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 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 almacena**r** 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