TP8 Ensamblado y desensamblado de LEGv8

> **Ejercicio 1:**
> Extender los siguientes números de 26 bits en complemento a dos a 64 bits. Si el número
> es negativo verificar que la extensión a 64 bits codifica el mismo número original de 26 bits.

1.1) 00 0000 0000 0000 0000 0000 0001

- Primero podemos observar que el bit mas significativo es “0”, y como esta escrito en complemento a dos, nos dice que el numero es POSITIVO.
- Como es positivo, lo que queremos hacer es extenderlo de 26bits a 64bits, (obviamente manteniendo el valor inicial). Por lo tanto lo que vamos hacer es agregar 0 a la izquierda.
- La cantidad de 0 a agregar es hasta completar el registro de 64 bits, entonces 64-26 = 38. Hay que agregar 38 ceros a la izquierda:

Respuesta: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001

Una forma de ver esto intuitivamente por ejemplo, si en decimal tenemos el numero 15, y quiero que tenga 4 cifras, lo que hago es agregarle 2 ceros a la izquierda: 0015 y me queda el mismo numero pero representado con mas cifras.

1.2) 10 0000 0000 0000 0000 0000 0000

- Observemos que el numero es negativo, ya que el bit mas significativo es “1” lo que indica, en complemento a 2, que el numero representado es negativo.
- Como es negativo, y lo que queremos hacer es extenderlo de 26bits a 64bits, manteniendo el valor inicial, vamos a agregarle unos a la izquierda.
- La cantidad de unos va a ser hasta completar el registro de 64bits, entonces 64-26 = 38. Por lo tanto hay que agregar 38 bits a la izquierda:

Respuesta: 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 0000 0000 0000 0000 0000

Verificación:

Como el numero es negativo vamos a verificar que la extensión a 64bits codifica el mismo numero original de 26 bits.

Yo lo voy hacer pasando los dos numero a decimal y comprobando que sean los mismos.

1- Valor en 26 bits: **10 0000 0000 0000 0000 0000 0000**

- Invertimos todos los números: 01 1111 1111 1111 1111 1111 1111
- Sumamos 1: 10 0000 0000 0000 0000 0000 0000
- Convertimos a decimal: $2^{(25)}$ (2^25)
- Agregamos el signo negativo ya que el bit mas significativo del numero original era 1 lo que nos indicaba en complemente a 2 que el numero es negativo: -$2^{(25)}$ -(2^25)

2- Valor en 64 bits: **1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 0000 0000 0000 0000 0000**

- Invertimos todos los números: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1111 1111 1111 1111
- Sumamos 1: 0000 0000 0000 0000 0000 0000 0000 0000 0010 0000 0000 0000 0000 0000 0000
- Convertimos a decimal: $2^{(25)}$ (2^25)
- Agregamos el signo negativo ya que el bit mas significativo del numero original era 1 lo que nos indicaba en complemente a 2 que el numero es negativo: -$2^{(25)}$ -(2^25)

Como vemos que el valor numérico de los dos números son los mismos, podemos deducir que el valor inicial del numero no se perdió.

> **Ejercicio 2:**
> Tenemos las siguientes instrucciones en assembler LEGv8:
> ADDI X9, X9, #0
> STUR X10, [X11,#32]

> **2.1)** ¿Qué formato (R, I, D, B, CB, IM) de instrucciones son?

▼ **Formatos de instrucciones en assembler de LEGv8**

Para poder hacer este ejercicio necesitamos conocer los diferentes formatos de instrucciones que existen en esta arquitectura: (Aclaración: Todas las instrucciones son de 32bits)

- R (Registro)
  - Las instrucciones con este formato, operan en registros y no tienen operandos inmediatos.
  - Ejemplos: ADD, SUB, AND, ORR, ETC.

- Core instruction format:

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- Explicacion de los campos
  - *opcode:* (Código de operacion), determina la operación que se va a realizar, permite hasta 2048 (2^11) códigos de operación diferente.
  - *Rm: R*egistro fuente que contiene uno de los operandos para la operación aritmética o lógica. El número del registro se codifica utilizando estos 5 bits. Segundo registro "fuente" de la instrucción.
  - **shamt:** (Cantidad de desplazamiento) especifica la cantidad de bits que se desplazarán en ciertas operaciones de desplazamiento lógico o rotación
  - *Rn:* Primer registro "fuente" utilizado en la operacion. Permite especificar hasta 32 (2^5) registros diferentes.
  - *Rd:* Registro de destino, es donde se almacenara el resultado de la operacion, también permite registrar hasta 32 registros diferentes.

- I (Inmediato)
  - Las instrucciones en formato "I" incluyen un operando inmediato de tamaño pequeño.
  - Ejemplos: ADDI, SUBI, ETC.
  - Core instruction format:

opcode	inmediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

  - Explicación de los campos:
    - *inmediate:* Especifica el valor inmediato que se utilizara en la implementación. El valor inmediato va desde -2048 hasta 2047.
    - **Aclaración:** Los campos con el mismo nombre que el anterior se usan de la misma manera solo que con la cantidad de bits qeu se especifican en el cif.
      - Las instrucciones en formato D utilizan un desplazamiento para acceder a la memoria.

- D (Desplazamiento)
  - Se usan para cargar o guardar valores desde o hacia la memoria. (load, store).
  - Ejemplos: LDUR, STUR, etc.
  - Core instruction format:

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

  - Explicación de los campos:

        - *addres:* Representa la dirección base utilizada en la operación. Contiene la dirección base de la memoria a la que se accederá. permite direccionar hasta 512 (2^9) direcciones diferentes.
        - *op2:* Especifica ciertas variantes de la operación, como cargar o almacenar bytes, medias palabras, palabras, dobles palabras, etc.
        - *Rt:* se llama Rt en lugar de Rd porque para la store de instrucciones, el campo indica una fuente de datos y no un destino de datos. Pero es lo mismo, representa el registro de destino donde se almacenará (o desde donde se cargará) el valor de memoria.

- B (Branch)
    - Realizar saltos condicionales o incondicionales a una direccion especifica en el flujo de ejecución del programa.
    - Ejemplos: B etiqueta
    - Core instruction format:

opcode	BR_address
6 bits	26 bits

    - Explicacion de los campos:
        - *BR_addres:* Especifica la dirección a la que se realizará el salto cuando se ejecute la instrucción de salto.

- CB (Condicional de desplazamiento)
    - Se utilizan para realizar saltos condicionales a una dirección específica en el código del programa basadas en el estado de las banderas de condición.
    - Ejemplos: B.EQ, CBZ, CBNZ, ETC.
    - Core instruction format:

opcode	Num instructions	Rt
8 bits	19 bits	5 bits

    - Explicación de los campos:
        - *Num instructions:* gdagadgdasg

- IM (Multiplo)
    - Se usan para operaciones aritméticas que implican registros y un inmediato extendido
    - Ejemplos: MOVZ, MOVK, ETC.
    - Core instruction format:

opcode	lsl	mov_inmediate	Rd
9 bits	2 bits	16 bits	5 bits

    - Explicación de los campos:

- *lsl:* Indica la cantidad de bits que se desplazará hacia la izquierda el valor inmediato antes de ser movido al registro de destino.
- *mov_inmediate:* Especifica el valor inmediato que se moverá al registro de destino. El valor de este campo determina el valor que se colocará en los bits del registro de destino después de aplicar el desplazamiento.

ADDI X9, X9, #0 es de formato I (ya que usa el operando inmediato #0)

STUR X10, [X11,#32] es de formato D (ya que hace un desplazamiento de #32 para acceder a la memoria en x11 y guardarlo en x10.)

2.2) Ensamblar a código de máquina LEGv8, mostrando sus representaciones en binario y luego en hexadecimal.

1- ADDI X9, X9, #0

- Por el ejercicio 2.1 observamos que esta instrucción es de tipo "I".
- Entonces la sección esta dividida así:

opcode	ALU_ inmediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

- **Opcode:** Viendo en la green card, observamos que el opcode de ADDI es **1001000100**
- El valor de ALU_inmediate, vemos que es #0, entonces como son 12 bits, son 12 ceros: **000000000000**
- El valor de **Rn** es x9, que en binario es: 1001, llenando el registro de 5 bits: **01001**
- El valor de **Rd** también es x9, usando la misma lógica es: **01001**
- Uniendo estos valores, vemos que:
  - ADDI X9, X9, #0 ensamblado en binario es: 1001000100 000000000000 01001 01001
  - ADDI x9, x9, #0 ensamblado en hexadecimal es: 0x91000249

2- STUR X10, [X11, #32]

- Por ele ejercicio 2.1 sabemos que la instrucción es de tipo "D"
- Entonces el formato es el siguiente:

opcode	dt_address	op	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- Viendo en la greencard el **opcode** es: **11111000000**
- El **Rt** es x10, entonces es: 1010, llenando el registro de 5bits: **01010**
- El **Rn** es x11, entonces es: 1011, llenando el registro de 5bits: **01011**

- El **dt_address** es #32, que en binario es: 100000, pero como tiene que tener 9 bits nos queda: **000100000.**
- El **op** en el stur no se utiliza entonces lo ponemos con 0s: **00**
- Reescribiendo estos valores obtenemos:
  - Ensamblado en binario: **11111000000 000100000 00 01011 01010**
  - Ensamblado en hexadecimal: **0xF802016A**

> **Ejercicio 3:**
> Dar el tipo de instrucción, la instrucción en assembler y la representación binaria de los
> siguientes campos de LEGv8:

3.1) op=0x658, Rm=13, Rn=15, Rd=17, shamt=0

1- Primero vamos a determinar que tipo de instrucción estamos viendo:

- El opcode es 0x658, viendo en la green card, observamos que es la instrucción SUB.
- La instrucción sub, sabemos que es de tipo “R”.
- Las instrucciones con formato tipo “R” se establecen asi:

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

2- Ahora desglosemos los campos:

- **Rm** = 13, en binario = 1101, completamos los 5 bits: **01101**
- **Rn** = 15, en binario = 1111, completamos los 5 bits: **01111**
- **Rd** = 17, en binario = **10001,** ya hay 5 bits.
- **shamt** = 0, en binario = 0, completamos los 5 bits: **00000**

3- Por ultimo unimos todo esto siguiendo el orden de la instrucción:

- 11001011000 **01101 00000 01111 10001**

4- Respondemos la consigna:

- **Tipo de instrucción:** sub (por 1) **Tipo de instrucción:** sub (por 1)
- **Instrucción en assembler:** sub x17, x15, x13 (esto porque viendo la green card sabemos que la instrucción sub R[Rd] = R[Rn] - R[Rm], entonces simplemente copiamos cada data, primero escribimos el nombre de la instrucción, después el Rd el registro donde se va a almacenar nuestra operacion, Rn el primer registro que vamos a usar, y Rm el segundo registro que vamos a utilizar.)
- **Representación binaria:** 11001011000 01101 00000 01111 10001 (por 3)

3.2) op=0x7c2, Rn=12, Rt=3, const=0x4

1- Primero vamos a determinar que tipo de instrucción estamos viendo:

- El opcode es 0x7C2, viendo en la green card, observamos que es la instrucción **ldur**.
- La instrucción ldur, es de tipo “D”
- Las instrucciones con formato tipo “D” se establecen así:

opcode	DT_Address	op	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

2- Ahora desglosemos los campos:

- **Rn** = 12, en binario = 1100, completamos los 5 bits: **01100**
- **Rt** = 3, en binario = 0011, completamos los 5 bits: **00011**
- **DT_Address** = 0x4, en binario = **000000100**
- **op** = 0,completamos los 2 bits: **00**

3- Por ultimo unimos todo esto siguiendo el orden de la instrucción:

- **11110110010 000000100 00 01100 00011**

4- Respondemos la consigna:

- **Tipo de instrucción:** ldur (por 1)
- **Instrucción en assembler:** ldur x3, [x12 + #4] (esto porque viendo la green card sabemos que la instrucción ldur R[Rt] = M[R[Rn]] + DTaddr, entonces simplemente seguimos los datos y copiamos tal cual.
- **Representación binaria:** 11110110010 000000100 00 01100 00011 (por 3)

Ejercicio 4:

Transformar de binario a hexadecimal. ¿Qué instrucciones LEGv8 representan en memoria?

4.1) 1000 1011 0000 0000 0000 0000 0000 0000

Nos damos cuenta que tenemos que hacer como “un proceso inverso” del punto 3.

1- Pasamos el código en binario a hexa:

- 1000 1011 0000 0000 0000 0000 0000 0000 = 0x8B000000

2- Sabemos que una instrucción en LEGv8 tiene un campo de 11 bits para el opcode seguido por otros campos específicos según el tipo de instrucción (R, I, D, B, CB, IW). Sabiendo esto obtengamos los primeros 11 bits: **0100 0101 1000** (aclaración como son 11 bits completo con 0 para poder pasarlo a hexa)

3- Pasemos estos 11 bits a hexadecimal: **0100 0101 1000 = 0x458**

4- Viendo en la green card la instrucción que corresponde al opcode 0x458 es “ADD”

5- Ahora sabiendo que la instrucción es “ADD” de tipo “R” simplemente desglosamos la cantidad de bit que le corresponde a cada parte de la instrucción. (Esto gracias a que sabemos que es de tipo “R”).

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

6- Desglosando: (Dividimos el binario de (1) en los bits que le corresponden)

- **opcode: 100 0101 1000**
- **Rm:** 00000
- **shamt:** 00000
- **Rn:** 00000
- **Rt:** 00000

7- Por ultimo desarrollamos para armar nuestra instrucción:

add x0, x0, x0

4.2) 1101 0010 1011 1111 1111 1111 1110 0010

1- Pasamos el código en binario a hexa:

- **1101 0010 1011 1111 1111 1111 1110 0010** = 0xD2BFFFE2

2- Sabemos que una instrucción en LEGv8 tiene un campo de 11 bits para el opcode seguido por otros campos específicos según el tipo de instrucción (R, I, D, B, CB, IW). Sabiendo esto obtengamos los primeros 11 bits: **01101 0010 101** (aclaración como son 11 bits completo con 0 para poder pasarlo a hexa)

3- Pasemos estos 11 bits a hexadecimal: **01101 0010 101 = 0x695**

4- Viendo en la green card la instrucción que corresponde al opcode 0x695 es “MOVZ”

5- Ahora sabiendo que la instrucción es “MOVZ” de tipo “IM” simplemente desglosamos la cantidad de bit que le corresponde a cada parte de la instrucción. (Esto gracias a que sabemos que es de tipo “IM”).

opcode	lsl	mov_inmediate	Rd
9 bits	2 bits	16 bits	5 bits

6- Desglosando: (Dividimos el binario de (1) en los bits que le corresponden)

Aclaraciones: el opcode en este caso solo tiene 9 bits, así que debemos prestar atención con eso.

- **opcode:** 110100101
- **lsl:** 01 = #16
- **mov_inmediate:** 1111111111111111 = 0xFFFF
- **Rd:** 00010 = #2

7- Por ultimo desarrollamos para armar nuestra instrucción:

MOVZ X2 0xFFFF, LSL #16

Aclaración: lsl 01 = # 16 ya que tenemos 2 bits para usar en este campo y son los siguiente:

- `00` → No desplazamiento (LSL #0, bits 0-15)
- `01` → Desplazamiento de 16 bits (LSL #16, bits 16-31)
- `10` → Desplazamiento de 32 bits (LSL #32, bits 32-47)
- `11` → Desplazamiento de 48 bits (LSL #48, bits 48-63)

> ### Ejercicio 5:
>
> Ejecutar el siguiente código assembler que está en memoria para dar el valor final del
> registro X1. El contenido de la memoria se da como una lista de pares, dirección de
> memoria: contenido, suponiendo alineamiento de memoria del tipo big endian. Describa
> sintéticamente que hace el programa.
>
> **0x10010000: 0x8B010029**
> **0x10010004: 0x8B010121**

Primero vamos a ver el contenido de la primera instrucción, para eso convertimos la instrucción a binario y decodificamos:

1. **0x8B010029** en binario es: **1000 1011 0000 0001 0000 0000 0010 1001**
2. **Decodificación:**

   Para decodificar seguimos los mismo pasos

   1. Los primeros 11 bits nos dicen de que instrucción se trata: 1000 1011 000 = 0100 0101 1000 = 0x458 (viendo en la green card el opcode que corresponde a 0x458 es la operacion ADD)
   2. Sabiendo que ADD es de tip “r” dividamos cada bits en su correspondiente ubicación:

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

   3. Ahora desglosamos:

   - **Opcode:** 1000 1011 000
   - **Rm:** 0 0001 = 0000 0001 = 0x01
   - **Shamt:** 0000 00 = 0000 0000 = 0x0

- **Rn:** 00 001 = 0000 0001 = 0x01
- **Rd:** 0 1001 = 0000 1001 = 0x09

4. Completamos toda la instruccion: ADD x9, x1, x1

3. Por lo tanto la primera instrucción es: ADD x9, x1, x1

Ahora vamos a hacer lo mismo para la segunda instrucción:

1. **0x8B010121** en binario es: **1000 1011 0000 0001 0000 0001 0010 0001**
2. **Decodificación:**

   Para decodificar seguimos los mismo pasos

   1. Los primeros 11 bits nos dicen de que instrucción se trata: 1000 1011 000 = 0100 0101 1000 = 0x458 (viendo en la green card el opcode que corresponde a 0x458 es la operacion ADD)
   2. Sabiendo que ADD es de tip “r” dividamos cada bits en su correspondiente ubicación:

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

   3. Ahora desglosamos:

- **Opcode:** 1000 1011 000
- **Rm:** 0 0001 = 0000 0001 = 0x01
- **Shamt:** 0000 00 = 0000 0000 = 0x0
- **Rn:** 001 001 = 0000 1001 = 0x09
- **Rd:** 0 0001 = 0000 0001 = 0x01

   4. Completamos toda la instruccion: ADD x1, x9, x1

4. Por lo tanto la segunda instrucción es: ADD x1, x9, x1

Entonces el bloque completo es:

0x10010000: 0x8B010029 (Add x9, x1, x1)
0x10010004: 0x8B010121
(Add x1, x9, x1)

Sintéticamente lo que hace el programa es multiplicar por tres a x1.

(Esto ya que en la primera instruccion en x9 guarda la operacion de x1 + x1 = 2*x1,por lo tanto en x9 tenemos guardado 2*x1, despues en x1, va a guardar la suma de x9 con x1, es decir x1 = x9 + x1

= x1 + x1 + x1 = 3 * x1)

> **Ejercicio 6:**
>
> Decidir cuáles de las siguientes instrucciones en assembler se pueden codificar en código
> de máquina LEGv8. Explique qué falla en las que no puedan ser ensambladas

1. **LSL XZR, XZR, 0** (Recordemos que LSL lo que hace es mover en este caso 0 bits a la izquierda del registro XZR y lo guarda en XZR) Por lo tanto esta instruccion si se puede hacer, pero sos medio boludo si lo haces, ya que no hace nada.
2. ADDI X1, X2, -1 (No ya que al ver la green card, vemos que ADDI se define asi: R[Rd] = R[RN] + ALUImm y ALUImm = {52'b0, ALU_INmediate), los que no sdice que los primeros 52bits son ceros, y los restantes son el valor inmediato, por lo tanto como los primeros son ceros es imposible que haya un numero negativo, ya que para eseto tendria que emepzar con 1)
3. ADDI X1, X2, 4096 (Al igual que antes, como los primeros 52 bits son ceros, el numero mas grande que puedo representar es: 0111111111111 = 2047, por lo tanto no puedo representar a 4096)
4. EOR X32, X31, X30 (No ya que x32 y x31 no existen en legv8, solo hay 31 registros (x0 ... x30)
5. ORRI X24, X24, 0x1FFF (No, ya que 0x1FFF = 0001 1111 1111 1111 = 8191, asi que usando la mism justificacion que (3) no se puede)
6. STUR X9, [XZR,#-129] (Si, ya que el stur me permite el offset negativo)
7. LDURB XZR, [XZR,#-1] (Si, al igual que en el anterior ldurb me permite el offset negativo)
8. LSR X16, X17, #68 (No, ya que el LSR es de tipo R, entonce el espacio para el shamt son 5 bits, el maximo numero que podemos poner entonces es: 11111 = 31 y como 68 > 31 no podemos usar 68 como shamt ya que no entra en 5bits)
9. MOVZ X0, 0x1010, LSL #12 (No porque el LSL puede ser 0, 16, 32 o 48, por lo tanto no podemos hacer LSL #12)
10. MOVZ XZR, 0xFFFF, LSL #48 (Si, 0xFFFF = 1111 1111 1111 1111 = 65535 y el inmediato maximo que puedo obtener es 1111111111111111 = 65535, por lo tanto no habria problema, y se puede hacer LSL #48, asi que tampoco habria problemas.)

> 7. Decidir cuáles de las siguientes instrucciones en assembler se pueden codificar en código
> de máquina LEGv8. Explique qué falla en las que no puedan ser ensambladas.
>
> Ensamblar estos delay loops:

```
    MOVZ X0, 0x1, LSL #48
L1: SUBI X0,X0,#1
    CBNZ X0, L1
```

Para ensamblar es el proceso inverso de desensamblar.

Empecemos con la primera instrucción: **MOVZ x0, 0x1, LSL #48**

- **Opcode:** 110100101
- **MOV_Immediate:** 0x1 = 0001
- **Rd:** 0000
- **LSL:** #11 (en el ejercicio 4 esta explicado el porque de esto)

Entonces MOVZ x0, 0x1, LSL #48 = 110100101 11 0000000000000001 0000

Pasándolo en hexa = 1101 0010 1110 0000 0000 0000 0010 0000 = **0xD2E00020**

Sigamos con la siguiente instrucción: **SUBI x0, x0, #1**

- **Opcode:** 1101000100
- **Alu_immediate:** 000000000001
- **Rn:** 00000
- **Rd:** 00000

Entonces SUBI x0, x0, # 1 = 1101000100 000000000001 00000 00000

Pasándolo en hexa: 1101 0001 0000 0000 0000 0100 0000 0000 = **0xD1000400**

Y la ultima instrucción es: **CBNZ X0, L1**

- **Opcode:** 10110101
- **COND_BR_Address:** 1111111111111111111 = -1 (ya que voy una instrucción para atrás)
- **Rt:** 00000

Entonces CBNZ x0, L1 = 10110101 1111111111111111111 00000

Pasándolo a hexa: 1011 0101 1111 1111 1111 1111 1110 0000 = **0xB5FFFFE0**

Por lo tanto todo el bloque del delay loop ensamblado es el siguiente:

0xD2E00020

0xD1000400

0xB5FFFFE0

```
MOVZ X0, 0xFFFF, LSL #32
L1: SUBIS X0,X0,#1

    B.NE L1
```

Empecemos con la primera instrucción: **MOVZ X0, 0xFFFF, LSL #32**

- **Opcode:** 110100101
- **MOV_Immediate:** 0xFFFF = 1111 1111 1111 1111
- **Rd:** 00000
- **LSL:** #32 = 10 (en el ejercicio 4 esta explicado el porque de esto)

Entonces **MOVZ X0, 0xFFFF, LSL #32** = 110100101 10 1111 1111 1111 1111 00000

Pasándolo en hexa = 1101 0010 1101 1111 1111 1111 1110 0000 = **0xD2DFFFE0**

Sigamos con la segunda instruccion: **SUBIS X0,X0,#1**

- **Opcode:** 1111000100
- **ALU_Immediate:** 000000000001
- **Rn:** 00000
- **Rd:** 00000

Entonces **SUBIS X0,X0,#1** = 1111000100 000000000001 00000 00000

Pasándolo en hexa = 1111 0001 0000 0000 0000 0100 0000 0000 = **0xF1000400**

La ultima instruccion es: B.NE L1

- **Opcode:** 01010100 (op code general de B.cond)
- **COND_BR_Address:** 1111111111111111111 = -1 (salto a la instruccion anterior)
- **Rt:** 00001(corresponde al NE)

Entonces B.NE L1 = 01010100 1111111111111111111 00001

Pasándolo en hexa = 0101 0100 1111 1111 1111 1111 1110 0001 = **0x54FFFFE1**

Por lo tanto todo el bloque del delay loop ensamblado es el siguiente:

0xD2DFFFE0

0xF1000400

0x54FFFFE1

```
MOVZ X0, 0x2, LSL #16
L1: SUBIS XZR,X0,#0
B.EQ EXIT
SUBI X0,X0,#1
B L1
EXIT:
```

Empecemos con la primera instrucción: **MOVZ X0, 0x2, LSL #16**

- **Opcode:** 110100101
- **MOV_Immediate:** 0x2 = 0000000000000010
- **Rd:** 00000
- **LSL:** #16 = 01 (en el ejercicio 4 esta explicado el porque de esto)

Entonces **MOVZ X0, 0x2, LSL #16** = 110100101 01 0000000000000010 00000

Pasándolo en hexa = 1101 0010 1010 0000 0000 0000 0100 0000= **0xD2A00040**

Sigamos con la segunda instruccion: **SUBIS XZR,X0,#0**

- **Opcode:** 1111000100
- **ALU_Immediate:** 000000000000
- **Rn:** 00000
- **Rd:** 11111 (es la representacion del registro XZR, el que siempre es 0)

Entonces **SUBIS XZR,X0,#0** = 1111000100 000000000000 00000 11111

Pasándolo en hexa = 1111 0001 0000 0000 0000 0000 0001 1111 = **0xF100001F**

La siguiente instruccion es: **B.EQ EXIT**

- **Opcode:** 01010100
- **COND_BR_Address:** 0000000000000000011
- **Rt:** 00000

Entonces **B.EQ EXIT** = 01010100 0000000000000000011 00000

Pasándolo en hexa = 0101 0100 0000 0000 0000 0000 0110 0000 = **0x54000060**

La otra instruccion es: **SUBIS X0,X0,#1**

- **Opcode:** 1111000100
- **ALU_Immediate:** 000000000001
- **Rn:** 00000
- **Rd:** 00000

Entonces **SUBIS X0,X0,#1** = 1111000100 000000000001 00000 00000

Pasándolo en hexa = 1111 0001 0000 0000 0000 0100 0000 0000 = **0xF1000400**

La ultima instruccion es: B L1

- **Opcode:** 000101 (op code general de B)
- **BR_Address:** 11111111111111111111111101 = -3 (salto 3 insturcciones anterior)

Entonces B.NE L1 = 000101 11111111111111111111111101

Pasándolo en hexa = 0001 0111 1111 1111 1111 1111 1111 1101 = **0x17FFFFFED**

Por lo tanto todo el bloque del delay loop ensamblado es el siguiente:

0xD2A00040

0xF100001F

0x54000060

0xF1000400

0x17FFFFFED

8. Dadas las siguientes direcciones de memoria:

- **0x00014000**
- **0x00114524**
- **0x0F000200**

8.1 Si el valor del PC es 0x00000000, ¿es posible llegar con una sola instrucción conditional branch a las direcciones de memoria arriba listadas?

Para hacer esto veamos como esta estructura el conditional branch en la green card.

Obseravamos que el campo inmediato (COND_BR_Address) es de 19 bits.

Entonces el maximo valor que podemos obtener para adelante es:

0x01111111111111111 = 262143 = 2^18 - 1

La maxima cantidad de posiciones de memoria que puedo saltar hacia adelante es la cantidad de instrucciones (2^18 -1) * 4 bytes (el tamano de cada palabra) = **262143 * 4 = 1048572**

1048572 = 11111111111111111100 = 0xFFFFC

Ahora partiendo del PC 0x00000000 podemos llegar a 0x00000000 + **0x000FFFFC = 0x000FFFFC**

Por lo tanto los que son menores o iguales a **0x000FFFFC** si podemos llegar con una sola instruccion en otro caso no podemos.

- 0x00014000
- 0x00114524

- 0x0F000200

> 8.2 Si el valor del PC es 0x00000600, ¿es posible llegar con una sola instrucción branch a las direcciones de memoria arriba listadas?

Usando el mismo razonamiento que en el ejercicio anterior, pero ahora el inmediato del branch tiene espacio para 26 bits (ver en la green card).

Por lo tanto el maximo valor que podemos mover para adelante es 2^25 - 1.

(Aclaracion en un campo de n bits, el valor maximo signado que se puede obtener es 2^(n-1) - 1)

La cantidad maxima de posiciones de memoria que puedo saltar hacia adelante es: **2^25 - 1 * 4 = 134217724 = 11111111111111111111111100 = 0x7FFFFC**

Por lo tanto partiendo del **PC = 0x00000600**, como maximo con una sola instruccion podemos llegar a **0x00000600** + **0x07FFFFC = 0x080005FC**

Por lo tanto los que son menores o iguales a **0x080005FC** si podemos llegar con una sola instruccion en otro caso no podemos.

- 0x00014000
- 0x00114524
- 0x0F000200

> 8.3 Si el valor del PC es 0x00000000 y quiero saltar al primer GiB de memoria
> 0x40000000 . Escribir exactamente 2 instrucciones contiguas que posibilitan el
> salto lejano (far jump).

Queremos saltar a una dirección específica (0x40000000) desde un valor inicial del PC (0x00000000), voy utilizar dos instrucciones contiguas (ya que es mas facil).

```
LDUR X1, =0x40000000    // Carga el valor inmediato 0x40000000 en el registro X1
BR X1                   // Salta a la dirección almacenada en X1
```

Tambien podemos hacer usando cualquier cosa que cargue una direccion en un registro, por ejemplo el movz.

> 9. Suponiendo que el PC está en la primera palabra de memoria 0x00000000 y se
> desea saltar a la última instrucción de los primeros 4 GiB o sea a 0xFFFFFFFC,
> ¿Cuántas instrucciones B son necesarias? (no se puede usar BR)

Por el ejercicio 8 vimos que la maxima cantidad de posiciones de memoria que puedo saltar usando un branch es: **0x07FFFFFC**

Entonces, por ejemplo, en criollo, si lo maximo que puedo saltar con un solo branch es a 10 y yo quiero llegar a 70, la logica me dice que voy a necesitar 7 branch, esto lo se porque hice 70/10 = 7.

Hagamos lo mismo **0xFFFFFFFC / 0x07FFFFFC = 32,000000923872 (obviamente lo hice en lapiz y papel,na joda lo hice** aca)

Por lo tanto la cantidad de instrucciones qeu necesito son 33.

10. ¿Qué valor devuelve en X0 este programa?

```
.org 0x0000
MOVZ X0, 0x0400, LSL #0
MOVK X0, 0x9100, LSL #16
STURW X0, [XZR,#12]
STURW X0, [XZR,#12]
```

```
MOVZ X0, 0x0400, LSL #0     // x0 = 0x0000000000000400
MOVK X0, 0x9100, LSL #16    // x0 = 0x0000000091000400
STURW X0, [XZR,#12]         // Almacena el valor de X0 en la posición de memo
STURW X0, [XZR,#12]         // sobrescribe la instrucción en la memoria con e

// (**) En ese punto la posición de memoria 12 contiene 0x91000400.
// En la instruccion de abajo de (**) la posición de memoria 12
// contiene la instrucción addi x0, x0, #1.

Detalles de la Instrucción addi
0x91000400 se traduce a:
    opcode: 10010001000 (0x488) => addi (tipo I)
    ALU_immediate: 00000000001 => (#1)
    Rn: 00000 => (x0)
    Rd: 00000 => (x0)

Esto corresponde a la instrucción:
    addi x0, x0, #1  // x0 = x0 + 1

Al finalizar el programa, x0 = 0x91000401
```