Soluciones comentadas de los problemas del examen de Estructuras de los Computadores (7 de Febrero de 1.997)

1. Se tiene un ordenador con los siguientes formatos de representación:
    - Números enteros con 16 bits, representados en signo-magnitud.
    - Números en coma flotante con las siguientes características:
        - Mantisa expresada en signo-magnitud
        - El ancho de la mantisa es de 16 bits.
        - La mantisa es fraccionaria.
        - El exponente viene representado en signo-magnitud.
        - El ancho del exponente son 8 bits.

    Se pide:

    a) Calcular el rango de representación para los números enteros.

    b) Calcular el rango de representación para los números en coma **flotante empleando la técnica del bit implícito.**

    c) Representar en el formato entero el número:
        - -127.

    d) Representar en formato de coma flotante el número:
        - 14,25.

    e) Calcular el valor del número, suponiendo **que no se ha utilizado la técnica del bit implícito** en su representación:

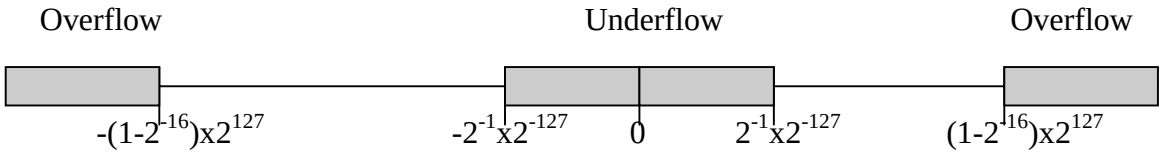
1	0000 0010	111 1000 0000 0000

    f) Se desea proteger el siguiente número: 101 0011 1001 empleando el método del código Hamming.
    Si recibimos el número: 111 1010 0011 1001, sería ¿correcto o incorrecto?

Solución:

a) Números enteros, quiere decirnos coma fija. El rango de los números en coma fija y signo-magnitud es el que podemos deducir de: $[-2^{n-1}+1, 2^{n-1}-1]$. El número n es el número de bits del que disponemos para poder representar el número en coma fija. En el enunciado nos dicen que contamos con 16 bits, representados en signo-magnitud. Esto es que en nuestro número n = 16. Por lo tanto, el rango de los números enteros en este sistema de representación será: $[-2^{16-1}+1, 2^{16-1}-1]= [-2^{16}+1, 2^{16}-1]$, que como podemos comprobar es un rango simétrico.
Números en coma flotante. Los números reales se representan según el formato $V(x) = M \times 2^e$. Es decir, una mantisa, multiplicada por 2 elevado a un exponente. Nos dicen que la mantisa es fraccionaria. En el caso de la representación de signo-magnitud una mantisa fraccionaria estará normalizada si tenemos algo de la forma ,1xxxxxx.....xxxx. Es decir, una coma, un uno y después cualquier combinación de ceros y unos (se representa por una x). En signo-magnitud el rango es simétrico, con lo que tendremos el mismo número de valores positivos que de valores negativos. Entonces nos podemos ahorrar un cálculo, ya que podemos calcular el rango de los positivos y con sólo afectarles del signo tendremos el de los negativos.
Empecemos por el menor número representable en mantisa normalizada fraccionaria en signo-magnitud y empleando la técnica del bit implícito. Tendremos una mantisa de la forma ,1xx.....xx. Como estamos en el caso de la mantisa más pequeña tendremos el valor ,100...00. Pero, ¿cuántos ceros? Nos dicen que la mantisa son 16 bits, pero uno se reserva para el signo; por tanto, tendremos 15 bits para representar la mantisa, pero también se nos pide que empleemos la técnica del bit implícito, con

lo que el número de bits que tenemos que tener en cuenta será de 16. Por tanto, la menor mantisa será: ,1000 0000 0000 0000. Y, ¿cuál es su valor? Tenemos un único uno, está afectado del peso -1, por lo que valdrá 2^{-1}, o lo que es igual 0,5.
En el caso de la mantisa más grande, tendremos ,111.....1. Es decir, en total 16 unos. Y valdrá: $1 - 2^{-n}$, siendo n el bit afectado del peso más negativo; en nuestro caso, -16. Por tanto, la mayor mantisa positiva valdrá $1-2^{-16}$, o lo que es igual, algo muy próximo a uno.
Resumiendo, el rango para la mantisa de los números positivos será $[2^{-1}, 1-2^{-16}]$, y al ser el rango simétrico, para los negativos será de $[-(1-2^{-16}), -2^{-1}]$
El exponente está en signo-magnitud con 8 bits. Recordando el apartado a) tendremos un rango de [-127, 127**]**.



c) Tenemos que representar los números -127. Como se trata de un número entero, emplearemos la representación de coma fija. Para ello teníamos 16 bits. Con lo que el número queda: 1 0000 0000 1111 111. Hemos subrayado el primer bit para indicar el signo.
d) En el caso del 14,25 iremos por pasos. De los 16 bits de la mantisa uno será para el signo y el resto para la magnitud. El signo es positivo luego el valor del bit de signo será 0. La magnitud del número es de 14,25; que en binario vale 01110,01. Entonces, el -14,25 será un número de la forma 1xxx....xxxxx01110,01; pero vemos que no se encuentra normalizado.
Pero como en signo-magnitud tendremos un número normalizado cuando se tenga una expresión s,1xxx ... xxxx, deberemos ajustar el exponente. Es decir es como si tuviésemos el número multiplicado por 2^4. Con ese valor del exponente el número ya es fraccionario y se encuentra normalizado.
El exponente se representa en signo-magnitud. Con lo que el exponente 4 se representará como: 0 000 0100.
En definitiva el número se representará como sigue:

Signo	Exponente	Mantisa fraccionaria normalizada
1	0 000 0100	1110 0100 0000 000

e) Debemos de saber cual es el valor del número:

1	0000 0010	111 1000 0000 0000

Vemos claramente diferenciados el bit de signo, el exponente y la mantisa. El valor del número es de la forma $V(x) = signo Mx2^e$.
Empecemos por el signo, tiene un valor 1 lo que indica que el número es negativo.
En cuanto al exponente vemos que el número representado vale 2.
Finalmente veamos cuanto vale la mantisa.

$1-2^{-4} = 3,75$

Si juntamos todas las piezas tendremos que el número vale -3,75.

f) Se desea proteger el siguiente número: 101 0011 1001 empleando el método del código Hamming.
Si recibimos el número: 111 1010 0011 1001, sería ¿correcto o incorrecto?

Para ver cuantos bits de paridad debemos añadir según Hamming, vemos que los bits de datos son 11, deberemos satisfacer la inecuación $2^p \geq p + n + 1$. O lo que es lo mismo, $2^p \geq p + 11 + 1$, con lo que $p = 4$.

1	**1**	1	**1**	0	1	0	**0**	0	1	1	1	0	0	1
$\mathbf{b_1}$	$\mathbf{b_2}$	b_3	$\mathbf{b_4}$	b_5	b_6	b_7	$\mathbf{b_8}$	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}

Los bits de paridad están resaltados en negrita. Cada uno de ellos protegen a los bits siguientes:
$\mathbf{b_1} = b_3 \oplus b_5 \oplus b_7 \oplus b_9 \oplus b_{11} \oplus b_{13} \oplus b_{15} = 1$ *correcto.*
$\mathbf{b_2} = b_3 \oplus b_6 \oplus b_7 \oplus b_{10} \oplus b_{11} \oplus b_{14} \oplus b_{15} = 1$ *correcto.*
$\mathbf{b_4} = b_5 \oplus b_6 \oplus b_7 \oplus b_{12} \oplus b_{13} \oplus b_{14} \oplus b_{15} = 1$ *correcto.*
$\mathbf{b_8} = b_9 \oplus b_{10} \oplus b_{11} \oplus b_{12} \oplus b_{13} \oplus b_{14} \oplus b_{15} = 0$ *correcto.*

Con lo que el número es correcto.

2. Realizar un programa en ensamblador que lea un número hexadecimal por teclado e imprima las cifras desde ese número hasta el 1. Se supone que el número introducido por teclado está comprendido entre 3h y 9h, que es impar, y que cualquier otro valor no será aceptado, lo que se deberá indicar mediante un mensaje

   Ejemplos:
   c:\>**programa**
   5
   54321
   c:\>**programa**
   C
   El número introducido no es válido, debe de estar comprendido entre 3 y 9 y ser impar.

NOTA: las letras en negrita indica lo que se debería introducir por teclado.

```
;
;
; A continuación declaramos el segmento de datos.
;
Datos SEGMENT
  LeerCifra          EQU 01h
  EscribeCifra       EQU 02h
  EscribeCadena      EQU 09h
  Terminar           EQU 4C00h
  ANumero            EQU 30h
  Impar              EQU 01h
  Tres               EQU 33h
  Nueve              EQU 39h
  LF                 EQU 0Ah
  CR                 EQU 0Dh
  MsgPideNumero  DB 'Introduzca un impar de una cifra entre 3 y 9',LF,CR,'$'
  MsgError1   DB 'El número $'
```

```
   MsgError2   DB ‘ introducido, no es válido’, LF, CR, ‘$’
   MsgResultado        DB ‘El resultado de la operación será:’, LF, CR, ‘$’
Datos ENDS
;
Pila SEGMENT STACK
         DB   1024  DUP (0)
Pila ENDS
;;
; Una posible implementación del segmento de código es la siguiente:
;
Codigo SEGMENT
   ASSUME CS:Codigo, DS:Datos, SS:Pila   ;Inicialización de   los
                                          ;segmentos del programa.
PRINCIPAL PROC FAR
   MOV AX, Datos
   MOV DS, AX
  XOR AX, AX
  CALL LeerNumero
  CALL ImprimeNumeros
  CALL TerminarPrg
   RET
PRINCIPAL ENDP
;
LeerNumero PROC NEAR
; Leemos un número por teclado. Sacando primero un mensaje que lo solicite.
   CALL EscribePideNumero
   MOV AH, LeerCifra
   INT 21h
  CALL Comprobar
  RET
LeerNumero ENDP
;
Comprobar PROC NEAR
  TEST AL, Impar
  JNZ CotaInferior
  CALL Finalizar
CotaInferior:
  CMP AL, Tres
  JGE CotaSuperior
  CALL Finalizar
CotaSuperior:
  CMP AL, Nueve
  JLE Final
  CALL Finalizar
Final:
  RET
Comprobar ENDP
```

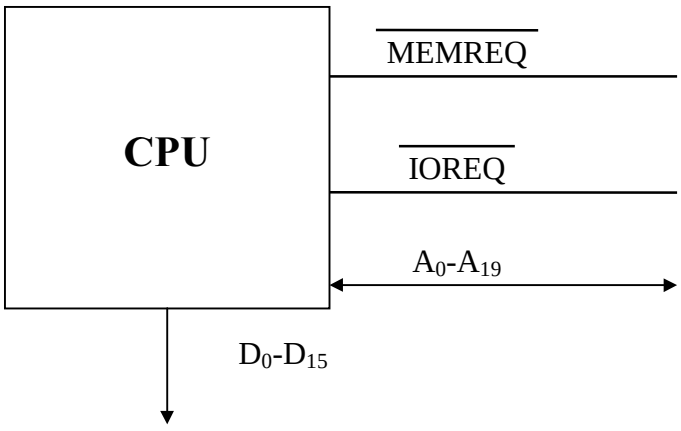
```
Finalizar PROC NEAR
  CALL ImprimeError
  CALL TerminarPrg
   RET
Finalizar ENDP
;
Terminar PROC NEAR
  MOV AX, Terminar
  INT 21h
  RET
Terminar ENDP
;
ImprimeNumeros PROC NEAR
  PUSH AX
  CALL Enter
  CALL EscribeResultado
  POP AX
  SUB AL, ANumero
  XOR CX, CX
  MOV CL, AL
  MOV AH, EscribeCifra
  POP DX
Bucle:
  INT 21h
  DEC DL
  LOOP Bucle
  RET
ImprimeNumeros ENDP
;
ImprimeError PROC NEAR
  PUSH AX
  CALL Enter
  MOV AH, EscribeCadena
  LEA DX, MsgError1
  INT 21h
  MOV AH, EscribeCifra
  POP DX
  INT 21h
  MOV AH, EscribeCadena
  LEA DX, MsgError2
  INT 21h
  RET
ImprimeError ENDP
```

```
Enter PROC NEAR
  MOV AH, EscribeCifra
  MOV DL, LF
  INT 21h
  MOV DL, CR
  INT 21h
  RET
Enter ENDP
;
EscribePideNumero PROC NEAR
  MOV AH, EscribeCadena
  LEA DX, MsgPideNumero
  INT 21h
  RET
EscribePideNumero ENDP
;
EscribeResultado PROC NEAR
  MOV AH, EscribeCadena
  LEA DX, MsgPideNumero
  INT 21h
  RET
EscribeResultado ENDP
Codigo ENDS
END
```

3. La CPU de la figura cuenta con un ancho de palabra de 16 bits. Se quiere dotar a esa CPU de una memoria, sin multiplexación de buses, con las siguientes características:
   - 128 k palabras (128 k x 16) de memoria ROM.
   - 640 k palabras (640 k x 16) de memoria RAM.

   La ROM debe estar antes que la RAM.
   Diseñar la memoria con el menor número de pastillas sabiendo que disponemos de las siguientes pastillas:

Pastillas de memoria ROM	Pastillas de memoria RAM
128 k x 1	128 k x 1
64 k x 8	256 k x 8
128 k x 16	256 k x 16



Solución

1. Comprobar que nos piden algo que realmente se pueda hacer.

Para poder comprobarlo, debemos fijarnos en el número de bits que tenemos en el bus de direcciones, y ver que con ese número de bits, podemos direccionar la memoria que se nos pide.

El bus de direcciones tiene las líneas desde la A_0 hasta la A_{19}, en total 20 bits. Nosotros necesitamos direccionar 640k palabras de memoria RAM y 128k palabras de memoria ROM, luego debemos de ser capaces de direccionar 640k + 128k de memoria total, RAM y ROM.

Para que seamos capaces de poder direccionar 640k + 128k = 768k, necesitaremos 20 bits (2^n ### 1024k, de donde n = 20). Por lo tanto, vemos que con los 20 bits del bus de direcciones **si** que podemos direccionar la memoria que se nos pide.

Tenemos que comprobar que el valor del bus de datos también sea correcto. Como queremos que cada posición de memoria almacene una palabra (16 bits), necesitaremos que el bus de datos nos soporte el ancho de la palabra.

El bus de datos tiene las líneas desde D_0 hasta D_{15}, en total 16 bits. Por tanto, también podemos acceder a posiciones de memoria en las que se almacenen 16 bits.

2. Calcular el menor número de módulos de memoria que nos harán falta.

Para la memoria RAM.

Disponemos de los siguientes módulos de memoria.

a) 128k x 1
b) 256k x 8
c) 256k x 16

640k = 5 pastillas del tipo a) para obtener el mapa de memoria.
128k

640k=2’5 ≈ 3 pastillas de los tipos b) y c)
256k

Pero además de direccionar 640k, debemos ser capaces de almacenar en cada posición de memoria 16 bits, es decir, una palabra.

Por consiguiente, para almacenar 16 bits con elementos del tipo a) - que solamente almacenan 1 bit- necesitaremos 16 módulos. Pero para almacenar 16 bits con elementos del tipo b) únicamente necesitaremos 2 módulos. Y para almacenar esa cantidad con módulos de tipo c) necesitaré 1 módulo.

Finalmente, el mínimo número de módulos lo calcularemos a partir del número de módulos necesarios para direccionar el mapa de memoria que nos piden y del número de módulos necesarios para almacenar el número de bits solicitado.

Con los datos del enunciado necesitaremos:

Módulos del tipo 128k x 1.

· 5 módulos de 128k (para poder direccionar 640k) x 16 (módulos necesarios para almacenar un byte), en total 5 x 16 = 80 módulos del tipo a)

Módulos del tipo 256k x 8.

· 3 módulos de 256k (para poder direccionar 640k) x 2 (módulos necesario para almacenar dos bytes), en total 3 x 2 = 6 módulos del tipo b)

Módulos del tipo 256k x 16.

· 3 módulos de 256k (para poder direccionar 640k) x 1 (módulos necesario para almacenar dos bytes), en total 3 x 1 = 3 módulos del tipo c)

Para la memoria ROM.

Disponemos de los siguientes módulos de memoria.

a) 128k x 1.
b) 64k x 8.
c) 128k x 16.

128k = 1 pastillas de los tipos a) y c) para obtener el mapa de memoria.
128k

128k= 2 pastillas del tipo b)
64k

Pero además de direccionar 128k, debemos ser capaces de almacenar en cada posición de memoria 16 bits, es decir, una palabra.

Por consiguiente, para almacenar 16 bits con elementos del tipo a) - que solamente almacenan 1 bit- necesitaremos 16 módulos. Para almacenar 16 bits con elementos del tipo b) únicamente necesitaremos 2 módulos. Y con elementos del tipo c) necesitaré 1 módulo para poder direccionar 16 bits.

Finalmente, el mínimo número de módulos lo calcularemos a partir del número de módulos necesarios para direccionar el mapa de memoria que nos piden y del número de módulos necesarios para almacenar el número de bits solicitado.

Con los datos del enunciado necesitaremos:
Módulos del tipo 128k x 1.

· 1 módulo de 128k x 16 (módulos necesarios para almacenar un byte), en total 1 x 16 = 16 módulos del tipo a)

Módulos del tipo 64k x 8.

· 2 módulos de 64k x 2 (módulos necesario para almacenar dos bytes), en total 2 x 2 = 4 módulos del tipo b)

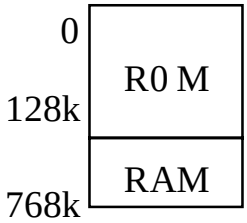
Módulos del tipo 128k x 16.

· 1 módulo de 128k x 2 (módulos necesario para almacenar dos bytes), en total 1 x 1 = 1 módulo del tipo c)

Entonces necesitaremos 3 módulos de 256k x 16 para la memoria RAM (de uno de ellos solamente emplearemos 128 k) y 1 módulo de 128k x 16 para la memoria ROM.

3. Diseñar el mapa de memoria.

Para diseñar el mapa de memoria, aunque no nos indican nada, haremos que la ROM esté antes que la RAM, tal y como reflejamos en la siguiente figura.



Además, para direccionar la memoria que se nos pide, necesitábamos 20 bits.

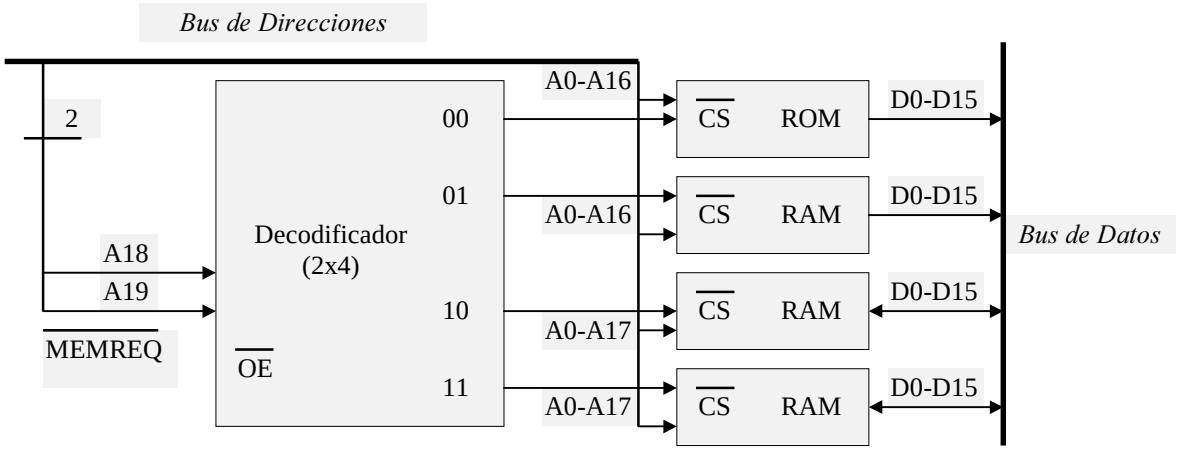
A_{19}	A_{18}	A_{17}	A_{16}	.....	A_1	A_0		
0	0	X	0	.....	0	0	Mínima dirección.	1ª fila de pastillas:
			1		1	1	Máxima dirección.	ROM (0k-128k)
0	1	X	0	.....	0	0	Mínima dirección.	2ª fila de pastillas:
			1		1	1	Máxima dirección.	RAM (128k-256k)
1	0	0	0	.....	0	0	Mínima dirección.	3ª fila de pastillas:
		1	1		1	1	Máxima dirección.	RAM (256k-512k)
1	1	0	0	.....	0	0	Mínima dirección.	4ª fila de patillas:
		1	1		1	1	Máxima dirección.	RAM (512k-768k)

Para poder direccionar 256k necesitaremos 18 bits, que se corresponden con las líneas desde la A_0 hasta la A_{17} del bus de direcciones ($256k = 2^{18}$)

Mientras que para poder acceder a 128k necesitaremos 17 bits, son las líneas desde la A_0 hasta la A_{16} del bus de direcciones ($128k = 2^{17}$)

Si miramos la tabla del mapa de memoria que estamos creando, observamos que solamente nos quedan las líneas A_{18} y A_{19} del bus de direcciones para poder distinguir entre las 4 filas de pastillas y que con 2 bits podemos distinguir $2^2 = 4$ elementos. Si nos fijamos en la parte de la tabla remarcada en gris, existen algunas combinaciones de la línea A_{17} que no se emplean.
Las combinaciones de la línea A_{17} que corresponden a la memoria ROM no se emplean. Ya que hemos visto que con las líneas A_{18} y A_{19} ya nos sirve.

4. Dibujar el esquema.



5. Especificarlas líneas que sean necesarias.

Las líneas que debemos escoger son:

- L/E lectura o escritura en las pastillas RAM.
- Lectura en las pastillas ROM.