puede expresarse brevemente en pseudo-PASCAL de la siguiente manera:

```
PROGRAM Mult;
CONST
     Val1 = 35;
     Val2 = 58;
VAR
     Mult1, Mult2 : WORD;
     Resul        : DOUBLE WORD;
BEGIN
     Mult1 := Val1;
     Mult2 := Val2;
     Resul := 0;
     WHILE Mult1 > 0 DO BEGIN
         Resul := Resul+Mult2;
         Mult1 := Mult1-1;
     END;
END.
```

Parte B:

Realizar una macro que realice la operación de multiplicación y reescribir el programa para que utilice dicha macro.

El diagrama de flujo asociado a este programa sería el siguiente:


Inicio
R1 ← Val1
R2 ← Val2
R3 ← 0
R1 > 0
NO
SI
Resul ← R3
R3 ← R3 + R2
R1 ← R1 - 1
Fin

SOLUCIÓN:

Parte A:

PRACT1A.ASM

```
        .BEGIN  ini

valor1: .DW 3
valor2: .DW 5
produ:  .RW 1

ini:    load Valor1(R0),R1
        load Valor2(R0),R2
        add R0,R0,R3
bucle:  subi R1,#1,R1
        bl fin
        add R2,R3,R3
        br  bucle
fin:    store R3,produ(R0)

        .END
```

Parte B:

MACROS.MR

```
.DEF  mul $1,$2,$3

          add R0,R0,$3
  bucle:  subi $1,#1,$1
          bl fin
          add $2,$3,$3
          br  bucle
  fin:

.ENDDEF
```

NOTA IMPORTANTE

Al editar la macro, introducir **una línea extra en blanco** al final del código fuente (detrás de la directiva .ENDDEF)

PRACT1B.MR

```
        .BEGIN  ini

valor1: .DW 3
valor2: .DW 5
produ:  .RW 1

ini:  load Valor1(R0),R1
      load Valor2(R0),R2
      mul R1,R2,R3
      store R3,produ(R0)

      .END
```

PRACTICA 2: Datos compuestos: vectores. Estructuras de control I.

OBJETIVO: Estudiar nuevos modos de direccionamiento y estructuras de control.

ESPECIFICACIÓN:

Parte A:

Realizar un programa que multiplique un escalar por un vector de 8 componentes y deposite el resultado en un segundo vector, utilizando la estructura de control "bucle FOR". Para ello se utilizará la macro de multiplicar realizada en la Práctica 1.

IMPORTANTE: tanto el valor escalar como las componentes del vector deben ser números naturales.

Parte B:

Realizar un programa que genere un vector resultado de invertir el orden de las componentes de otro dado (ambos vectores de 8 componentes)

PRACTICA 3: Estructuras de control II.

OBJETIVO: Afianzar los conocimientos de evaluación de condiciones, estructuras de control y direccionamiento de vectores.

ESPECIFICACIÓN:

Diseñar un programa que calcule el máximo común divisor de dos números por el algoritmo de Euclides.

ALGORITMO: (en pseudo-PASCAL)

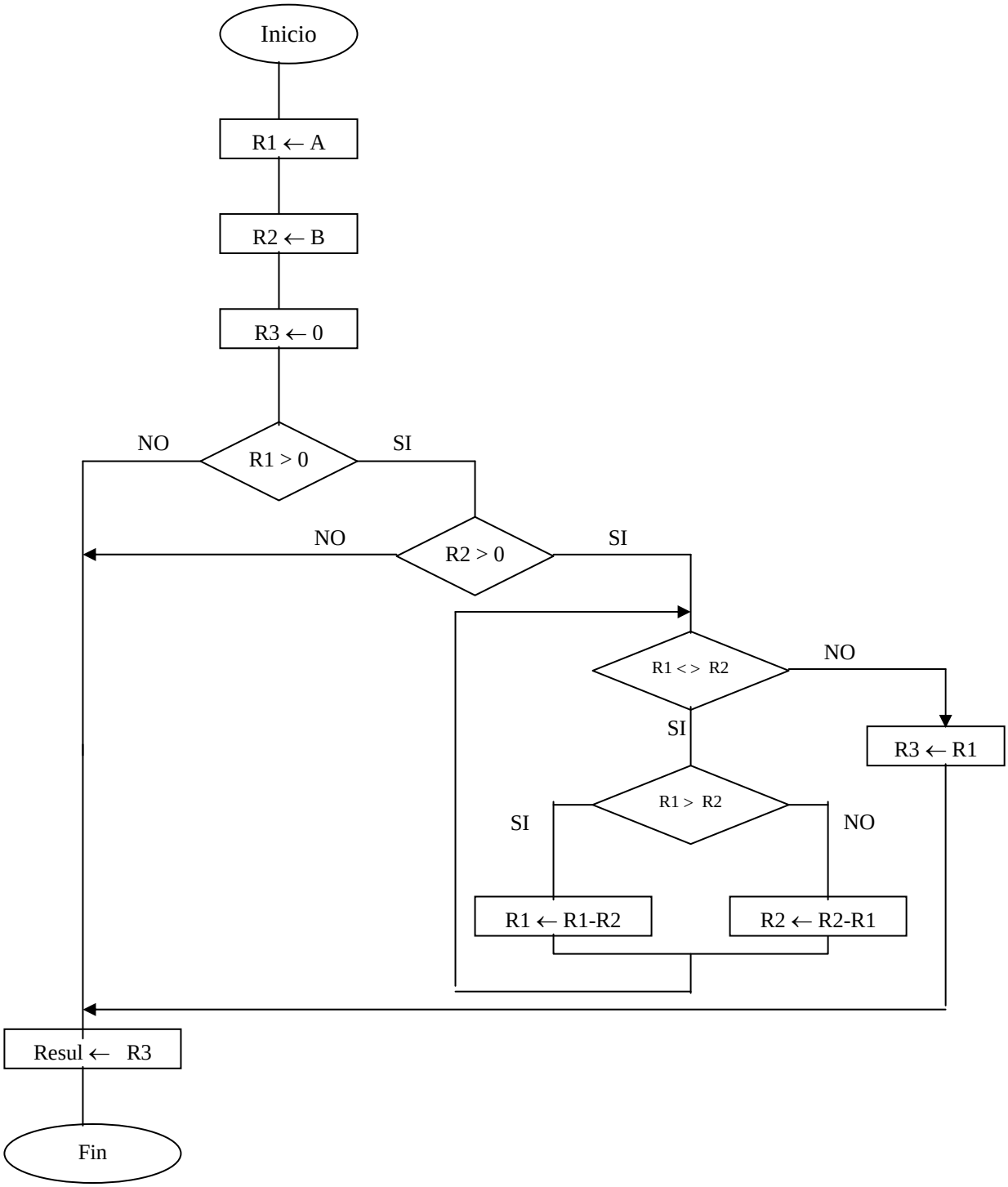
```
PROGRAM Mcd;

CONST    A := 51; B := 595;

VAR      Num1, Num2, Resul : NATURAL.W;

BEGIN
  Num1 := A; Num2 := B;
  IF  Num1 > 0  AND  Num2 > 0 THEN
  BEGIN
    WHILE  Num1 <> Num2  DO
      IF  Num1 > Num2  THEN  Num1 := Num1-Num2
                       ELSE  Num2 := Num2-Num1;
    Resul := Num1
  END
  ELSE  Resul := 0;
END.
```

El diagrama de flujo asociado a este programa sería el siguiente:



PRACTICA 4: Datos compuestos: matrices.

OBJETIVO: Tratamiento de vectores multidimensionales.

ESPECIFICACIÓN:

Diseñar, implementar y verificar un programa en ensamblador que realice la siguiente operación:

- Dada una matriz cuadrada NxN, el programa calcula el número de componentes con valor cero, con valor positivo y con valor negativo que hay **en cada fila de la matriz**. Los resultados se almacenan en tres vectores, Vcero, Vpos y Vneg respectivamente, de dimensión N cada uno de ellos.

- Ejemplo (dimensión 3x3):

$$\text{Matriz}=\begin{pmatrix} 1 & 1 & 0 \\ -1 & 0 & 1 \\ -1 & -1 & -1 \end{pmatrix} \quad \text{Vcero}=\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \quad \text{Vpos}=\begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix} \quad \text{Vneg}=\begin{pmatrix} 0 \\ 1 \\ 3 \end{pmatrix}$$

ALGORITMO: (en pseudo-PASCAL)

```
PROGRAM calcula_cero_pos_neg;

CONST  N = 3;

VAR  A : ARRAY [0..N-1,0..N-1] OF WORD;
     Vcero : ARRAY [0..N-1] OF WORD;
     Vpos : ARRAY [0..N-1] OF WORD;
     Vneg : ARRAY [0..N-1] OF WORD;
BEGIN
  FOR  I := 0 TO N-1  DO
    BEGIN
      Vcero[i] := 0;
      Vpos[i] := 0;
      Vneg[i] := 0;
    END;
  FOR  I := 0 TO N-1  DO
    FOR  J := 0 TO N-1  DO
    BEGIN
      IF  A[i,j] > 0  THEN
        Vpos[i] := Vpos[i] + 1;
      ELSE
          IF A[i,j] < 0  THEN
            Vneg[i] := Vneg[i] + 1;
          ELSE
            Vcero[i] := Vcero[i] + 1;
    END;
END.
```

Diagrama de flujo: