Apellidos: ..... Nombre: .....

DNI: .....

1. ¿Cuál es la función de la unidad de control del procesador de un computador?

   La unidad de control es la encargada de generar y secuenciar las señales eléctricas que sincronizan el funcionamiento tanto del propio procesador, mediante señales internas del circuito integrado, como del resto del ordenador, con líneas eléctricas que se propagan al exterior a través de los pines de conexión del procesador.

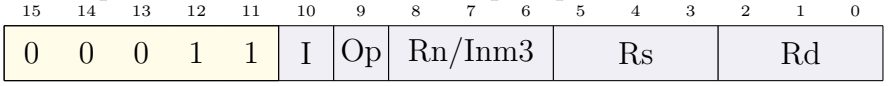
2. ¿En qué fases de la ejecución de una instrucción interviene (o puede intervenir) la memoria? ¿Qué acciones realiza en cada una de ellas?

   Interviene en la fase de *lectura de la instrucción* y puede intervenir, dependiendo de la instrucción, en la fase de *ejecución de la instrucción*.

   En la fase de lectura de la instrucción, proporciona al procesador la instrucción almacenada en la dirección indicada por el procesador.

   En la fase de ejecución de la instrucción, si la instrucción escribe o lee en memoria, proporcionará el valor almacenado en la dirección que se le haya indicado o escribirá el valor que se le haya proporcionado en la dirección que se le haya indicado.

3. Codifica en binario y en hexadecimal la instrucción «**`add r2, r3, #5`**» sabiendo que el formato de instrucción utilizado para dicha instrucción es el que aparece a continuación:



I Inmediato: 1, inmediato; 0, registro.

Op Tipo de operación: 1, resta; 0, suma.

Rn/Inm3 Registro o dato inmediato.

Rs Registro fuente.

Rd Registro destino.

El contenido en binario de cada uno de los campos, de izquierda a derecha, será:

- `00011`, que es fijo y nos indica de qué tipo de instrucción se trata.
- El campo `I` es un 1, pues se trata de una instrucción con dato inmediato.
- El campo `Op` es un 0, pues se trata de una instrucción de suma.
- El campo `Rn/Inm3` codifica el dato inmediato, 5, `101`.
- El registro fuente es `r3`, `011`.
- El registro destino es `r2`, `010`.

Juntando los campos se tiene:

Que en hexadecimal es 0x1D5A.

4. Dado el modo de direccionamiento *relativo al PC* utilizado en las instrucciones de control de flujo de ARM Thumb,

   a) ¿Cuál es la dirección efectiva de un operando que utilice dicho modo de direccionamiento en una instrucción de control de flujo?
   La dirección efectiva es una posición de memoria, cuya dirección viene dada por la suma del PC+4 más un desplazamiento, que se obtiene multiplicando por 2 un número indicado en la propia instrucción.

   b) Indica dos instrucciones de control de flujo que utilicen dicho modo de direccionamiento especificando cuáles de sus operandos lo usan.
   La instrucción de salto incondicional, «**`b`**», y la instrucción de salto condicional, «**`beq`**». Este modo de direccionamiento se usa para indicar el operando fuente en ambas instrucciones.

   c) ¿Qué campo o campos son necesarios para codificar dicho modo de direccionamiento en una instrucción de control de flujo? (especifica qué información se almacena en este/estos)
   Solo es necesario un campo, en el que se especifica el desplazamiento (que se codificará dividido entre 2). El número de bits empleado dependerá del tipo de instrucción de control de flujo.

5. Dada el siguiente fragmento de código en ensamblador Thumb de ARM:

   - Describe su funcionamiento comentando adecuadamente cada una de sus líneas.
   - Indica qué valores habrá en el momento de ejecutar la instrucción «`mov pc, lr`» en los registros indicados, sabiendo que la zona de datos comienza en la dirección `0x2007 0000`.

```
1         .data
2 player: .word 24, 12    @ vector player con 2 elementos
3
4 enemy:  .word 20, 10    @ vector enemy con 4 elementos
5         .word 34, 16
6
7         .text
8 main:   ldr r7, =player  @ r7 <- dirección vector player
9         ldr r0, [r7]     @ r0 <- player[0]
10        ldr r7, =enemy   @ r7 <- dirección vector enemy
11        ldr r1, [r7]     @ r1 <- enemy[0]
12        ldr r2, [r7, #8] @ r2 <- enemy[2]
13        bl coldet        @ Llama a la subrutina coldet
14        @ ...
15        wfi
16
17 coldet: mov r3, #0      @ r3 <- 0
18        cmp r0, r1       @ Compara r0 con r1 y
19        blt cend         @  si r0 es menor que r1, salta a cend
20        cmp r0, r2       @ Compara r0 con r2 y
21        bgt cend         @  si r0 es mayor que r2, salta a cend
22        mov r3, #1       @ r3 <- 1
23 cend:  mov r0, r3       @ r0 <- r3
24        mov pc, lr       @ Retorna de la subrutina
```

El programa muestra parte de un código de detección de colisiones inspirado en la del juego Contra de NES (ver «Game Development in Eight Bits» de Kevin Zurawel <`https://youtu.be/TPbroUDHG0s?t=1094`>). En concreto, la subrutina «`coldet`» devuelve un 1 en `r0` si una de las coordenadas asociadas al jugador está dentro de las coordenadas de la caja asociada a uno de los enemigos. El fragmento de código mostrado tan solo comprueba las coordenadas en el eje «x», falta el código que comprueba las coordenadas en el eje «y» y el que decide que se ha producido una colisión si en ambos ejes se ha obtenido un 1.

El programa principal carga en `r0` la coordenada «x» del jugador (el elemento 0 del vector «`player`», «`player[0]`»); en `r1` la coordenada «x1» del enemigo (el elemento 0 del vector «`enemy`», «`enemy[0]`»); en `r2` la coordenada «x2» del enemigo (el elemento 2 del vector «`enemy`», «`enemy[2]`»); y llama a la subrutina «`coldet`».

La subrutina «`coldet`» inicializa el registro `r3` a 0 y solo lo cambia a 1 si el contenido de `r0` es mayor o igual al de `r1` y menor o igual al de `r2`. Puesto que en este caso se cumplen ambas condiciones, `r3` pasa a tomar el valor 1 y al copiarlo sobre `r0`, este también pasa a valer 1.

r0	1
r1	20
r2	34
r3	1
r7	0x20070008