Parcial 2 - 2024

Ejercicio 1:

Escribe un skip, UNA instrucción que no hace nada, con cada nemónico. Si no se puede, poner raya.

ADDI	ADDI XZR, XZR, #0	LDURH	LDURH XZR, [XZR, #0]
ADDS	- (Resetea las flags)	LSL	LSL XZR, XZR, #0
ANDI	ANDI XZR, XZR, XZR	MOVK	MOVK XZR, 0x0, LSL #0
ANDS	- (Setea las flags)	ORR	ORR XZR, XZR, XZR
B.cond	B.EQ 1	STUR	- (modifico algun valor en memoria)
BR	-	STURH	-
CBZ	CBZ x1, 1	SUB	SUB XZR, XZR, XZR
EORI	EORI xzr, xzr, #0	SUBI	SUBI XZR, XZR, #0

Ejercicio 2:

a) Se aplica una técnica de loop unrolling en un programa para que duplique los elementos de un arreglo, enteros de 64 bits.

```
    ADD X0,XZR,XZR
        ADDI X1,XZR,#0x32
        ADD X2,XZR,XZR
L1: LDUR X8,[X0,#0x50]
        ADD X8,X8,X8
        STUR X8,[X0,#0x50]
        ADDI X0,X0,#0x8
        SUBI X1,X1,#0x1
        CBNZ X1,L1
RET
```

```
    ADD X0,XZR,XZR
        ADDI X1,XZR,#0x32
        ADD X2,XZR,XZR
L1: LDUR X8,[X0,#0x50]
        ADD X8,X8,X8
        STUR X8,[X0,#0x50]
        LDUR X8,[X0,#0x58]
        ADD X8,X8,X8
        STUR X8,[X0,#0x58]
        ADDI X0,X0,0x10
        SUBI X1,X1,#0x2
        CBNZ X1,L1
RET
```

	Ocupación de memoria	Instrucciones ejecutadas
Prog. Izq.	4 * 10 = 40 bytes = 0x26	304 (4 + 6*50)
Prog. Der	4 * 13 = 52 bytes = 0x34	229 (4 + 9 * 25)

b) Este es un fragmento de ProgIzq. Escribe 2 permutaciones equivalentes y 2 no equivalentes. Hay 5!=125 permutaciones posibles pero no todas hacen lo mismo que el código original.

	Equivalente 1	Equivalente 2	No equivalente 1	No equivalente 2
LDUR X8, [X0,#0x50]	SUBI X1,X1,#0x1 LDUR X8,	LDUR X8, [X0,#0x50]	LDUR X8, [X0,#0x50] STUR	STUR X8, [X0,#0x50]

ADD X8,X8,X8
STUR X8,
[X0,#0x50]
ADDI X0,X0,#0x8
SUBI X1,X1,#0x1
	[X0,#0x50]
	ADD X8,X8,X8
	STUR X8,
	[X0,#0x50]
	ADDI X0,X0,#0x8
		ADD X8,X8,X8
		SUBI X1,X1,#0x1
		STUR X8,
		[X0,#0x50]
		ADDI X0,X0,#0x8
			X8,[X0,#0x50]
			ADD X8,X8,X8
			SUBI X1,X1,#0x1
			ADDI X0,X0,#0x8
				ADD X8,X8,X8
				SUBI X1,X1,#0x1
				ADDI X0,X0,#0x8
				LDUR X8,
				[X0,#0x50]

Ejercicio 3:

Dado el siguiente programa en Assembler LEGv8 y el estado inicial de la memoria:

```
    MOVZ X0,#0x100,LSL#0
      LSL X1,X0,#1
      ORRI X2,X1,#0x100
loop: LDUR X3,[X0,#0]
        ADDI X3,X3,#1
        CBZ X3,end
        SUBI X3,X3,#1
        LSL X3,X3,#3
        ADD X3,X1,X3
        LDUR X3,[X3,#0]
        STUR X3,[X2,#0]
        ADDI X0,X0,#8
        ADDI X2,X2,#8
        B loop
end:
```

```
Dirección: contenido
...
0x100: 0x1
0x108: 0x2
0x110: 0x0
0x118: 0xFFFFFFFFFFFFFFFF
...
0x200: 0xCAFECAFE
0x208: 0xCOCACO1A
0x210: 0xDEADBEEF
...
0x300: 0x0
0x308: 0x0
0x310: 0x0
...
```

a) Mostrar el valor final de la memoria escribiendo al costado solo las posiciones que cambian.

```
    // LOOP 1
    MOVZ X0,#0x100,LSL#0   // x0 = 0x00000100
      LSL X1,X0,#1          //  x1 = 0x00000200
      ORRI X2,X1,#0x100     // x2  = 0x00000300
loop: LDUR X3,[X0,#0]     // x3 = mem[x0 + 0] = mem[0x00000100] = 0x1
        ADDI X3,X3,#1         // x3 = 1 + 1 = 2 = 0x2
        CBZ X3,end           // comparo (2, 0) no salto 2 != 0
        SUBI X3,X3,#1        // x3 = 2 - 1 = 1 = 0x1
        LSL X3,X3,#3         // x3 = 1 * 8 = 8 = 0x8
        ADD X3,X1,X3         // x3 = 0x208
        LDUR X3,[X3,#0]      // x3 = mem[x3 + 0] = mem[0x208] = 0xCOCACO1A
        STUR X3,[X2,#0]      // mem[x2 + 0] = x3 = mem[0x00000300] = 0x0
        ADDI X0,X0,#8        // x0 = 0x00000108
        ADDI X2,X2,#8        // x2 = 0x00000308
        B loop
end:
```

```
    // LOOP 2
    MOVZ X0,#0x100,LSL#0   // x0 = 0x00000100
      LSL X1,X0,#1          //  x1 = 0x00000200
      ORRI X2,X1,#0x100     // x2  = 0x00000300
loop: LDUR X3,[X0,#0]     // x3 = mem[x0 + 0] = mem[0x00000108] = 0x2
```

```
        ADDI X3,X3,#1         // x3 = 2 + 1 = 3 = 0x3
        CBZ X3,end           // comparo (3, 0) no salto 3 != 0
        SUBI X3,X3,#1        // x3 = 3 - 1 = 2 = 0x2
        LSL X3,X3,#3         // x3 = 2 * 8 = 16 = 0x10
        ADD X3,X1,X3         // x3 = 0x210
        LDUR X3,[X3,#0]      // x3 = mem[x3 + 0] = mem[0x210] = 0xDEADBEEF
        STUR X3,[X2,#0]      // mem[x2 + 0] = x3 = mem[0x00000308] = 0x0
        ADDI X0,X0,#8        // x0 = 0x00000110
        ADDI X2,X2,#8        // x2 = 0x00000310
        B loop
end:
```

```
    // LOOP 3
    MOVZ X0,#0x100,LSL#0   // x0 = 0x00000100
      LSL X1,X0,#1          //  x1 = 0x00000200
      ORRI X2,X1,#0x100     // x2  = 0x00000300
loop: LDUR X3,[X0,#0]     // x3 = mem[x0 + 0] = mem[0x00000110] = 0x0
        ADDI X3,X3,#1         // x3 = 0 + 1 = 1 = 0x1
        CBZ X3,end           // comparo (1, 0) no salto 1 != 0
        SUBI X3,X3,#1        // x3 = 1 - 1 = 0 = 0x0
        LSL X3,X3,#3         // x3 = 0 * 8 = 0 = 0x0
        ADD X3,X1,X3         // x3 = 0x200
        LDUR X3,[X3,#0]      // x3 = mem[x3 + 0] = mem[0x200] = 0xCAFECAFE
        STUR X3,[X2,#0]      // mem[x2 + 0] = x3 = mem[0x00000310] = 0x0
        ADDI X0,X0,#8        // x0 = 0x00000118
        ADDI X2,X2,#8        // x2 = 0x00000318
        B loop
end:
```

```
    // LOOP 4
    MOVZ X0,#0x100,LSL#0   // x0 = 0x00000100
      LSL X1,X0,#1          //  x1 = 0x00000200
      ORRI X2,X1,#0x100     // x2  = 0x00000300
loop: LDUR X3,[X0,#0]     //x3=mem[x0 + 0]=mem[0x00000118]=0xFFFFFFFFFFFFFFFF
        ADDI X3,X3,#1         // x3 = -1 + 1 = 0 = 0x0
        CBZ X3,end           // comparo (0, 0) salto000 0=0
        SUBI X3,X3,#1        // x3 = 1 - 1 = 0 = 0x0
        LSL X3,X3,#3         // x3 = 0 * 8 = 0 = 0x0
        ADD X3,X1,X3         // x3 = 0x200
        LDUR X3,[X3,#0]      // x3 = mem[x3 + 0] = mem[0x200] = 0xCAFECAFE
        STUR X3,[X2,#0]      // mem[x2 + 0] = x3 = mem[0x00000310] = 0x0
        ADDI X0,X0,#8        // x0 = 0x00000118
        ADDI X2,X2,#8        // x2 = 0x00000318
        B loop
end:                    // porfin se termino el bucle del orto
```

Entonces recapitulando estas son las modificaciones:

```
Dirección: contenido
...
0x100: 0x1
0x108: 0x2
0x110: 0x0
0x118: 0xFFFFFFFFFFFFFFFF
...
0x200: 0xCAFECAFE
0x208: 0xCOCACO1A
0x210: 0xDEADBEEF
...
0x300: 0x0  //OxCOCACO1A (se puede ver en loop 1)
0x308: 0x0  //0xDEADBEEF (se puede ver en loop 2)
0x310: 0x0  //0xCAFECAFE (se puede ver en loop 3)
...
```

b) Explicar en una línea que hace el código.

El código lo que hace es poner en la dirección de memoria 0x300 lo que hay en 0x200, lo que hay en 0x208 en 0x308 y lo que hay en 0x210 en 0x310. ?

Ejercicio 4:

Desensamblar el programa que se volcó directamente desde la memoria RAM, byte a byte.

08 00 45 F8 08 01 08 8B 08 00 05 F8 00 20 00 91 21 04 00 D1

Primero vamos a separar en los 4 grupos que necesitamos, ya que cada instrucción tiene 32 bits, y cada grupo de 1 byte tiene 8bits entonces 4 ∗ 8 = 32

```
- 08 00 45 F8
- 08 01 08 8B
- 08 00 05 F8
- 00 20 00 91
- 21 04 00 D1
```

Como las instrucciones están en little endian (en realidad no lo dice, pero en algunas aulas preguntaron y los profes dijeron que si, ademas si estarían en big endian no tendrían sentido). Por lo tanto las vamos a dar vuelta.

Y dadas vueltas las vamos a desensamblar normalmente como ya estamos acostumbrados.

- F8 45 00 08 = 1111 1000 0100 0101 0000 0000 0000 1000

  Los primeros 11 bits corresponden al opcode de la instrucción:

  1111 1000 010 = 0111 1100 0010 = 0x7C2, viendo en la green card observamos que se trata de LDUR, y ademas sabemos que es de tipo “d”:

opcode	DT_address	op	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits
11111000010	001010000	00	00000	01000

  Entonces tenemos:

  Rn = 00000 = 0 = x0

  Rt = 01000 = 8 = x8

DT_address = 001010000 = 80

Completando la instrucción tenemos: **LDUR x8, [x0 + #80]**

- 8B 08 01 08 = 1000 1011 0000 1000 0000 0001 0000 1000

  Los primeros 11 bits corresponden al opcode de la instrucción:

  1000 1011 000 = 0100 0101 1000 = 0x458, viendo en la green card observamos que se trata de ADD, y ademas sabemos que es de tipo “r”:

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits
10001011000	01000	000000	01000	01000

  Entonces tenemos:

  Rm = 01000 = 8 = x8

  shamt = 000000 = 0

  Rn = 01000 = 8 = x8

  Rd = 01000 = 8 = x8

  Completando la instrucción tenemos: **ADD x8, x8, x8**

- F8 05 00 08 = 1111 1000 0000 0101 0000 0000 0000 1000

  Los primeros 11 bits corresponden al opcode de la instrucción:

  1111 1000 000 = 0111 1100 0000 = 0x7C0, viendo en la green card observamos que se trata de STUR, y ademas sabemos que es de tipo “d”:

opcode	DT_address	op	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits
1111 1000 000	01010000	00	00000	01000

  Entonces tenemos:

  Rn = 00000 = 0 = x0

  Rt = 01000 = 8 = x8

  DT_address = 001010000 = 80

  Completando la instrucción tenemos: **STUR x8, [x0 + #80]**

- 91 00 20 00 = 1001 0001 0000 0000 0010 0000 0000 0000

  Los primeros 11 bits corresponden al opcode de la instrucción:

  1001 0001 000 = 0100 1000 1000 = 0x488, viendo en la green card observamos que se trata de ADDI, y ademas sabemos que es de tipo “i”:

opcode	ALU_immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits
1001000100	000000001000	00000	00000

  Entonces tenemos:

  Rn = 00000 = 0 = x0

  Rd = 00000 = 0 = x0

ALU_immediate = 000000001000 = 8

Completando la instrucción tenemos: ADDI x0, x0, #8

- D1 00 04 21 = 1101 0001 0000 0000 0000 0100 0010 0001

  Los primeros 11 bits corresponden al opcode de la instrucción:

  1101 0001 000 = 0110 1000 1000 = 0x688, viendo en la green card observamos que se trata de SUBI, y ademas sabemos que es de tipo “i”:

opcode	ALU_immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits
1101000100	000000000001	00001	00001

Entonces tenemos:

Rn = 00001 = 0x01 = 1 = x1

Rd = 00001 = 0x01 = 1 = x1

ALU_immediate = 000000000001= 1

Completando la instrucción tenemos: **SUBI x1, x1, #1**

Recapitulando las instrucciones son:

LDUR x8, [x0 + #80]
ADD x8, x8, x8
STUR x8, [x0 + #80]
ADDI x0, x0, #8
SUBI x1, x1, #1

Ejercicio 5:

Ensamblar estos dos programas:

	Ensamblador	Codigo maquina
Delay loop 1:	.org 0x2000 L0: SUBI X0,X0,#1 CBNZ X0,L0	# Ensambla en 0x2000 0x __ __ __ __ __ __ __ __ 0x __ __ __ __ __ __ __ __
Delay loop 2:	.org 0x4000 L1: SUBI X0,X0,#1 CBNZ X0,L1	# Ensambla en 0x4000 0x __ __ __ __ __ __ __ __ 0x __ __ __ __ __ __ __ __

En los programas de ensamblador ARMv8, la instrucción .org se utiliza para **especificar la dirección de memoria donde comienza el código o los datos**.

La función principal de la instrucción .org es establecer la dirección base para las siguientes instrucciones o declaraciones de datos. Esto significa que todas las direcciones de memoria utilizadas en el código a partir de ese punto se calcularán como ofsets relativos a la dirección base especificada por .org.

Entonces en el código de ejemplo, no se estaría usando, o no se modificaría en nada que cambie el valor del .org.

Asi que simplemente desensamblamos una instrucción y ya esta, es la misma para las dos:

```
L0: SUBI X0,X0,#1
    CBNZ X0,L0
```

SUBI x0, x0, #1, como es una instruccion de tipo “I” tenemos:

opcode	ALU_immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits
1101000100	000000000001	00000	00000

Entonces: 1101 0001 0000 0000 0000 0100 0000 0000 = **0xD1000400**

CBNZ X0,L0, como es de tipo “CB” tenemos:

opcode	COND_BR_address	Rt
7 bits	19 bits	5 bits
10110101	1111111111111111111	00000

Entonces: 1011 0101 1111 1111 1111 1111 1110 0000 = **0xB5FFFFE0**

Ejercicio 6:

Dada la compilación con la siguiente relación entre variables y registros:

- x ↔ x0
- score ↔ x1
- speedx ↔ x2

```
if (x==0)
    score++;
else
    speedx = -speedx
```

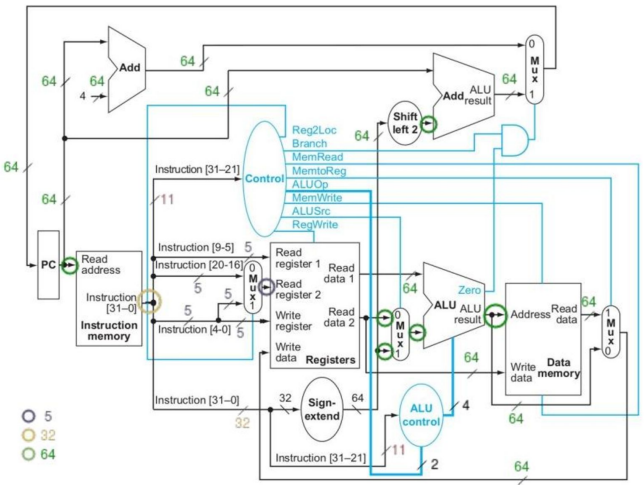
```
  CBZ X0,L1
    SUBI X8,XZR,#1
    ADD X2,X2,X8
    ADDI X2,X2,#1
L1: ADDI X1,X1,#1
```

Explicar en una línea si la compilación es correcta o no y por qué.

La compilación no es correcta, ya que en el caso que x0 ≠ 0, va a llegar a la ultima instrucción ADDI x2, x2, #1 y va a seguir con la otra que es L1: ADDI x1, x1, #1, por lo tanto en el caso que x0 ≠ 0 también se va a ejecutar score++ lo cual esta incorrecto.

Ejercicio 7:

Marcar el ancho en bits de las 15 líneas de datos marcadas.



Ejercicio 8:

Un opcode genera una instrucción no documentada que hace que Control tome estos valores

Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALU
0	0	1	1	1	0	0	ADD

Describir qué operación realiza: Se va a copiar el valor de la suma de dos registros en un nuevo registro, en memoria.

Invente su nemónico: Seria como un ADDLDUR :)