Examen final IC (*Parte 2)*

- Duración del examen: 2:00 horas.
- Los problemas tienen que resolverse en las **HOJAS DE RESPUESTAS**.
- No podéis utilizar calculadora, móvil, apuntes, etc.
- La solución del examen se publicará en Atenea mañana por la mañana.

Nota: La indicación de la puntuación de los ejercicios es sobre 10 puntos, pero esta parte del examen final solo representa 4 puntos de la nota del examen final.

Ejercicio 1 (2.1 puntos)

El programa en ensamblador SISA que se muestra a continuación se ha traducido a lenguaje máquina situando la sección de datos (.data) a partir de la dirección `0xB000` y justo a continuación la sección de código (.text).

```
        .data
                v1: .word 16, 3, 15, 5, 8, 0, 8195, 8, 0  ; Números naturales
                v2: .space 8*2
        .text
                movi R0, lo(v1)
                movhi R0, hi(v1)
                movi R1, lo(v2)
                movhi R1, hi(v2)

        bucle:  ld R2, 0(R0)
                bz R2, end
                ld R3, 2(R0)
                bnz R3, a
                movi R4, -1
                movi R2, -1
                bz R3, c
        a:      movi R4,0
        b:      cmpltu R7, R2, R3
                bnz R7, c
                sub R2, R2, R3
                addi R4, R4, 1
                bnz  R3, b
        c:      st 0(R1), R4
                st 2(R1), R2
                addi R0,R0,4
                addi R1,R1,4
                bnz R2, bucle
        end:    ;Fin de programa
        .end
```

Una vez ensamblado y cargado el programa en la memoria y antes de comenzar su ejecución:

a) ¿A qué direcciones de memoria corresponden las etiquetas o direcciones simbólicas siguientes? *(0.5 puntos)*

`v2=0x` `bucle=0x`

b) ¿Cuántos ciclos tardará la ejecución completa del programa? (desde el inicio hasta que el registro PC tenga el valor de la etiqueta end). *(1 punto)*

`Num. ciclos=`

c) ¿Cuál será el contenido del vector `v2` cuando el programa finalice su ejecución? Indicad los valores separándolos por comas, interpretándolos como números naturales y de tamaño 16 bits. *(0.6 puntos)*

`v2=`

Ejercicio 2 (1.5 puntos)

Cada uno de los apartados pregunta sobre un ciclo concreto de la ejecución de varias instrucciones en el SISC Von Neumann. Escribid el contenido del registro IR, el contenido de la ROM_OUT y el valor de los bits de la **palabra de control** que genera el bloque **SISC CONTROL UNIT** durante el ciclo a que hace referencia cada apartado**.** Para cada apartado/fila se indica el nodo/estado de la UC en ese ciclo y la instrucción (en ensamblador) que está almacenada en el IR en ese ciclo. Podéis ver el grafo de estados de Moore de la UC en el anexo. Suponed que el contenido de todos los registros, `Rk` para k=0,...7, antes de ejecutarse cada instrucción es -1.

a) Indica el contenido del registro IR. Utilizad el valor 0 para los bits que sean x (solo para este apartado). *(0.3 puntos)*

Nodo / Estado (Mnemo Salida)	**Instrucción en IR** (en ensamblador)	**Valor del IR** (en hexadecimal)
D	`JALR R1,R2`	`0x`
Stb	`STB 2(R1),R2`	`0x`
Bz	`BZ R2,4`	`0x`

b) Indica el contenido de la ROM_OUT en hexadecimal usando las conexiones en el orden que están en el anexo. Utilizad el valor 0 para los bits que sean x (solo para este apartado). *(0.6 puntos)*

Nodo / Estado (Mnemo Salida)	**Instrucción en IR** (en ensamblador)	**Contenido ROM_OUT** (en hexadecimal)
D	`JALR R1,R2`	0x
Stb	`STB 2(R1),R2`	0x
Bz	`BZ R2,4`	0x

c) Indica el valor de los bits de la **palabra de control** que genera el bloque **SISC CONTROL UNIT** durante el ciclo a que hace referencia cada apartado**. Poned x siempre que no se pueda saber el valor de un bit** (ya que no podemos suponer cómo se han implementado las x en la ROM_OUT). *(0.6 puntos)*

Apartado	**Nodo / Estado** (Mnemo Salida)	**Instrucción en IR** (en ensamblador)	**Palabra de Control**																		
			@A	@B	Pc/Rx	Ry/N	OP	F	P/I/L/A	@D	WrD	Wr-Out	Rd-In	Wr-Mem	Ldlr	LdPc	Byte	Alu/R@	R@/Pc	N (hexa)	ADDR-IO (hexa)
a	D	`JALR R1,R2`																			
b	Stb	`STB 2(R1),R2`																			
c	Bz	`BZ R2,4`																			

Ejercicio 3 **(1.4 puntos)**

Uno de los primeros algoritmos que se aprenden en programación son los algoritmos de ordenación de vectores de elementos. Se desea implementar un algoritmo de ordenación en lenguaje SISA para ser ejecutado que el computador **SISC Von Neumann**. Se ha escogido implementar el algoritmo de ordenación de burbuja (*Bubble Sort*) por su simplicidad. *Bubble Sort* es el algoritmo de clasificación más simple (y uno de los más ineficientes) que funciona intercambiando repetidamente los elementos adyacentes si están en el orden incorrecto. Funciona recorriendo todo el vector a ordenar, revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden incorrecto. Al terminar de recorrer los elementos, se determina si hubo algún cambio, y de haberlo, se repite el método hasta que ya no haya cambio alguno.

Por ejemplo, suponed que el vector a ordenar es el (5 1 4 2 8), entonces la secuencia de pasos seria la siguiente:

Primer pase:
(**5 1** 4 2 8) → (**1 5** 4 2 8), Aquí, el algoritmo compara los dos primeros elementos y los intercambia puesto que 5 > 1.
(1 **5 4** 2 8) → (1 **4 5** 2 8), Intercambiar puesto que 5 > 4
(1 4 **5 2** 8) → (1 4 **2 5** 8), Intercambiar puesto que 5 > 2
(1 4 2 **5 8**) → (1 4 2 **5 8**), Ahora, dado que estos elementos ya están en orden (8 > 5), el algoritmo no los intercambia.

Segundo pase:
(**1 4** 2 5 8) → (**1 4** 2 5 8)
(1 **4 2** 5 8) → (1 **2 4** 5 8), Intercambiar puesto que 4 > 2
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

Ahora, la matriz ya está ordenada, pero nuestro algoritmo no sabe si está completa. El algoritmo necesita una pasada completa sin ningún intercambio para saber que está ordenado.

Tercer pase:
(**1 2** 4 5 8) → (**1 2** 4 5 8)
(1 **2 4** 5 8) → (1 **2 4** 5 8)
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

Se ha decidido implementar el algoritmo *Bubble Sort* usando un valor booleano que indica si ha habido una permutación o no durante el pase. En el siguiente fragmento en seudocódigo podéis ver el algoritmo. El algoritmo se podría implementar ligeramente más eficiente porque a cada pase queda un elemento menos por ordenar. Pero hemos decidido no hacerlo para simplificar el algoritmo.

```
void bubble_sort (vector[0:n-1]) {
  do {
      permutacion=FALSO
      for (i=0; i<=n-2; i++)
          if (vector[i] > vector[i+1]) {
              permutacion=VERDADERO
              /* intercanviar vector[i] y vector[i+1] */
              aux=vector[i];
```

```
                vector[i]=vector[i+1];
                vector[i+1]=aux;
            }
        }
    while {permutacion==VERDADERO}
}
```

Se desea implementar en código SISA el algoritmo de ordenación de la burbuja mostrado anteriormente para que ordene un vector alojado en la zona de memoria para datos. Se garantiza que la longitud del vector a ordenar nunca tendrá más de 30 elementos. Completa el siguiente código con las partes subrayadas que faltan.

```
LONG=12
.data
     vector: .word 2,6,24,12,26,92,18,74,105,36,52,64
.text
      _____ R0, lo(vector)
      _____ R0, hi(vector)
do:   MOVI  R1,0              ;permutación
      MOVI  R2,0              ;i
      MOVI  R3, _____
      _____ R3, R3, -2
for:  _____ R7, R2, R3
      B____ R7, ___
if:   A____ R4, ___, ___
      A____ R4, ___, ___
      _____ _____, _____
      _____ _____, _____
      CMPLEU R7, R5, R6
      B____ R7, _____
      MOVI  R1, _____
      _____ _____, _____
      _____ _____, _____
fif:  ADDI  R2, R2, _____
      BNZ   R2, _____
ffor: BNZ   R1, _____
.end
```

Ejercicio 4 **(1.5 puntos)**

Especificad el **camino crítico** (indicando la suma ordenada de los tiempos de propagación de los bloques por los que pasa) y calculad el **tiempo de ciclo mínimo** para que el computador **SISC Von Neumann** pueda ejecutar correctamente el tipo de instrucción SISA que se indica en cada apartado (este sería el tiempo de ciclo mínimo del computador si solo ejecutara instrucciones como la indicada u otras que requieran menor tiempo). No tenéis que añadir ningún porcentaje de seguridad en el cálculo del tiempo de ciclo mínimo. Suponed que los tiempos de propagación de los bloques que forman el computador son los siguientes:

T_p(ROM_Q+) = 80 u.t.

T_p(ROM_OUT) = 120 u.t.

T_p(MUX-2-1) = 50 u.t.

T_p(MUX-4-1) = 100 u.t.

T_p(DEC-3-8) = 150 u.t.

T_p(REG) = 110 u.t. // Tiempo de propagación de un registro.

T_p(REGFILE) = 240 u.t. // Tiempo de lectura del banco de registros

T_p(ALU-slow) = 650 u.t. // Tp de la ALU para las operaciones/funciones lentas: `ADD`, `SUB`, `CMP*`.

T_p(ALU-quick) = 280 u.t. // Tp de la ALU para las operaciones/funciones rápidas: cualquier otra distinta de `ADD`, `SUB`, `CMP*`.

T_{acc}(MEMORY) = 800 u.t. // Tiempo de acceso (para la lectura o escritura) a la memoria

T_p(AND-2) = T_p(OR-2) = 20 u.t.

T_p(NOT) = 10 u.t.

El tiempo de propagación de un bloque combinacional (T_p) y el tiempo de acceso a memoria para realizar una lectura (T_{acc}) es el tiempo desde que están estables todas las entradas necesarias hasta que se estabilizan las salidas requeridas al valor correcto para las entradas aplicadas. Desconocemos como se han implementado internamente los bloques (y podría ser de forma diferente a los vistos en clase). Recordad que un registro con señal de carga (Ld), REGwLd, está construido con un REG y un MUX-2-1 (no os damos el esquema interno del REGwLd, porque lo tenéis que saber).

a) T_c correspondiente al nodo de **D** (decode).

b) T_c correspondiente al nodo de **Movhi**.

c) T_c correspondiente al nodo de **Stb**.

Ejercicio 5 (3.5 puntos)

Debido a que es muy frecuente en programas tener que inicializar un vector con un valor concreto, se ha decidido crear unas nuevas instrucciones que hagan esto. De modo que estas nuevas instrucciones se encarguen de inicializar un vector en memoria con el valor deseado.

Completad el diseño del SISC Von Neumann para que pueda ejecutar, además de las 25 instrucciones originales SISA, dos nuevas instrucciones `INIVEC` y `INIVECB` (una para vectores de words y otra para vectores de bytes) que permitan inicializar un vector en memoria. Y que tienen el formato y codificación, la sintaxis ensamblador y la semántica siguientes:

Codificación: `1011 aaa bbb ccc xxx`
Sintaxis: `INIVEC Ra, Rb, Rc`
Semántica: `PC=PC+2; MEM`$_w$`[Ra&(~1)]=Rc; Ra=Ra+2; Rb=Rb-1; if (Rb!=0) {PC=PC-2};`

Codificación: `1100 aaa bbb ccc xxx`
Sintaxis: `INIVECB Ra, Rb, Rc`
Semántica: `PC=PC+2; MEM`$_b$`[Ra]=Rc<7..0>; Ra=Ra+1; Rb=Rb-1; if (Rb!=0) {PC=PC-2};`

Donde `Ra` es la dirección de memoria donde está el primer elemento del vector, `Rb` es el número de elementos del vector y `Rc` el valor con el que se desea inicializar. Por ejemplo, supongamos que queremos inicializar un vector de 25 words, que está almacenado en la posición de memoria 0x1260, con el valor 7. En los siguientes códigos podéis observar como se escribiría el código en el SISA actual y como con las nuevas instrucciones.

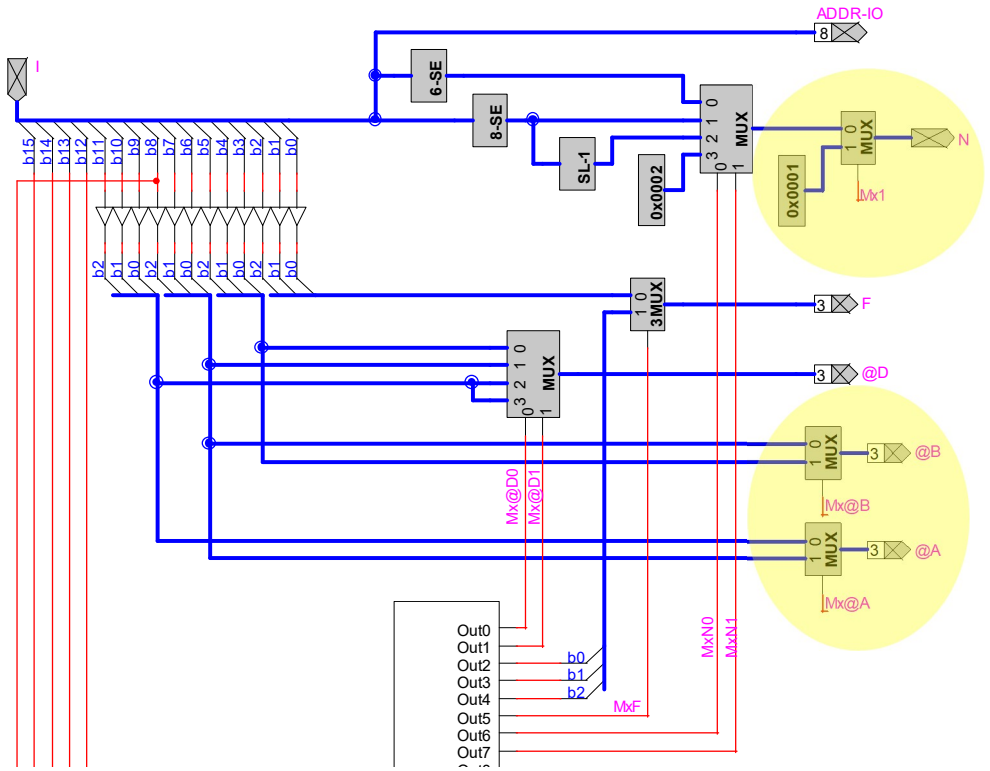
Código original

```
        MOVI  R0, lo(0x1260)
        MOVHI R0, hi(0x1260)
        MOVI  R1, 25
        MOVI  R2, 7
bucle:  ST    0(R0), R2
        ADDI  R0, R0, 2
        ADDI  R1, R1, -1
        BNZ   R1, bucle
```

Código con la nueva instrucción

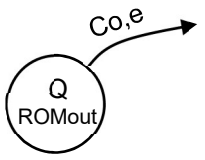
```
MOVI   R0, lo(0x1260)
MOVHI  R0, hi(0x1260)
MOVI   R1, 25
MOVI   R2, 7
INIVEC R0, R1, R2
```

Para añadir estas nuevas instrucciones no es necesario modificar el hardware de la UPG. Para ejecutar estas nuevas instrucciones solamente se ha modificado el hardware de la unidad de control del computador (UCG) añadiendo tres nuevos MUX-2-1 con señales de selección Mx@A, Mx@B y Mx1. La siguiente figura muestra la parte de la Unidad de Control que ha cambiado respecto de la original. Las tres nuevas señales de selección se conectan a la salida de la ROM_OUT "por nombre" para simplificar el dibujo del esquema lógico. Además de estos nuevos multiplexores, ha cambiado la ROM_OUT que ahora tiene 27 bits por palabra (los 24 originales más los 3 nuevos).



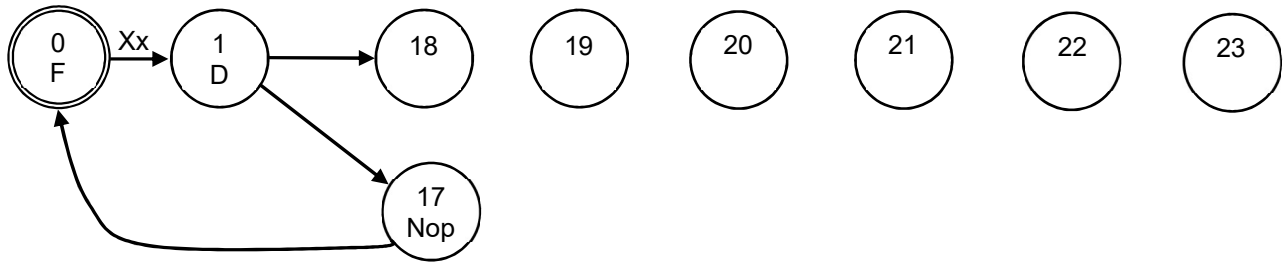
a) Completad el fragmento del grafo de estados de la figura correspondiente al circuito secuencial de la unidad de control **únicamente para la instrucción `INIVEC`**. Se da la leyenda del grafo y los nodos necesarios para ejecutar las nuevas instrucciones, pero faltan arcos y etiquetas. Dibujad todos los arcos que faltan y todas las etiquetas. En número de nodos necesarios para ejecutar las nuevas instrucciones dependerá de la solución que propongáis, pero no deberían ser más de 6 y deberá ser coherente con las acciones que pongáis en el apartado b). Los nodos que os sobren podéis dejarlos sin conectar. *(0.5 puntos)*

Leyenda:



CO: Código de operación de la Instrucción, I_{15} I_{14} I_{13} I_{12} (en hexadecimal)
e: Bit de extensión del código de operación (I_8)
Q: Estado (en decimal)
ROMout: Mnemotécnico de salida

Grafo:



b) Completad el contenido de la siguiente tabla que indica, mediante una fila para cada nodo del grafo de estados de la unidad de control, la acción (o acciones en paralelo) que se realiza en el computador en cada uno de los ciclos/nodos que requiere la ejecución **únicamente de la nueva instrucción `INIVEC`** (Fetch, Decode, y los ciclos/nodos de la ejecución propiamente dicha). Para especificar las acciones usad el mismo lenguaje de transferencia de registros que en la documentación. En número de filas/nodos de la tabla a rellenar dependerá de la solución que propongáis y deberá ser coherente con el apartado a). Nota: Existe una solución con 6 nodos (F, D y 4 nodos nuevos). *(1 punto)*

Nodo	Mnemotécnico	Acciones
E0	F	IR ← Mem_w[PC] // PC ← PC+2
E1	D	RX ← Ra // RY ← Rb // R@ ← PC+SE(N8)*2
E18		
E19		
E20		
E21		
E22		
E23		

c) Completad (poniendo 0, 1 o x en cada bit) las filas de la tabla que especifican el contenido de la ROM_OUT para las direcciones de los nodos que hayáis usado para implementar la nueva instrucción (de la 18 a la 23). **Poned x siempre que el valor de un bit no importe**. *(1 punto)*

@ROM	**Mx1**	**Mx@A**	**Mx@B**	Bnz	Bz	WrMem	RdIn	WrOut	WrD	LdIr	Byte	R@/Pc	Alu/R@	Pc/Rx	Ry/N	P/I/L/A_1 P/I/L/A_0	OP_1 OP_0	MxN_1 MxN_0	MxF	F_2 F_1 F_0	Mx@D_1 Mx@D_0	***Nodo***
18																						
19																						
20																						
21																						
22																						
23																						

d) Suponiendo que la ejecución de la nueva instrucción se realiza en 6 ciclos (F, D y 4 nodos nuevos). Indica cuantas instrucciones se ejecutarían y cuantos ciclos se tardaría en ejecutar el programa de ejemplo del inicio del enunciado si ejecutásemos el código original y si ejecutásemos el programa con la nueva instrucción. *(0.5 puntos)*

Número instrucciones código original: ____ Número instrucciones con la nueva instrucción: ____

Ciclos de ejecución código original: ____ Ciclos de ejecución con la nueva instrucción: ____

e) Deseamos añadir la instrucción **`INIVECB`** suponiendo que ya tenemos implementada la instrucción **`INIVEC`**. Respecto a **`INIVEC`** simplemente se diferencia ligeramente en dos estados. Así que solo se necesitarían dos estados adicionales para implementarla. Indicad en la siguiente tabla, mediante una fila para cada nodo del grafo de estados de la unidad de control, la acción (o acciones en paralelo) que se realizan en el computador en cada uno de los ciclos/nodos que requiere la ejecución de estos dos estados que son diferentes a los de la instrucción **`INIVEC`**. Para especificar las acciones usad el mismo lenguaje de transferencia de registros que en la documentación. *(0.5 puntos)*

Nodo	Mnemotécnico	Acciones
Exx	IniVecB1	
Exx	IniVecB2	