

Todas las preguntas tienen la misma puntuación, salvo indicación expresa de lo contrario. Las respuestas erróneas no restan puntos.

- ❑ Imagina que la CPU elemental tiene un interfaz de red que le permitiera comunicar datos con otras CPUs elementales. Imagina también que en su viaje a través de la red, los datos pueden resultar modificados aleatoriamente, de modo que cambien algunos de sus bits y así lo que llega no es igual a lo que había salido. Es conveniente introducir datos adicionales que permitan detectar cuándo se ha producido uno de estos errores.

Un método muy sencillo de detección de errores consiste en contar cuántos bits iguales a 1 tiene el dato que se pretende transmitir, y añadirle un bit extra que se pondrá a 0 ó a 1, de modo que el número total de “unos” en el dato sea par. Al llegar a su destino de nuevo se cuentan cuántos “unos” tiene el dato recibido. Si no es par, es seguro que el dato es erróneo (si es par pudiera darse el caso de que también sea erróneo porque dos de sus bits hayan cambiado, pero este caso ya no se podría detectar).

Queremos transmitir una lista de números naturales, todos ellos menores de  $2^{15}$ , de modo que el bit más significativo no se está usando para representar el dato y por tanto es siempre cero. Usaremos este bit libre para contener el bit de paridad.

A continuación se muestra el listado del programa “emisor”. No se muestra cómo enviaría los datos (ya que la CPU elemental no tiene realmente un interfaz de red), sino sólo como los prepara para su envío. Esto es, cómo añade el bit de paridad (en el bit de signo) a cada uno de los datos. Para ello hace uso de dos funciones:

- **contar\_bits** Esta función recibe un dato (por valor) y devuelve en R0 cuántos de sus bits valen 1. Por ejemplo, si el dato es 00F1, la respuesta sería 5. El “truco” del que se vale la función para contar los bits igual a uno se basa en la idea de que si multiplicamos un dato binario por 2, el resultado que se obtiene en binario es igual al número original, sólo que todos sus bits desplazados una posición a la izquierda. El que era el bit más alto del dato original, “se sale” del resultado y podemos detectar así si era cero o uno. Repitiendo esto 16 veces acabamos por “sacar por la izquierda” todos los bits del dato, de uno en uno, y así podemos contarlos.
- **anyadir\_paridad** Esta función recibe la dirección de un dato (referencia), le añade el bit de paridad necesario (pone un 0 ó un 1 en su bit de signo de modo que el número total de unos sea par), y lo escribe de nuevo en la dirección del dato original. Para determinar si el bit que debe añadir

es 0 ó 1, hace uso de la función `contar_bits` para saber cuántos unos tiene el dato. Si el resultado es impar, debe añadir un 1, en otro caso no hace nada.

El programa principal se limita a llamar repetidamente a la función `anyadir_paridad` pasándole las direcciones de cada uno de los datos de la lista. El código es el siguiente:

```

1  ORIGIN 0700h
2  INICIO ini
3  .PILA 22h
4  .DATOS
5  lista VALOR 22h, 7FFFh, 6532h, 7, 6999h, 256, 1
6
7  .CODIGO
8  PROCEDIMIENTO contar_bits
9      PUSH R6
10     MOV R6, R7
11     PUSH R1
12     PUSH R2
13     INC R6
14     INC R6
15     MOV R1, [R6] ; Leer parámetro (dato)
16     XOR R0, R0, R0 ; Inicializar a cero el contador
17     MOVH R2, 0
18     MOVL R2, 16 ; Contador de repeticiones
19     repetir:
20         ADD [R6], [R6]
21         ; salta si bit más alto de R1 era cero
22         INC R0 ; si era 1, incrementar contador de unos
23     cero:
24         DEC R2
25         BRNZ repetir
26         POP R2
27         POP R1
28         POP R6
29         RET
30     FINP
31
32     PROCEDIMIENTO anyadir_paridad
33         PUSH R6
34         MOV R6, R7
35         PUSH R1
36         PUSH R2
37         PUSH R3
38         INC R6
39         INC R6
40         MOV R1, [R6] ; Leer parámetro
41         ; Leer dato
42         PUSH R3 ; Averiguar cuántos unos tiene el dato
43         CALL contar_bits
44         INC R7
45         ; Tenemos en R0 el número de bits del dato
46         ; Su último bit nos dice si es par (0) o impar (1)
47         ; Para ello preparo una máscara
48         MOVH R2, [R6]
49         MOVL R2, [R6]
50         ; Y la aplico
51         [R6], R2, R0
52         BR [R6], es_par

```

```

53     ; Si no es par, es impar
54     ; hay que añadirle el bit de paridad al dato
55     MOVH R2, [R6]
56     MOVL R2, [R6]
57     [R6], R2
58     ; Volvemos a meterlo en su sitio
59     MOV [R1], R3
60     es_par:
61     ; Si era par, no es necesario añadir nada
62     ; Terminamos
63     POP R3
64     POP R2
65     POP R1
66     POP R6
67     RET
68     FINP
69
70     ; El programa principal recorre la lista
71     ini:
72         MOVH R2, BYTEALTO DIRECCION lista
73         MOVL R2, BYTEBAJO DIRECCION lista
74         MOVH R1, 0
75         MOVL R1, 7 ; Numero de elementos a procesar
76     bucle:
77         PUSH R2
78         CALL anyadir_paridad
79         INC [R6]
80         INC [R6]
81         DEC [R6]
82         BRNZ bucle
83     final:
84         JMP final ; La ejecución quedará "parada" aquí
85     FIN

```

— Completa la línea 20.

— ¿Qué falta en la línea 21?

— ¿Qué falta en la línea 41?

— En las líneas 48 a 52 hay trozos de código tapados. Reescribe estas líneas incluyendo lo que falta (el tamaño de los huecos no es una pista sobre su contenido)

- En las líneas 55 a 57 parte del código está oculto. Reescribe a continuación estas líneas completando lo que falta (el tamaño de los huecos no es una pista sobre sus contenidos).

- Completa los huecos a partir de la línea 79.

- Cambiamos el valor 0700h de la directiva ORIGIN por otro valor  $N$ , y el valor 22h de la directiva .PILA, por otro valor  $P$ . En la zona de datos se definen  $D$  datos. Tras estos cambios, compilamos de nuevo el listado anterior y obtenemos un nuevo ejecutable. Las tres primeras líneas del .eje contienen tres números, que denominaremos de forma genérica  $N$ ,  $M$  y  $Q$  (observar que el primero de ellos es el valor utilizado en la directiva ORIGIN).

Escribe una fórmula en la que participen las variables  $N$ ,  $M$ ,  $Q$ ,  $P$  y/o  $D$  (no necesariamente todas ellas) que permita obtener el tamaño ocupado por la zona de código del programa.

- Escribe los dos primeros datos que quedarán almacenados a partir de la dirección 0700h cuando el programa haya finalizado. Da la respuesta en hexadecimal.

- ¿Cuál es el código máquina de la instrucción JMP final en el listado anterior? Responder en hexadecimal.

- Asumiendo que no se producen interrupciones durante la ejecución de este programa, ¿cuánto espacio ha quedado sin usar en la pila? Responde en decimal.

- Las cuatro primeras líneas del fichero .eje contienen los números 0700, 0733, y 0760. ¿Qué valor está almacenado en la dirección de memoria 075F cuando el programa termine? Responder en hexadecimal.

Seguidamente abordamos el problema de procesar los datos una vez recibidos. El procesamiento consiste en comprobar la paridad de cada dato, si es impar marcar el dato como erróneo; si es par asumir que el dato es correcto y en este caso limitarse a borrar su bit de signo (donde estaba almacenado el bit de paridad) para recuperar el dato original.

A continuación se muestra el listado de un programa que hiciera este tipo de transformación sobre los datos recibidos. De nuevo no se muestra el código que haría la recepción de datos, sino que se supone que esos datos ya han llegado y son los que se muestran tras la etiqueta lista (no son los mismos datos que transmitió el programa “emisor” antes mostrado, se trataría de otro ejemplo con otros datos).

Para llevar a cabo su cometido, implementa dos funciones:

- `contar_bits` Esta función es exactamente la misma que ya explicamos para el “emisor”. Simplemente recuenta cuántos bits tiene a 1 el dato que recibe por la pila, y devuelve el resultado en R0.
- `comprobar_paridad` Esta función recibe la dirección de un dato (referencia), averigua cuántos bits tiene a 1 ese dato (llamando a `contar_bits`) y se encuentra en uno de estos dos casos:
  - El resultado es impar. Esto indica un error en la recepción del dato. Sustituye dicho dato por FFFFh como indicador del error.
  - El resultado es par. Asumimos que en este caso no hubo error. El dato se sustituye por el resultado de borrar su bit de paridad.

El programa principal se limita a llamar repetidamente a la función `comprobar_paridad` pasándole las direcciones de cada uno de los datos de la lista. El código es el siguiente (no está completo pues hay partes iguales a las del “emisor”):

```

1  ORIGIN 0900h
2  INICIO ini2
3  .PILA 22h
4  .DATOS
5  lista VALOR 0F003h, 0444h, 7654h, 0B0CAh,
6           8001h, 2222h, 9441h
7  .CODIGO
8  PROCEDIMIENTO contar_bits
9  ; No se muestra. Es idéntico al del ``emisor``
10 FINP
11
12 PROCEDIMIENTO comprobar_paridad
13     PUSH R6
14     MOV R6, R7
15     PUSH R1
16     PUSH R2
17     PUSH R3
18     INC R6
19     INC R6
20     MOV R3, [R6] ; Leer parámetro (dirección del dato)
21     MOV R1, [R3] ; Leer dato
22     PUSH R1 ; Averiguar cuántos bits tiene a 1
23     CALL contar_bits
24     INC R7
25     ; El test para ver si R0 es par o impar es igual al
26     ; mostrado en la línea 48 y siguientes
27     ; del "emisor". Por eso se oculta aquí también
28     MOVH R2, [R3]
29     MOVL R2, [R3]
30     ROR R0, R0, R2
31     BR [R3] es_par
32     ; si no es par, es impar --> dato erróneo
33     ; que se cambia por FFFFh
34     [R3] = 0xFFFFh
35     JMP retornar
36
37 es_par:
38     ; Si es par el dato es correcto. Borramos su
39     ; bit de paridad para recuperar el dato original
40     ; Basta aplicar la máscara apropiada que
41     ; deje pasar todos los bits, salvo el más alto
42     ; que lo pone a cero
43     MOVH R2, [R3] ; Preparo la máscara en R2
44     MOVL R2, [R3]
45     [R3] = [R3] AND R2 ; se la aplico al dato
46     MOV [R3], R1 ; y guardo el resultado
47     ; Ya podemos retornar
48     RET
49 retornar:
50     POP R3
51     POP R2
52     POP R1
53     POP R6
54     RET
55 FINP
56
57 ini2:
58 ; El programa principal recorre los datos de la lista
59 ; No se muestra, pues es análogo al del "emisor",
60 ; pero llamando a comprobar_paridad,
61 ; en lugar de a anyadir_paridad
62 FIN

```

- En las líneas 44 a 46 hay trozos de código tapados. Reescribe estas líneas incluyendo lo que falta (el tamaño de los huecos no es una pista sobre su contenido).

- ¿Qué falta en los huecos a partir de la línea 34?

- Cuando el programa haya finalizado ¿cuántos de sus datos habrán sido sustituidos por FFFFh?

- ¿Qué habrá en la dirección de memoria 0904h cuando el programa haya finalizado?

- ☐ El siguiente código carga un dato desde la memoria al registro R0 y pretende saltar a la etiqueta `es_cero` si el dato leído es cero.

```
MOV R0, [R1]
BRZ es_cero
```

¿Es necesario colocar alguna instrucción en el hueco? Si crees que no, responde “No es necesario”. Si crees que sí, responde una instrucción (sólo una) que sea adecuada al caso.

- ☐ Considera el listado siguiente, cuyo objetivo es sumar una serie de 50 números almacenados a partir de la dirección de memoria 0500h, dejando el resultado en R0:

```
1  MOVL R1, 00
2  MOVH R1, 05
3  MOVL R2, 50
4  MOVH R2, 0
5  XOR R0, R0, R0
6  bucle:
7  MOV R3, [R1]
8  ADD R0, R0, R3
9  INC R1
10 DEC R2
11 BRNZ bucle
```

- Si el reloj de la CPU que ejecuta este programa tiene una frecuencia de 2MHz ¿cuánto tiempo tardaría en finalizar el bucle? (sin contar las instrucciones de inicialización previas al mismo) Indicar las unidades en la respuesta.

- Imagina que la CPU contara con una instrucción llamada `ADD Rd, Rs, [Ri]`, que permitiera sumar un registro `Rs` con el contenido de una posición de memoria indicada en otro registro `Ri`, dejando el resultado en un tercer registro `Rd`. ¿Cómo usarías esta instrucción para hacer más eficiente el programa anterior? Escribe a continuación cómo sería el nuevo bucle.

- ¿Cómo podría implementarse dicha instrucción? (`ADD Rd, Rs, [Ri]`) Escribe en la tabla siguiente las señales necesarias en cada ciclo. Nota: quizás la tabla contiene más filas de las necesarias.

Paso	Señales de control
4	
5	
6	
...	

- ☐ La CPU acaba de finalizar el paso 3 de una instrucción y se dispone a iniciar el paso 4. En ese instante los valores de algunos de los registros de la CPU y de algunas de las direcciones de la memoria son los que se muestran en las tablas siguientes:

	DIR.	CONT.
	10FE	7124
	10FF	AB38
R7=1100h	1100	3951
MAR=20F0h	1101	0000
PC=20F1h	1102	5720
MDR=B800h		

- ¿Cuál será el valor que tendrá PC durante el paso 1 de la siguiente instrucción que se ejecute? Responder en hexadecimal.

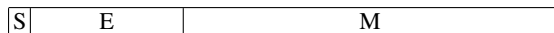
- Si la próxima instrucción que se ejecuta es un salto condicional `BRZ`, ¿cuántos pasos de ejecución tomará la ejecución de este salto?

- ☐ Se tiene una serie de códigos binarios, que son respectivamente las codificaciones de:

- A) 130 (natural)
- B) -15 (complemento a 2)
- C) -15 (signo-magnitud)
- D) -15 (exceso a 127)

Si un algoritmo intenta ordenar de menor a mayor estos códigos sin saber qué significan, simplemente comparándolos como si fueran naturales, ¿qué ordenación produciría? Ejemplo de respuesta: B) A) C) D)

- ☐ El formato IEEE-754 de precisión simple usa 8 bits para codificar el campo exponente (en exceso a 127) y 24 para la mantisa (uno para signo y 23 para la magnitud). Si el campo exponente es “todo ceros” el número es desnormalizado, si es “todo unos” no se trata de un número, sino de un “concepto” como infinito. Los campos se empaquetan en 32 bits con el siguiente esquema:



- ¿Cuál es la codificación en la norma IEEE-754 de precisión simple de la cantidad  $2^{-126}$ ? Escribir la respuesta en hexadecimal.

- ¿Y la de  $2^{-127}$ ?

- ¿Cuántos números desnormalizados *diferentes* se pueden representar en este formato?