Soluciones Ejercicios Tema 4-1

- **Pregunta 1**: Empareja de manera correcta:
    - ADD r1, r1, #1 → Se actualiza r1 con r1+1.
    - ADDEQ r1, r1, #1 → En este caso, la instrucción está condicionada por el valor del registro Z. Como tenemos el sufijo EQ, solamente se ejecuta la instrucción cuando Z==1. Por lo tanto, la respuesta correcta es solo se actualiza r1 con r1+1 si el registro de estado Z==1.
    - ADDNE r1, r1, #1 → De nuevo, tenemos una instrucción condicionada por el valor de Z. En este caso, como usamos el sufijo NE, solo se ejecuta cuando Z!=1. Tanto en este caso como en el anterior, el valor del registro Z ha tenido que ser actualizado por una instrucción previa.

- **Pregunta 2**: Empareja la secuencia de instrucciones en ensamblador de ARM con el estado resultante tras su ejecución, sabiendo que cuando se realiza el fetching de la última instrucción de la secuencia (el salto) LR=236 y PC=60, siendo el valor de la etiqueta "destino" 324.
    - SECUENCIA 1 → Estado 3. En esta secuencia, la clave es el tipo de salto: bl. Dado que es un salto incondicional, se va a ejecutar siempre independientemente del valor del registro Z, por lo tanto, PC=324 (destino). Por último, como es una operación BL (branch and link) el registro LR se actualiza con la dirección de la siguiente instrucción: 60 + 4, por lo tanto LR=64. De esta forma podemos hacer un retorno para seguir por donde iba el programa.
    - SECUENCIA 2 → Estado 1. En este caso, el salto es condicional y depende del valor del registro Z. Nos centramos en la operación ANDS que es la que actualiza el valor de Z. Esta operación aplica una AND lógica sobre los valores de R1 (1) y R0 (2). Si pasamos dichos valores a binario, tenemos una operación AND(01b, 10b), cuyo resultado es 00b. Por lo tanto, como el resultado es 0, el valor de Z pasa a ser Z=1 y se ejecuta el salto.
    - SECUENCIA 3 → Estado 2. Este caso es similar al anterior, solo cambia la condición de salto, NE. Es decir, saltamos si el resultado no es igual, por lo que Z debe ser 0. Como hemos visto, el valor de Z es 1 por lo que no se produce el salto y PC=64 y LR=236.
    - SECUENCIA 4 → Estado 1. De nuevo, el tipo de salto es incondicional por lo que siempre se produce y el valor de PC=324 (destino). Dado que es una operación de salto sin linkado, el valor de LR no se ve alterado por lo que LR=236 (su valor inicial).

- **Pregunta 3**: La sección de datos de un programa ARM se ubica en memoria como se muestra a continuación:
    - Caso 1. Cuando se alcanza la etiqueta "watch" el valor de r0 es: 140h puesto que el inmediato =tam se traduce en la dirección de memoria asociada a la variable tam (140h).
    - Caso 2. Cuando se alcanza la etiqueta "watch" el valor de r1 es: 8d porque la operación ldr r1, [r0] carga el valor almacenado en la dirección de memoria contenida en r0, que previamente ha sido inicializado con la dirección de memoria de tam (140h).

- Caso 3. Cuando se alcanza la etiqueta "watch" el valor de r1 es: 12d y el valor de r0 es 140h. En este caso, la operación ldr r1, [r0, #4] primero suma 4 al valor contenido en r0 que, en nuestro caso, es la dirección de memoria de tam (140h). Por lo tanto [r0, #4] se traduce en 144h y lo que haya en esa dirección de memoria es lo que se carga en r1. Si nos fijamos en las variables que hemos declarado, en la dirección 144h está almacenado el primer valor del array datos, es decir 12.
- Caso 4. Cuando se alcanza la etiqueta "watch" el valor de r1 es: 12 y el valor de r0 es 144h. Este caso es muy similar al anterior salvo por el símbolo ! en la operación ldr r1, [r0, #4]!. El símbolo ! indica que es una operación pre-indexada y el valor del registro contenido en los corchetes debe actualizar su valor. En nuestro caso, al valor almacenado en el registro r0 se le suma 4, por lo que su valor pasa a ser r0=144h. En cuanto a r1, no hay diferencia con respecto al caso anterior y su valor es r1=12 ya que la actualización de r0 se produce antes de la operación ldr sobre r1.
- Caso 5. Cuando se alcanza la etiqueta "watch" el valor de r1 es: 8 y el valor de r0 es 144h. En este caso, tenemos una operación post-indexada, es decir, la actualización del registro indexado se produce DESPUÉS de la operación ldr. Por lo tanto, en r1 se carga el valor almacenado en la dirección de memoria que contiene r0 (140h). Una vez se ha hecho esta carga, se actualiza el valor de r0 sumándole 4, por lo que pasa a ser 144h.

- **Pregunta 4**: Dada la siguiente sección de código en ensamblador del ARM:
  - cuando se alcanza la etiqueta "watch" el valor de r1 es 2d. Como r0 y r1 contienen valores diferentes, la operación cmp r0, r1 pone el registro Z=0. Por lo tanto, la operación orreq no se ejecuta ya que la condición (EQ) no se cumple. Sin embargo, la operación adne si que se ejecuta ya que la condición es NE, es decir, que Z=0. Por lo tanto, si aplicamos la operación AND sobre los valores binarios contenidos en r0, r1, tenemos que: AND(010b, 011b) = 010b, es decir, 2 en decimal, que es el valor que se almacena en r1.
  - cuando se alcanza la etiqueta "watch" el valor de r1 es 3d. En este caso, dado que los registros contienen el mismo valor, se ejecuta la operación orreq sobre los valores binarios de r0 y r1: ORR(011b, 011b) = 011b, es decir, 3 en decimal, que se almacena en r1.

- **Pregunta 5**: Dado el siguiente código de alto nivel:
  - a. SOL1. Comenzamos viendo el funcionamiento del código. Lo primero que hacemos es cargar en r0 el valor del contador i, es decir 9. En r1 almacenamos la suma, que se irá actualizando a lo largo de las iteraciones. Su valor inicial es 0, de acuerdo con el código fuente en alto nivel. Tras esto, entramos al cuerpo del bucle donde actualizamos el valor de la suma contenido en r1 y decrementamos el valor del contador i almacenado en r0. Por último, bge se realiza mientras el valor de r0 sea mayor o igual que 1. A priori, parece estar bien, sin embargo se ha cometido un error crítico, la operación sub no actualiza los flags por lo que el bucle no va a parar nunca. Para que se actualicen los flags, se debería añadir el sufijo “s” a la operación sub, pasando a ser subs.

    - b. SOL2. Este caso es exactamente igual al anterior pero incluyendo el sufijo “s” a la instrucción sub. Por lo tanto, es correcto.
    - c. SOL3. En este caso, se actualizan los flags pero la operación de salto es diferente, se produce mientras que el valor de r0 sea mayor que 1. Por lo tanto, estamos realizando una iteración menos de lo que deberíamos.

- **Pregunta 6**: Queremos inicializar el registro r0 con el valor entero 3. ¿Cuáles de las siguientes instrucciones lo hacen?
    - a. mov r0, #0x03. Correcto, 3 en hexadecimal equivale a 3 en decimal.
    - b. mov r0, #0b011. Correcto, es 3 en binario.
    - c. mov r0, #3. Correcto, es el valor en decimal.

- **Pregunta 7**: Queremos inicializar el registro r0 con el valor entero -3. ¿Cuáles de las siguientes instrucciones lo hacen?
    - a. mov r0, #-0b11. Correcto. El - se puede usar con cualquier tipo de dato (binario, decimal o hexadecimal) y el ensamblador lo traduce en complemento a 2.
    - b. mov r0, #-3. Correcto, al igual que el caso anterior.
    - c. mov r0, #0xFFFFFFFD. Correcto, es -3 en complemento a 2 expresado en hexadecimal.

- **Pregunta 8**: Necesitamos que el bloque de código cuya primera instrucción está etiquetada como "inicio2" sólo se ejecute si el bit 3 del registro r4 tiene el valor 1 (recuerda que el bit menos significativo tiene índice 0), continuándose con la ejecución secuencial en caso contrario. Las soluciones propuestas son:
    - a. Solución 1. Incorrecta, la operación cmp mira si r4 y r0 tienen el mismo valor. El enunciado nos pide que el valor del tercer bit sea 1, este código no hace lo que se pide.
    - b. Solución 2. Correcta. Lo primero es definir la máscara con un 1 en la tercera posición y el resto de valores a 0. Por lo tanto, esa máscara equivale a un 8 en decimal (se podría usar una máscara de 32 bits con un 1 en la posición 3. La forma de inicializar la máscara es indiferente). Una vez definimos la máscara, la operación tst realiza una AND lógica entre r4 y r0. Esta operación devolverá un 1 en la posición 3 si el valor de la tercera posición de r4 es 1. Tras esto, se actualizan los flags y pasamos a la operación bne. Puesto que la operación tst coloca Z=1 cuando el resultado es 0, debemos saltar cuando esto no se cumple puesto que debe devolver 1 en la tercera posición.
    - c. Solución 3. Correcta. Esta solución es exactamente igual a la segunda solo que almacenamos el resultado de la AND lógica en R3. La instrucción TST es equivalente a ANDS pero sin almacenar el resultado de la operación lógica.
    - d. Solución 4. Incorrecta. Al igual que la solución 1, se comprueba el valor de los registros, no el del tercer bit. Por lo tanto, el código no hace lo que se pide.
    - e. Solución 5. La idea de esta solución es rotar los bits para dejar el tercer bit en la posición 0 y luego comparar con el valor 1. Sin embargo, la rotación se está realizando con la instrucción lsl, que rota los bits hacia la izquierda por lo

que estamos alejando el bit que queremos comparar de la posición 0. Habría que usar LSR para rotar a la derecha y el código estaría correcto.