Soluciones comentadas de los problemas del examen de Estructuras de los Computadores (1 de Febrero de 1.996)

1) Se tiene un ordenador con los siguientes formatos de representación:
    - Números enteros con 24 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 32 bits.
        - La mantisa es fraccionaria.
        - El exponente viene representado en exceso.
        - 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.. En los siguientes casos:
        - Con bit implícito.
        - Sin bit implícito.

    c) Representar en dicho formato de representación, sin bit implícito, los números:
        - 125
        - -125,25

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

1	10000110	1001 0111 0000 0000 0000 0000 0000 000

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 24 bits, representados en signo-magnitud. Esto es que nuestro número n = 24. Por lo tanto, el rango de los números enteros en este sistema de representación será: $[-2^{24-1}+1, 2^{24-1}-1]= [-2^{23}+1, 2^{23}-1]$, que como podemos comprobar es un rango simétrico.

b) Números en coma flotante, nos quieren decir números reales. 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, con lo que como ejemplo, tendremos que si queremos representar el 32,5 lo podemos representar como $0{,}325 \times 10^2$. Siendo 0,325 la mantisa M y 2 el exponente e. Además, si queremos emplear la técnica del bit implícito, deberemos de tener la mantisa normalizada.

   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 sin emplear 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 32 bits, pero uno se reserva para el signo; por tanto, tendremos 31 bits para representar la mantisa, es decir, será: ,1000 0000 0000 0000 0000 0000 0000 000. 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 31 unos. Y valdrá: $1 - 2^{-n}$, siendo n el bit afectado del peso más negativo; en nuestro caso, -31. Por tanto, la mayor mantisa positiva valdrá $1-2^{-31}$, o lo que es igual, algo muy próximo a uno.

Resumiendo, el rango para la mantisa de los números positivos será $[2^{-1}, 1-2^{-31}]$, y al ser el rango simétrico, para los negativos será de $[-(1-2^{-31}), -2^{-1}] = [2^{-31}-1, -2^{-1}]$. Siendo este el caso de no utilizar la técnica del bit implícito.

Si empleásemos la técnica del bit implícito, contaríamos con un bit más que nos sacamos de la manga. Bueno realmente no es así, pero ya que sabemos que los números están normalizados, y que por lo tanto, en signo-magnitud deberemos tener una mantisa de la forma ,1xxxxxx.....xx entonces, no representamos ese uno, y toda la mantisa servirá para almacenar dígitos del número. En definitiva, será como si realmente si tuviéramos un bit más, a la hora de calcular el rango o el valor de un número.

Calculemos el rango con bit implícito en nuestro ejemplo. Es como si tuviésemos ,1xxx.....xxxx. Pero esta vez, serán 31 x y el 1 es el bit implícito. Si volvemos a calcular el mayor y el menor número representable tendremos:

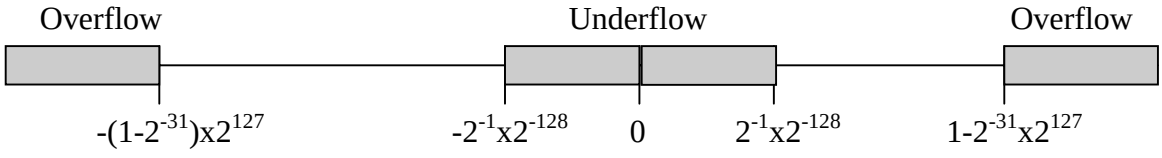
,1000 0000 0000 0000 0000 0000 0000 0000 para el menor, es decir, 2^{-1} y:

,1111 1111 1111 1111 1111 1111 1111 1111 para el mayor, que es igual a $1-2^{-32}$.

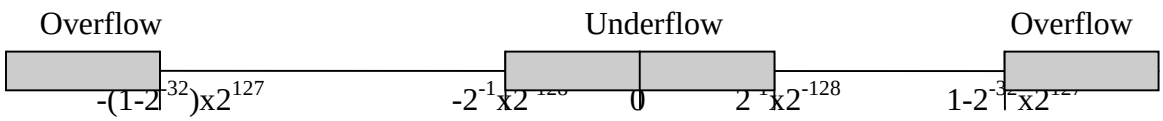
Si este es el rango para los números positivos, también lo será para los negativos, sólo que precedidos del signo menos. Tendremos como rango positivo $[2^{-1}, 1-2^{-32}]$ y para los negativos $[-(1-2^{-32}), -2^{-1}]$.

Dado que la técnica del bit implícito solamente afecta a la mantisa, el rango del exponente será igual en ambos casos. Tenemos un exponente, representado en exceso y con 8 bits. En definitiva su rango será: $[-2^{n-1}, 2^{n-1}-1] = [-2^{8-1}, 2^{8-1}-1] = [-128, 127]$

Entonces, la recta real para el caso de mantisa normalizada fraccionaria sin bit implícito tendremos.



Y en el caso de que si estemos empleando la técnica del bit implícito:



c) Tenemos que representar los números 125 y -125,25. Empecemos por el primero. Como se trata de un número entero, emplearemos la representación de coma fija. Para ello teníamos 24 bits. Uno 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 125, que en binario vale 111 1101. Pero como tenemos 24 bits en total, el número quedará representado de la forma: 0xxx xxxx xxxx xxxx x111 1101. ¿Qué valor tomarán esas x? Si tomasen el valor 1, como la magnitud de este sistema de representación es binario puro, entonces, el número sería cualquier cosa menos el 125. Por lo tanto, deberemos cambiar las x por ceros. Es decir, nos quedará el número: 0000 0000 0000 0000 0111 1101. Hemos subrayado el cero del bit de signo para diferenciarlo.

Bien, vamos ahora con el número -125,25. El signo menos lo indicaremos colocando un 1 en el bit de signo. Hemos visto que el 125 en binario puro se representaba como 111 1101. Por tanto, solamente nos queda ver como representar el 0,25. Si hacemos el proceso de conversión llegaremos a: ,01. Dicho proceso es iterar repetidamente los siguientes pasos.

1. Multiplicar la parte fraccionaria por la base a la que convirtamos.
2. Tomar la parte entera y colocarla como parte del número convertido.
3. Repetir los pasos hasta que o bien, el número no tenga parte fraccionaria o se nos acaben los dígitos que tenemos para representar el número.

En nuestro caso, convertimos el número 0,25 a base 2.
0,25 x 2 = **0**,5 tomamos la parte entera y la colocamos como el primer dígito a la derecha de la coma, es decir, de momento nuestro primer dígito será ,0 ...
0,5 x 2 = **1**,0 hemos terminado, al tomar la parte entera no queda un número fraccionario, sino un 0. Entonces el 0,25 se convertirá en base 2 a ,01.
Hemos puesto en negrita la parte entera del número para hacer más fácil que se siga el proceso de los pasos 1 a 3.
Entonces, el -125,25 será un número de la forma 1xxx....xxxxx111 1101,01. Pero vemos que no se encuentra normalizado. Además tenemos esa x que no sabemos con que rellenarlas. Para que el número estuviese normalizado deberíamos tener 1,111 1101 01 xxxx ... xxx. Pero esas x deberían ser ceros. Si recordamos cuando representábamos el número en coma fija, indicábamos que la magnitud se representaba en binario puro ya que teníamos un bit de signo independiente para poder indicar si el número era positivo o negativo.
Pero como en signo-magnitud tendremos un número normalizado cuando se tenga una expresión s,1xxx ... xxxx, deberemos ajustar el exponente para pasar de la expresión 1xxx....xxxxx111 1101,01 a la final 1,111 1101 01 xxxx ... xxx. Pero, ¿qué ocurre aquí? Deberemos cambiar las x por ceros es decir, llegar a algo de la forma: 1 000 0000 0000 0000 0000 0001 1111 01,01 y no se encuentra normalizado ni es fraccionario. Para que sea fraccionario tendremos que dividir el número 29 veces 1 ,000 0000 0000 0000 0000 0001 1111 0101; o lo que es lo mismo, multiplicar por 2^{29}, ya que al desplazar la coma a la izquierda dividimos el número por 29 con lo que deberemos multiplicarlo por 29 para dejar el mismo valor del número.
Pero ese número no está normalizado, para que se encontrase normalizado deberíamos tener 1 ,111 1101 0100 0000 0000 0000 0000 0000; con lo que habríamos vuelto a desplazar la coma a la derecha, es decir habríamos multiplicado 22 veces por 2, y el resto lo rellenamos con ceros porque la magnitud

se representa en binario puro. En definitiva, hemos desplazado el exponente 29 veces a la izquierda y 22 a la derecha, o lo que es lo mismo 7 veces a la izquierda (29-22) en total. Es decir es como si tuviésemos el número multiplicado por 2^7. Con ese valor del exponente el número ya es fraccionario y se encuentra normalizado. ¿Hemos terminado?

No hemos terminado, porque también nos dicen que el exponente se representa en exceso. Esta representación lo que hace es sumar el valor del número (o restar si el número es positivo) a una determinada cantidad, el exceso. Como nos dicen que el exponente son 8 bits, el exceso es $2^{n-1} = 2^{8-1} = 2^7 = 128$, que será el número al que tenemos que sumar el exponente ajustado. Es decir, en exceso el exponente 7 se representará como 128 + 7 = 135, que en binario es: 1000 0111.

En definitiva el número se representará como sigue:

Signo	Exponente	Mantisa fraccionaria normalizada
1	1000 0111	111 1101 0100 0000 0000 0000 0000 0000

Si nos fijamos, existe un pequeño truco, una vez que sabemos con que se rellenan los bits sobrantes, las x que hemos puesto, el número de veces que debemos desplazar a la derecha o a la izquierda la coma (que es ajustar el exponente) es igual a normalizar el número original. Si observamos con detenimiento, el número -125,25 era el 1 111 1101,01 y si queremos que se normalice podemos desplazar la coma 7 veces a la izquierda, que es el resultado que hemos obtenido al final.

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

1	10000110	1001 0111 0000 0000 0000 0000 0000 000

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 134. Pero nuestro exponente estaba en exceso, con lo que si el exceso era 128, tenemos que:

Exceso + Número = Número en exceso => 128 + N = 134 => N = 6. Es decir, el exponente vale 6.

Finalmente veamos cuanto vale la mantisa. Tenemos dos unos aislados y luego una tira de dos o más unos. Los unos aislados valen su peso, y la tira de 2 o más unos vale $2^{-\text{peso último cero antes del primer 1}} - 2^{-\text{peso del último uno de la tira}}$. Si sumamos todos los componentes del número tendremos que la mantisa vale:

$$2^{-1}+2^{-4}+(2^{-5}-2^{-8}) = 0{,}58984375$$

Si juntamos todas las piezas tendremos que el número vale - 0,58984375 x 2^6, y si lo operamos será el número -37,75.

2) Realizar un programa en ensamblador que lea un número hexadecimal por teclado e imprima esa cifra ese número de veces. Se supone que el número introducido por teclado está comprendido entre 1h y Fh.

```
;
;
; A continuación declaramos el segmento de datos.
;
Datos SEGMENT
  Leer                  EQU 01h
  Escribir              EQU 02h
  He_Leido_Numero       EQU 3Ah
  Corrige_Numero        EQU 30h
  Corrige_Letra         EQU 07h
  Salto_De_Linea        EQU 0Ah
  Enter                 EQU 0Dh
  Codigo_ASCII_Leido     DB  00h
  Numero_Leido           DB  00h
Datos ENDS
;
;
; Una posible implementación del segmento de código es la siguiente:
;
Codigo SEGMENT
  ASSUME CS:Codigo, DS:Datos   ;Inicialización de   los
  MOV AX, Datos                ;segmentos del programa.
  MOV DS, AX
;
; Leemos un número que se supone comprendido entre 01h y 0Fh por teclado.
;
  XOR AX, AX
  MOV AH, Leer
  INT 21h
;
; Salvaguardamos el número leído por teclado en la variable que tenemos
; para ese fin. Realmente  lo que hemos  leído no es  el número en sí, sino
; su código ASCII.
;
  MOV Codigo_ASCII_Leido, AL
;
; Convertimos el  código  ASCII  en el número  de veces que se debe mostrar
; en pantalla. Para  ello, debemos de  comprobar  si el número se corresponde
; con  el  código ASCII  de  una  letra (y  le   restaremos 37h para  obtener  su
; valor numérico)o un número (en cuyo caso se  le restara 30h). El  valor  con
; el  que   debemos   comparar  es  3Ah, ya  que   el  código  ASCII  que   se
; corresponde  con los números va desde el 30h hasta el 39h. Y almacenamos
; el valor corregido en la variable Numero_Leido.
;
  CMP AL, He_Leido_Numero
```

```
    JLE Era_Numero
    SUB AL, Corrige_Letra
Era_Numero:
    SUB AL, Corrige_Numero
    MOV Numero_Leido, AL
;
; Ahora saltamos de línea para que el resultado lo escriba en la línea
; siguiente. Para ello deberemos de escribir un 0Ah y después un 0Dh.
;
    XOR AX, AX
    XOR DX, DX
    MOV AH, Escribir
    MOV DL, Salto_De_Linea
    INT 21h
    MOV DL, Enter
    INT 21h
;
; Tenemos ahora en AL el verdadero  valor del  número que  se ha leído por
; teclado. Como tenemos que  escribir el número  ese t antas veces  como el
; valor  que  hemos  leído, debemos  emplear un  bucle. El  valor  inicial del
; bucle  será  precisamente  el contenido de  AL. Y el  valor que  deberemos
; imprimir es el número que leímos antes pero almacenado en DL. También
; deberemos de indicar que deseamos escribir y no leer.
;
    XOR AX, AX
    XOR CX, CX
    MOV AH, Escribir
    MOV DL, Codigo_ASCII_Leido
    MOV CL, Numero_Leido
Bucle_De_Escribir:
    INT 21h
    LOOP Bucle_De_Escribir
;
; Finalmente deberemos de indicar al sistema operativo que se ha terminado
; el programa. Para ello se emplea la interrupción 21h y se almacena en AH
; el valor 4Ch.
;
    XOR AX, AX
    MOV AH, 4Ch
    INT 21h
;
; Con todo lo anterior ya hemos terminado el segmento de código.
;
Codigo ENDS
```

2) END

3) La CPU de la figura cuenta con un ancho de palabra de 16 bits. Se quiere dotar a esa CPUde una memoria con las siguientes características:

- 256 k palabras (256 k x 16) de memoria ROM.
- 512 k palabras (512 k x 16) de memoria 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 64 k x 8 128 k x 8	128 k x 1 256 k x 8

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 512k palabras de memoria RAM y 256k palabras de memoria ROM, luego debemos de ser capaces de direccionar 512k + 256k de memoria total, RAM y ROM.

Para que seamos capaces de poder direccionar 512k + 256k = 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.

<u>Para la memoria RAM.</u>

Disponemos de los siguientes módulos de memoria.

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

512k de RAM es una potencia de dos, así que:

512k = 4 pastillas de los tipos a) para obtener el mapa de memoria.
128k

512k=2 pastillas del tipo b)
256k

Pero además de direccionar 512k, 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 un byte con elementos del tipo b) únicamente necesitaremos 2 modulos.

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.

· 4 módulos de 128k (para poder direccionar 512k) x 16 (módulos necesarios para almacenar un byte), en total 4 x 16 = 64 módulos del tipo a)

Módulos del tipo 128k x 8.

· 2 módulos de 256k (para poder direccionar 512k) x 2 (módulos necesario para almacenar dos bytes), en total 2 x 2 = 4 módulos del tipo b)

Para la memoria ROM.

Disponemos de los siguientes módulos de memoria.

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

256k = 2 pastillas de los tipos a) y c) para obtener el mapa de memoria.
128k

256k= 4 pastillas del tipo b)
64k

Pero además de direccionar 256k, 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 un byte con elementos del tipo b) o c) únicamente necesitaremos 2 modulos.

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.

· 2 módulos de 128k (para poder direccionar 256k) x 16 (módulos necesarios para almacenar un byte), en total 2 x 16 = 32 módulos del tipo a)

Módulos del tipo 64k x 8.

· 4 módulos de 64k (para poder direccionar 256k) x 2 (módulos necesario para almacenar dos bytes), en total 4 x 2 = 8 módulos del tipo b)

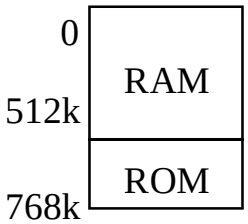
Módulos del tipo 128k x 8.

· 2 módulos de 128k (para poder direccionar 256k) x 2 (módulos necesario para almacenar dos bytes), en total 2 x 2 = 4 módulos del tipo c)

Entonces necesitaremos 4 módulos de 256k x 8 para la memoria RAM y otros 4 módulos de 128k x 8 para la memoria ROM.

3. Diseñar el mapa de memoria.

Para diseñar el mapa de memoria, aunque no nos indican nada, haremos que la RAM esté antes que la ROM, 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 1	0 1	.....	0 1	0 1	Mínima dirección. Máxima dirección.	1ª fila de pastillas: RAM (0k-256k)
		0 1	0 1	.....	0 1	0 1	Mínima dirección. Máxima dirección.	2ª fila de pastillas: RAM (256k-512k)
			0 1	.....	0 1	0 1	Mínima dirección. Máxima dirección.	3ª fila de pastillas: ROM (512k-640k)
			0 1	.....	0 1	0 1	Mínima dirección. Máxima dirección.	4ª fila de patillas: ROM (640k-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 ($128k = 2^{17}$)

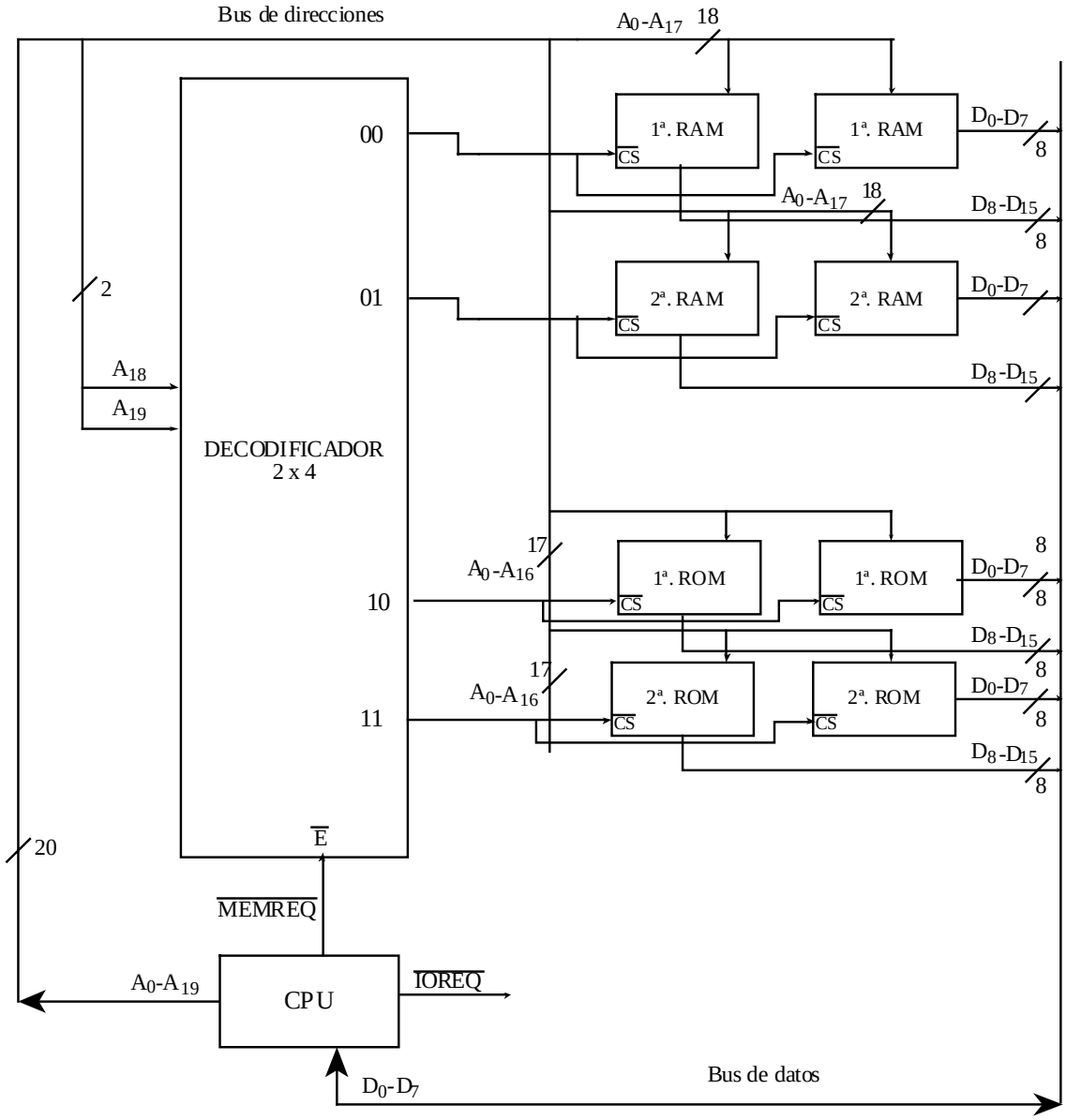
Si miramos la tabla del mapa de memoria que estamos creando, observamos que solamente nos quedan las líneas A_{18} y A_{19} del bus de direcciones para poder distinguir entre las 4 filas de pastillas y que con 2 bits podemos distinguir $2^2 = 4$ elementos. Si nos fijamos en la parte de la tabla remarcada en gris, existen algunas combinaciones de la línea A_{17} que no se emplean. Por lo tanto, la tabla nos queda de la forma:

A_{19}	A_{18}	A_{17}	A_{16}	.....	A_1	A_0		
0	0	0 1	0 1	.....	0 1	0 1	Mínima dirección. Máxima dirección.	1ª fila de pastillas: RAM (0k-256k)
0	1	0 1	0 1	.....	0 1	0 1	Mínima dirección. Máxima dirección.	2ª fila de pastillas: RAM (256k-512k)
1	0	X	0 1	.....	0 1	0 1	Mínima dirección. Máxima dirección.	3ª fila de pastillas: ROM (512k-640k)
1	1	X	0 1	.....	0 1	0 1	Mínima dirección. Máxima dirección.	4ª fila de patillas: ROM (640k-768k)

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. De todas formas, esta es una soñución posible. Otra nos llevaría a considerar las líneas A_{17}, A_{18} y A_{19}.

Finalmente, y en cuanto al ancho de la palabra de la memoria, como queremos acceder a datos de 16 bits, tendremos que tener dos pastillas por fila, tal y como vimos en el paso anterior. Así, por ejemplo, la primera pastilla de cada fila será la *"parte alta"* del dato y la segunda la *"parte baja"*.

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.