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 `0x6000` y la sección de código (.text) a partir de la dirección `0xB400`.

```
.data
        long=5
        A: .word 0x000F, 0x9A00, 0x0102,0xFFFE, 0x0010
        B: .byte 0X06, 0x88, 0x82, 0xFA, 0x25
        C: .space 100
.text
        MOVI  R3, long
        MOVI  R1, lo(A)
        MOVHI R1, hi(A)
        MOVI  R2, lo(B)
        MOVHI R2, hi(B)
        MOVI  R7, 0
        MOVHI R7, 0x80
loop:   LD    R4, 0(R1)
        LDB   R5, 0(R2)
        SUB   R6, R4, R5
        AND   R0, R6, R7
        BNZ   R0, neg
pos:    ST    22(R1), R6
        BZ    R0, cont
neg:    ST    22(R1), R5
cont:   ADDI  R1, R1, 2
        ADDI  R2, R2, 1
        ADDI  R3, R3, -1
        BNZ   R3, loop
.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? *(0.2 puntos)*

`C=0x` `loop=0x`

b) ¿Cuál es la dirección de memoria y su contenido donde han quedado almacenadas las siguientes instrucciones una vez cargado el programa? *(0.6 puntos)*

Instrucción	@ memoria y contenido
`ST 22(R1), R5`	Mem_w[0x] = 0x
`SUB R6, R4, R5`	Mem_w[0x] = 0x
`BNZ R0, neg`	Mem_w[0x] = 0x

c) Una vez ejecutado el programa en el SISC von Neumann, completa la siguiente tabla con el contenido de la memoria en hexadecimal (no es necesario poner **0x** del valor). Las posiciones de memoria que no se pueda saber su contenido dejadlas en blanco. *(0.4 puntos)*

@memoria	Valor
0x6000	
0x6001	
0x6002	
0x6003	
0x6004	
0x6005	
0x6006	
0x6007	

@memoria	Valor
0x6008	
0x6009	
0x600A	
0x600B	
0x600C	
0x600D	
0x600E	
0x600F	

@memoria	Valor
0x6010	
0x6011	
0x6012	
0x6013	
0x6014	
0x6015	
0x6016	
0x6017	

@memoria	Valor
0x6018	
0x6019	
0x601A	
0x601B	
0x601C	
0x601D	
0x601E	
0x601F	

d) Si se ejecuta el código mostrado, ¿Cuántas instrucciones se ejecutan, cuántas son lentas y cuántas son rápidas en la Harvard multiciclo? Si se ejecuta el código en las tres versiones de los computadores SISC (Harvard uniciclo, el Harvard multiciclo y el Von Neumann) ¿Cuántos ciclos tarda en ejecutarse el código en cada una se los tres computadores? ¿Cuánto tarda en ejecutarse el código en el Harvard uniciclo, en el Harvard multiciclo y en el Von Neumann suponiendo que los tiempos de ciclo son 3.000, 2000 y 1.000 u.t. respectivamente? *(0.9 puntos)*

Nº de instruc. ejecutadas =	Nº instr. lentas (H. multiciclo)=	Nº instr. rápidas (H. multiciclo)=
Nº de ciclos (H. uniciclo)=	Nº de ciclos (H. multiciclo)=	Nº de ciclos (Von Neumann)=
Tejec(Harvard uniciclo) =	Tejec(Harvard multiciclo) =	Tejec(Von Neumann) =

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 0.

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 R6,R4`	`0x`
Ldb	`LDB R2,2(R1)`	`0x`
Bnz	`BNZ 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 R6,R4`	`0x`
Ldb	`LDB R2,2(R1)`	`0x`
Bnz	`BNZ 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 R6,R4`																			
b	Ldb	`LDB R2,2(R1)`																			
c	Bnz	`BNZ 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+) = 90 u.t.

T_p(ROM_OUT) = 110 u.t.

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

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

T_p(DEC-3-8) = 140 u.t.

T_p(REG) = 120 u.t. // Tiempo de propagación de un registro.

T_p(REGFILE) = 250 u.t. // Tiempo de lectura del banco de registros

T_p(ALU-slow) = 700 u.t. // Tp de la ALU para las operaciones/funciones lentas: `ADD`, `SUB`, `CMP*`.

T_p(ALU-quick) = 300 u.t. // Tp de la ALU para las operaciones/funciones rápidas: cualquier otra distinta de `ADD`, `SUB`, `CMP*`.

T_{acc}(MEMORY) = 900 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 **F** (fetch).

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

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

Ejercicio 5 **(3.5 puntos)**

Debido a que es muy frecuente en programas, en el momento de hacer un bucle, decrementar el contenido de un registro (el iterador del bucle) en una unidad, compararlo el contenido con cero y romper el secuenciamiento implícito (saltar). Se ha decidido crear una nueva instrucción que hagan esto (ejemplo). De modo que esta nueva instrucción haga las dos acciones en una única ejecución de una instrucción.

Código original

```
bucle:  instr1
        instr2
        ADDI  R6,R6,-1
        BZ    R6, bucle
        instr3
```

Código con la nueva instrucción

```
bucle:  instr1
        instr2
        DOBZ R6, bucle
        instr3
```

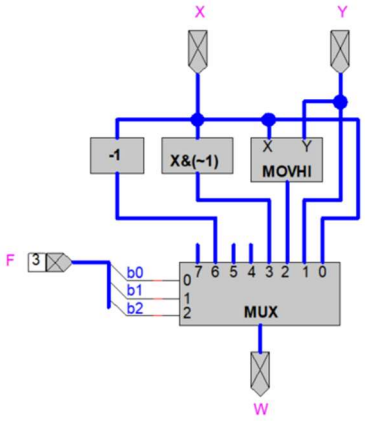
Completad el diseño del SISC Von Neumann para que pueda ejecutar, además de las 25 instrucciones originales SISA, dos nuevas instrucciones (DOBZ y DOBNZ) que permiten decrementar en una unidad el contenido de un registro y romper el secuenciamiento implícito en caso de que el resultado sea igual o distinto a cero. Y que tienen el formato y codificación, la sintaxis ensamblador y la semántica siguientes:

Codificación: `1011 aaa 0 nnnnnnnn`
Sintaxis: `DOBZ Ra, N8`
Semántica: `PC=PC+2; Ra=Ra-1; if (Ra==0) {PC=PC+SE(N8)*2};`

Codificación: `1011 aaa 1 nnnnnnnn`
Sintaxis: `DOBNZ Ra, N8`
Semántica: `PC=PC+2; Ra=Ra-1; if (Ra!=0) {PC=PC+SE(N8)*2};`

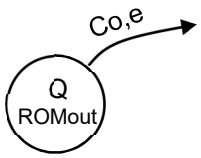
Para añadir estas nuevas instrucciones no es necesario modificar el hardware de la UCG que no sea cambiar el contenido de las ROM's, y solo hay que realizar una simple modificación en el hardware de la UPG. Se ha modificado la ALU para que sea capaz de realizar la operación de decrementar un valor en una unidad. Se añade a la ALU un bloque `DEC(-1)` que decrementa un valor entero en una

unidad. Este bloque pertenecerá al grupo operaciones de miscelánea (`OP=10`) y estará conectado de la siguiente forma. En el siguiente esquema correspondiente al bloque de miscelánea de la ALU podéis ver la modificación que se ha realizado.



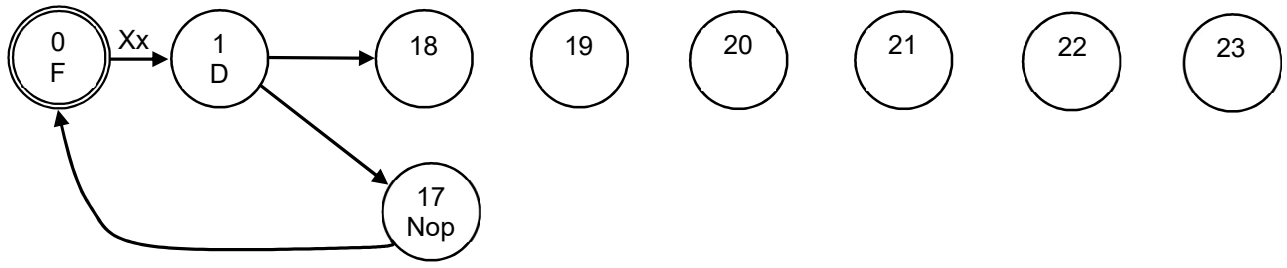
a) Completad el fragmento del grafo de estados de la figura correspondiente al circuito secuencial de la unidad de control. Se da la leyenda del grafo y los nodos necesarios para ejecutar las nuevas instrucciones, pero faltan arcos y etiquetas. Dibujad todos los arcos que faltan y todas las etiquetas. En número de nodos necesarios para ejecutar las nuevas instrucciones dependerá de la solución que propongáis, pero no serán 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. *(1 punto)*

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 de la nueva instrucción (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 b). *(1 punto)*

Nodo	Mnemotécnico	Acciones
E0	F	//
E1	D	// //
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 0 (F), 1 (D) y las 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**. *(1.5 puntos)*

@ROM	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***
0																			F
1																			D
18																			
19																			
20																			
21																			
22																			
23																			