PRÁCTICO 6 - LEGv8 Básico

Ejercicio 1: Dadas las siguientes sentencias en “C”:

1) f = g + h + i + j;
2) f = g + (h + 5);
3) f = (g + h) + (g + h);

- Escribir la secuencia mínima de código assembler LEGv8 asumiendo que f, g, h, i y j se asignan en los registros X0, X1, X2, X3 y X4 respectivamente.
- Dar el valor de cada variable en cada instrucción assembler si f, g, h, i y j se inicializan con valores de 1, 2, 3, 4, 5, en base 10, respectivamente.

1.	Add X0, X1, X2	(f = g + h)	(f = 2 + 3)
	Add X9, X3, X4	(X9 = i + j)	(X9 = 4 + 5)
	Add X0, X0, X9	(f = g + h + i + j)	(f = 14)
2.	Addi X0, X2, #5	(f = h + 5)	(f = 3 + 5)
	Add X0, X1, X0	(f = g + h + 5)	(f = 10)
3.	Add X9, X1, X2	(f = g + h)	(f = 2 + 3)
	Add X0, X9, X9	(f = (g + h) + (g + h))	(f = 10)

Luego, dadas las siguientes sentencias en assembler LEGv8:

1. ADD X0, X1, X2
2. ADDI X0, X0, #1
   ADD X0, X1, X2

- Escribir la secuencia mínima de código “C” asumiendo que los registros X0, X1 y X2 contienen las variables f, g y h respectivamente.
- Dar el valor de cada variable en cada instrucción assembler si f, g y h se inicializan con valores de 1, 2, 3, en base 10, respectivamente.

1.	f = g + h	f = 2 + 3
2.	f = f + 1	f = 2 + 1
	f = g + h	f = 2 + 3

Ejercicio 2: Dadas las siguientes sentencias en “C”:

1. f = -g - f;
2. f = g + (-f - 5);

- Escribir la secuencia mínima de código assembler LEGv8 asumiendo que f y g se asignan en los registros X0 y X1 respectivamente.
- Dar el valor de cada variable en cada instrucción assembler si f y g se inicializan con valores de 4 y 5, en base 10, respectivamente.

1. Add X9, X1, X0	(X9 = g + f)	(X9 = 4 + 5)
Sub X0, XZR, X9	(f = -g - f)	(f = -9)
2. Addi X9, X0, #5	(X9 = f + 5)	(X9 = 4 + 5)
Sub X9, XZR, X9	(X9 = -f - 5)	(X9 = -4 - 5)
Add X0, X1, X9	(f = g + (-f - 5))	(f = - 4)

Ejercicio 3: Dadas las siguientes sentencias en assembler LEGv8:

1. SUB X1, XZR, X1	(g = - 2)
ADD X0, X1, X2	(f = -2 + 3)
2. ADDI X2, X0, #1	(h = 1 + 1)
SUB X0, X1, X2	(f = 2 - (1 + 1))

- Escribir la secuencia mínima de código “C” asumiendo que los registros X0, X1, y X2 contienen las variables f, g, y h respectivamente.
- Dar el valor de cada variable en cada instrucción assembler si f, g, y h se inicializan con valores de 1, 2 y 3, en base 10 respectivamente.

1. f = -g + h
2. f = g - (f + 1)

Ejercicio 4: Dadas las siguientes sentencias en “C”:

1. f = -g - A[4];
2. B[8] = A[i - j];

- Escribir la secuencia mínima de código assembler LEGv8 asumiendo que f, g, i y j se asignan en los registros X0, X1, X2 y X3 respectivamente, y que la dirección base de los arreglos A y B se almacenan en los registros X6 y X7 respectivamente.
- ¿Cuántos registros se utilizan para llevar a cabo las operaciones anteriores?

1.
```
Ldur X0, [X6, #32]      (f = A[4])
Add X0, X0, X1          (f = A[4] + g)
Sub X0, XZR, X0         (f = -g - A[4])
```

2.
```
Sub X9, X2, X3          (X9 = i - j)
Lsl X9, X9, #3          (X9 = (i - j) * 8)
Add X10, X6, X9         (X10 = &A + A[(i - j)*8])
Ldur X11, [X10, #0]     (X11 = A[i - j])
Stur X11, [X7, #64]     (B[8] = A[i - j])
```

Ejercicio 5: Dadas las siguientes sentencias en assembler LEGv8:

1.
```
LSL X2, X4, #1          h = 2 * j
ADD X0, X2, X4          f = (2 * j) + j
ADD X0, X0, X4          f = (2 * j) + j + j
```

2.
```
LSL X9, X3, #3          X9 = 8 * i
ADD X9, X6, X9          X9 = &A + (8 * i)
LSL X10, X4, #3         X10 = 8 * j
ADD X10, X7, X10        X10 = &B + (8 * j)
LDUR X12, [X9, #0]      X12 = A[i]
ADDI X11, X9, #8        X11 = A[i] + 8 = A[i + 1]
LDUR X9, [X11, #0]      X9 = A[i + 1]
ADD X9, X9, X12         X9 = A[i + 1] + A[i]
STUR X9, [X10, #0]      B[j] = A[i + 1] + A[i]
```

- Escribir la secuencia mínima de código “C” asumiendo que los registros X0, X1, X2, X3 y X4 contienen las variables f, g, h, i y j respectivamente, y los registros X6, X7 contienen las direcciones base de los arreglos A y B.
- Para las instrucciones LEGv8 anteriores, re-escriba el código para minimizar (de ser posible) la cantidad de instrucciones manteniendo la funcionalidad.

1. f = (2 * j) + j + j
2. B[j] = A[i + 1] + A[i]

1. Lsl X0, X4,#2

Ejercicio 6: Dadas las siguientes sentencias en assembler LEGv8:

ADDI X9, X6, #8	X9 = &A + 8 = &A[1]
ADD X10, X6, XZR	X10 = &A[0] + 0
STUR X10, [X9, #0]	A[1] = &A[0]
LDUR X9, [X9, #0]	X9 = A[1] = &A[0]
ADD X0, X9, X10	f = &A[0] + &A[0]

- Asumiendo que los registros X0, X6 contienen las variables f y A (dirección base del arreglo), escribir la secuencia mínima de código “C” que representa.
- Asumiendo que los registros X0, X6 contienen los valores 0xA, 0x100, y que la memoria contiene los valores de la tabla, encuentre el valor del registro X0 al finalizar el código assembler.

Dirección	Valor
0x100	0x64
0x108	0xC8
0x110	0x12C

1. f = &A[0] + &A[0]

Ejercicio 7: Dado el contenido de los siguientes registros:

1. X9 = 0x55555555, y X10 = 0x12345678
2. X9 = 0x00000000AAAAAAAA, y X10 = 0x1234567812345678

- ¿Cuál es el valor del registro X11 luego de la ejecución del siguiente código assembler en LEGv8?

  LSL X11, X9, #4

  ORR X11, X11, X10

1. X11 = 0x0000000555555550

   X11 =

0000 0000 0000 0000 0000 0000 0000 0101 0101 0101 0101 0101 0101 0101 0101 0000
0000 0000 0000 0000 0000 0000 0000 0000 0001 0010 0011 0100 0101 0110 0111 1000
0000 0000 0000 0000 0000 0000 0000 0000 0001 0010 0011 0100 0101 0110 0111 1000

   X11 = 0x0000000557755778

- ¿Cuál es el valor del registro X11 luego de la ejecución del siguiente código assembler en LEGv8?

  LSL X11, X10, #4 X11 = 0x0000000123456780

  ANDI X11, X11, #FFF X11 = 0x0000000000000780

0000 0000 0000 0000 0000 0000 0000 0001 0010 0011 0100 0101 0110 0111 1000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111 1000 0000
0 0 0 0 0 0 0 0 0 0 0 0 0 7 8 0

- ¿Cuál es el valor del registro X11 luego de la ejecución del siguiente código assembler en LEGv8?

  LSR X11, X9, #3 X11 = 0x00000000AAAAAAAA

  ANDI X11, X11, #0x555 X11 = 0x0000000000000000

0000 0000 0000 0000 0000 0000 0000 0101 0101 0101 0101 0101 0101 0101 0101 0101
0000 0000 0000 0000 0000 0000 0000 0000 1010 1010 1010 1010 1010 1010 1010 1010

0000 0000 0000 0000 0000 0000 0000 0000 1010 1010 1010 1010 1010 1010 1010 1010
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0101 0101 0101
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Ejercicio 8: Suponga que el registro X9 contiene el Exception Syndrome Register (ESR). Dé una secuencia mínima de instrucciones LEGv8 para poner en X10 el número que codifica la clase de excepción Exception Class (EC).

Lsr, X10, X9, #26

Ejercicio 9: Suponga que el registro X9 contiene un número entero representado en complemento a dos. Dé una secuencia mínima de operaciones a realizar para devolver en X10 un 1 si y sólo si el contenido de X9 es negativo.

Lsr X10, X9, #63
Andi X10, X10, #1

Ejercicio 10: Utilizar MOVZ, MOVK para cargar los registros:

1. {X0 = 0x1234000000000000}
2. {X1 = 0xBBB0000000000AAA}
3. {X2 = 0xA0A0B1B10000C2C2}
4. {X3 = 0x0123456789ABCDEF}

1. Movz X0, 0x1234, Lsl 48
2. Movz X1, 0x0AAA, Lsl 0
   Movk X1, 0XBBB0, Lsl 48
3. Movz X2, 0xC2C2, Lsl 0
   Movk X2, 0xB1B1, Lsl 32
   Movk X2, 0xA0A0, Lsl 48
4. Movz X3, 0xCDEF, Lsl 0
   Movk X3, 0x89AB, Lsl 16
   Movk X3, 0x4567, Lsl 32
   Movk X3, 0x0123, Lsl 48

PRÁCTICO 7 - LEGv8 Avanzado

1. Para estos dos programas con entrada y salida en X0, decir que función realizan
   a.
```
        SUBIS X0, X0, #0
        B.LT else
        B done
else:   SUB X0, XZR, X0
done:
```
   b.
```
        MOV X9, X0
        MOV X0, XZR
loop:   ADD X0, X0, X9
        SUBI X9, X9, #1
        CBNZ X9, loop
done:
```

a. El algoritmo del inciso “a”, calcula el valor absoluto de un número ubicado en el registro X0.
b. El algoritmo de este inciso calcula la sumatoria de x0.

Si inicialmente X0 = 10

Iteración	0*	1	2	3	4	5	6	7	8	9
Valor de X9	10	9	8	7	6	5	4	3	2	1
Valor de X0	0	10	19	27	34	40	45	49	52	55

2. Ejercicio 2: Dado el siguiente programa LEGv8, dar el valor final de X10, dado que inicialmente {X10=0x0000000000000001}.

```
        SUBIS XZR, X9, #0
        B.GE else
        B done
else:   ORRI X10, XZR, #2
done:
```

- Dado que inicialmente {X9=0x0000000000101000}.
- Dado que inicialmente {X9=0x8000000000001000}.

Con X9 = 0x0000000000101000 :

```
        SUBIS XZR, X9, #0       set flags = X9 - 0
        B.GE else               salto al “else” ya que X9 es mayor/igual a cero
        B done                  salto incondicional
else:   ORRI X10, XZR, #2       X10 = 0 ORRI 2
done:
```

Con X9 = 0x8000000000001000, interpretándose como signado (en Complemento a dos) es un número negativo puesto que su bit más significativo es 1. Luego, como es un salto de representación signada, entonces X9 es negativo, al restarle cero sigue siendo negativo, y no es cero. Finalmente, salta al done y X9 queda sin alterarse.

3. Ejercicio 3: Dado el siguiente programa “C” y la asignación i, j, k, N ↔ X0, X1, X2, X9, escribir el programa LEGv8 que lo implementa.

```
long i, j, k;
if (i == N || j == N) {
        ++k;
} else {
        ++i;
        ++j;
}
Luego:
          Subs XZR, X0, X9
          B.eq, add_k
          Subs XZR, X1, X9
          B.eq, add_k
          Addi X0, X0, #1
          Addi X1, X1, #1
          B end
add_k :   Addi X2, X2, #1
end :
```

4. Dados los siguientes programas LEGv8:

loop: ADDI X0, X0, #2
SUBI X1, X1, #1
CBNZ X1, loop
done:
	loop: SUBIS X1, X1, #0
	B.LE done
	SUBI X1, X1, #1
	ADDI X0, X0, #2
	B loop
	done:

1. Dar los valores finales de X0, teniendo en cuenta que inicialmente vale {X0=0, X1=10}. Dada la asignación a X0, X1 ↔ acc, i, escribir el programa “C” equivalente donde todas las variables son de tipo long.
2. Dado que inicialmente {X1=N} ¿Cuántas instrucciones LEGv8 se ejecutan?
3. Para el programa de la derecha. Si reemplazamos B.LE done por B.MI done ¿Cuál es el valor final de X0 suponiendo que inicialmente {X0=0}?

4. Dada la asignación a X0, X1 ↔ acc, i, escribir el programa “C” equivalente del punto “4.4”, donde todas las variables son de tipo long.

   Ambos algoritmos, comenzando con X0 = 0 Y X1 = 10, deja como resultado a, X0 = 20 y X1 = 0.

   N + 1 veces puesto que B.MI hace el salto dependiendo de si el resultado anterior es negativo. → X1 = -1, X0 = 22.

```
long i, acc;
while (i > 0)
{
        acc = acc + 2
        i = i - 1
}
```

5. Dados los siguientes programas en LEGv8 :

ADD X10, XZR, XZR
loop: LDUR X1, [X0,#0]
ADD X2, X2, X1
ADDI X0, X0, #8
ADDI X10, X10, #1
CMPI X10, #100
B.LT loop
	ADDI X10, XZR, #50
	loop: LDUR X1, [X0,#0]
	ADD X2, X2, X1
	LDUR X1, [X0,#8]
	ADD X2, X2, X1
	ADDI X0, X0, #16
	SUBI X10, X10, #1
	CBNZ X10, loop

- ¿Cuántas instrucciones LEGv8 ejecuta cada uno?
- Reescribir en “C” dada la asignación X10, X1, X2, X0 ↔ i, a, result, MemArray

En el algoritmo de la izquierda, el bucle itera 100 veces, por lo cual tenemos 5 * 100, ya que “B.LT loop” se realiza 99 veces debido a que en el último CMPI X10 ya es 100. Ahí tendríamos un total de 599, y sumamos la primera instrucción ADD X10, XZR, XZR y obtenemos un total de 600 instrucciones. Analizando análogamente el algoritmo de la derecha, este ejecuta un total de 1 + 349 = 350.

long j, i = 0;
while (i < 100)
{
a = MemArray[j]
	Análogo.

result = result + a;
j = j + 8;
i = i + 1
}	

6. Traducir el siguiente programa en “C” a ensamblador LEGv8 dada la asignación de variables a registros X0, X1, X2, X9 ↔ str, found, i, N. El número 48 se corresponde con el carácter ‘0’ en ASCII, por lo tanto el programa cuenta la cantidad de ‘0’s que aparecen en una cadena de caracteres de longitud N.

```
#define N (1<<10)
char *str;
long found, i;
for (found=0, i=0; i!=N; ++i)
        found += (str[i]==48);
```

```
        Ldr X0, =str            X0 = &str[0]
        Ldr X9, N               N = 16
        Add X1, XZR, XZR        found = 0
        Add X2, XZR, XZR        i = 0
for:    Cmp X2, X9              Comparo i y N
        B.eq end                Salto si son iguales (termina el for)
        Add X11, X0, X2         X11 = &str[0] + i
        Ldurb W12, [X11, #0]    W12 = str[i]
        Cmp W12, #48            Verifico si el byte que traje es un 0
        B.ne skip               Si son distintos no lo cuento
        Add X1, X1, #1          Si es un cero, found +=1
skip:   Add X2, X2, #1          i = i + 1
        B for
end:
```

7. Traducir el siguiente programa “C” a LEGv8. La asignación de variables a registros X0, X1, X2, X3, X9 ↔ A, s, i, j, N. Notar que en “C” los arreglos bidimensionales se representan en memoria usando un orden por filas, es decir &A[i][j] = A + 8*(i*N+j).

```
#define N (1<<10)
long A[N][N], s, i, j;
s = 0;
```

```
for (i=0; i<N; ++i)
        for (j=0; j<N; ++j)
                s += A[i][j];
Ldr X0, =A                      x0 =&A[0][0]
Ldr X9, N                       N = 3
Add X1, XZR, XZR                s = 0
Add X2, XZR, XZR                i = 0
Add X3, XZR, XZR                j = 0
oLoop: Cmp X2,X9                if(i == N)
        B.eq oEnd               goto oEnd;
        Add X3, XZR, XZR        j = 0
iLoop:  Cmp X3,X9               if (j == N)
        B.eq iEnd               goto iEnd;
        Mul X12, X2, X9         X12 = i * N
        Add X12, X12, X3        X12 = (i * N) + j
        Lsl x12, x12, #3        X12 = ((i * N ) + j) * 8
        Add X12, X12, X0        X12 = &A[0][0] + ((i * N ) + j) * 8
        Ldur X11, [X12,#0]      X11 = A[i][j]
        Add X1, X1, X11         s+ = A[i][j]
        Addi X3, X3, #1         j ++;
        B iLoop
iEnd:   Addi X2, X2, #1         i ++;
        B oLoop
oEnd:
```

8. Mostrar cómo se implementarían las siguientes pseudoinstrucciones con la mínima cantidad de instrucciones LEGv8, pudiendo usar el registro X9 para almacenar valores temporales.

Nemónico	Operación	Semántica
CMP	Comparación	FLAGS = R[Rn] - R[Rm]
CMPI	Comparación con offset	FLAGS = R[Rn] - ALUImm
MOV	Copia de valores entre reg	R[Rd] = R[Rn]
NOP	No operación	
NOT	Operación lógica bit a bit	R[Rd] = ~R[Rn]

- CMP :

  Subs Rd, Rn, Rm

  B.eq “label”
- CMPI :

  Subis Rd, Rn, #”offset”

  B.eq “label”
- MOV :

  Add Rd, Xzr, Rm
- Nop :

  B.“cond” skip

  skip :
- Not :

  Eori Rd, Rn, -1

9. Suponiendo que el microprocesador LEGv8 está configurado en modo little-endian, decir que valores toman los registros X0 a X7 al terminar este programa.

MOVZ X9, 0xCDEF, LSL 0	X9 = 0xFEDC000000000000
MOVK X9, 0x89AB, LSL 16	X9 = 0xFEDCBA9800000000
MOVK X9, 0x4567, LSL 32	X9 = 0xFEDCBA9876540000
MOVK X9, 0x0123, LSL 48	X9 = 0xFEDCBA9876543210
STUR X9, [XZR, #0]	&XZR = 0xFEDCBA9876543210
LDURB X0, [XZR, #0]	X0 = 0xFE00000000000000
LDURB X1, [XZR, #1]	X1 = 0xDC00000000000000
LDURB X2, [XZR, #2]	X2 = 0xBA00000000000000
LDURB X3, [XZR, #3]	X3 = 0x9800000000000000
LDURB X4, [XZR, #4]	X4 = 0x7600000000000000
LDURB X5, [XZR, #5]	X4 = 0x5400000000000000
LDURB X6, [XZR, #6]	X4 = 0x3200000000000000
LDURB X7, [XZR, #7]	X7 = 0x1000000000000000

PRÁCTICO 8 - ENSAMBLADO Y DESENSAMBLADO DE LEGv8

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.
    a. 00 0000 0000 0000 0000 0000 0001
    b. 10 0000 0000 0000 0000 0000 0000

Al estar en formato de complemento a2, su bit más significativo nos indica si el número es positivo o negativo, luego para extenderlo, debemos hacerlo con su respectivo signo:

a. 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
b. 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 0000 0000 0000 0000 0000 0000

2. Tenemos las siguientes instrucciones en assembler LEGv8:

ADDI X9, X9, #0

STUR X10, [X11,#32]

- ¿Qué formato (R, I, D, B, CB, IM) de instrucciones son?
- Ensamblar a código de máquina LEGv8, mostrando sus representaciones en binario y luego en hexadecimal.

La instrucción Addi tiene formato “I”, mientras que Stur tiene formato “D”.

Addi X9, X9, #0 = 1001 0001 0000 0000 0000 0001 0010 1001 = 0x91000129

Stur X10, [X11, #32] = 1111 1000 0000 0010 0000 0001 0110 1010 = 0xF802016A

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

1. op=0x658, Rm=13, Rn=15, Rd=17, shamt=0
2. op=0x7c2, Rn=12, Rt=3, const=0x4

1. Desglosamos el op = 0x858 y obtenemos en binario 011001011000 luego como el cero está a la izquierda, no afecta :
    - El op en binario corresponde a SUB x17, X15, X13
    - Rd = 10001
    - Rn = 01111
    - Rm = 01101
    - Es una instrucción de tipo R

    En binario : 1100 1011 0000 1101 0000 0001 1111 0001

2. Desglosamos el op = 0x7c2 y obtenemos en binario 011111000010 luego como el cero está a la izquierda, no afecta :
    - El op en binario corresponde a Ldur de tipo D
    - Rt = 00011
    - Rn = 01100
    - Offset = 000000100
    - Ldur x3, [x12, #4]

    En binario : 1111 1000 0100 0000 0100 0001 1000 0011

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

1. 1000 1011 0000 0000 0000 0000 0000 0000
2. 1101 0010 1011 1111 1111 1111 1110 0010

1. El opcode corresponde a a Add, luego tenemos que es una instrucción de tipo R:
    - Rm = 00000
    - Shamt = 000000
    - Rn = 00000
    - Rd = 00000

Finalmente tenemos Add X0, X0, X0, en hexadecimal se corresponde a : 0x8B000000

2. Análogo

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

La primer línea se corresponde a :

- 0x10010000: 0x8B010029 → 1000 1011 0000 0001 0000 0000 0010 1001
- El opcode indica que la instrucción es un Add de tipo R
- Rm = 00001 → x1
- Shamt = 000000
- Rn = 00001 → x1
- Rd = 01001 → x9

Luego la instrucción en 0x10010000 es Add x9, x1, x1

- Guarda en X9 el producto de 2 y x1

La segunda línea corresponde a :

- 0x10010004: 0x8B010121 → 1000 1011 0000 0001 0000 0001 0010 0001
- El opcode indica que la instrucción es un Add de tipo R
- Rm = 00001 → x1
- Shamt = 00000
- Rn = 010001 → x9
- Rd = 00001 → x1

Luego la instrucción en 0x10010004 es Add x1, x9, x1

- Guarda en x1 la suma de x9 (2*x1) y x1.

Finalmente se puede deducir que este conjunto de instrucciones multiplica por 3 el valor de x1.

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.

- LSL XZR, XZR, 0 → Si se puede codificar, de hecho no modifica nada.
- ADDI X1, X2, -1 → No se puede codificar, no puede ir un offset signado.
- ADDI X1, X2, 4096 → Si se puede codificar, el offset llega hasta 4095.
- EOR X32, X31, X30 → No se puede codificar, no existe el X32 como registro.
- ORRI X24, X24, 0x1FFF → No se puede codificar, puesto que el offset se pasa (debe ser de 12 bits).
- STUR X9, [XZR,#-129] → Se puede codificar, pero hay un problema de desalineamiento de memoria.
- LDURB XZR, [XZR,#-1] → Se puede codificar.
- LSR X16, X17, #68 → No se puede codificar puesto que nos excedemos en el shamt.
- MOVZ X0, 0x1010, LSL #12 → No puede codificar, el offset solo es 0, 16, 32 o 48.
- MOVZ XZR, 0xFFFF, LSL #48 → Se puede codificar.

7. Ensamblar estos delay loops.

MOVZ X0, 0x1, LSL #48	MOVZ X0, 0xFFFF, LSL #32	MOVZ X0, 0x2, LSL #16

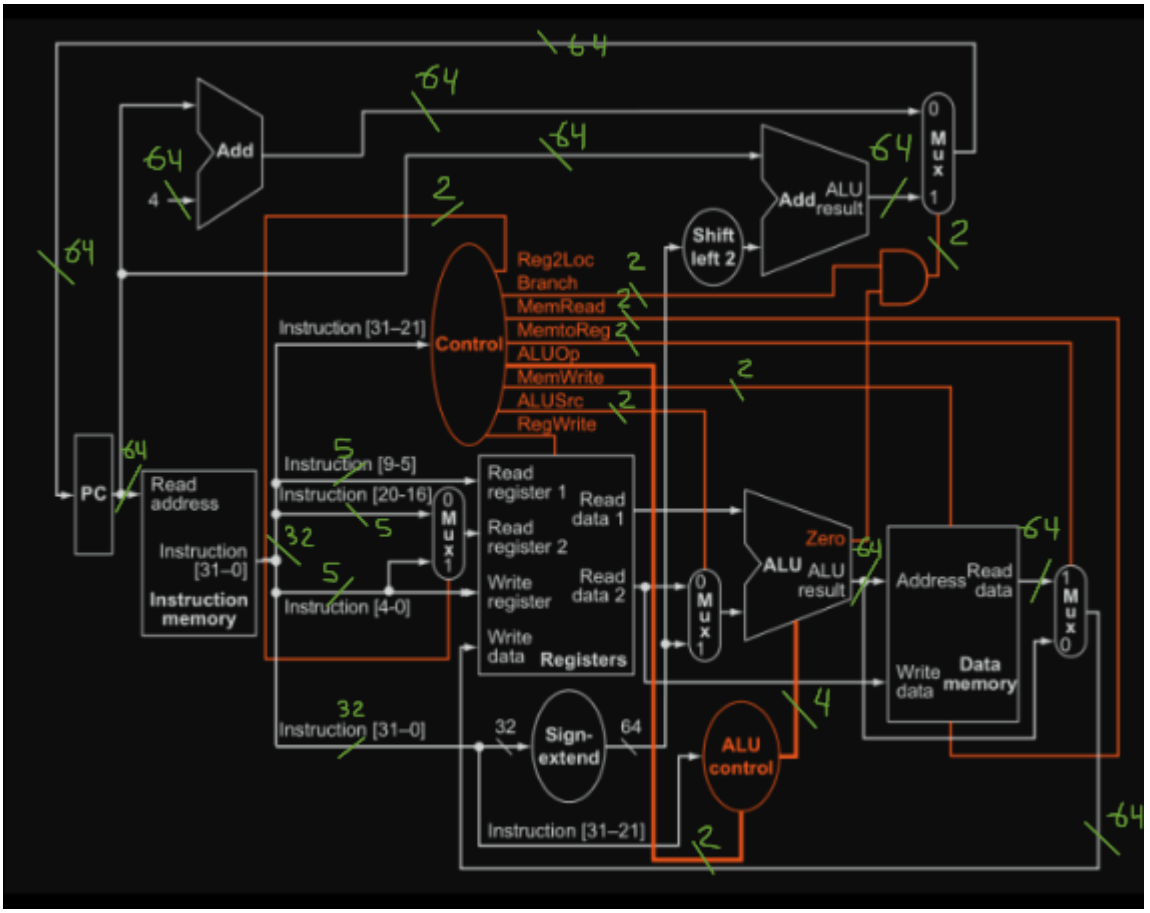
L1: SUBI X0,X0,#1 CBNZ X0, L1	L1: SUBIS X0,X0,#1 B.NE L1	L1: SUBIS XZR,X0,#0 B.EQ EXIT SUBI X0,X0,#1 B L1 EXIT:

0110 1001 0111 0000 0000 0000 0010 0000
L1 : 0110 1000 1000 0000 0000 0100 0000 0000
1011 0101 1111 1111 1111 1111 1110 0000

Análogos los demás, recordar que el Branch toma un inmediato signado, y significa cuantas instrucciones avanza hacia adelante o hacia atrás.

PRÁCTICO 9 - Implementación de la ISA

1. Marque con una barra invertida e indique el número de bits que representa cada una de las líneas del data path.



2. Considerando la siguiente distribución de instrucciones en un programa:

R-type	I-type	Ldur	Stur	Cbz	B
24	28	25	10	11	2

- ¿Qué porcentaje de todas las instrucciones utiliza data memory?
- ¿Qué porcentaje de todas las instrucciones utiliza instruction memory?
- ¿Qué porcentaje de todas las instrucciones utiliza el sign extend?

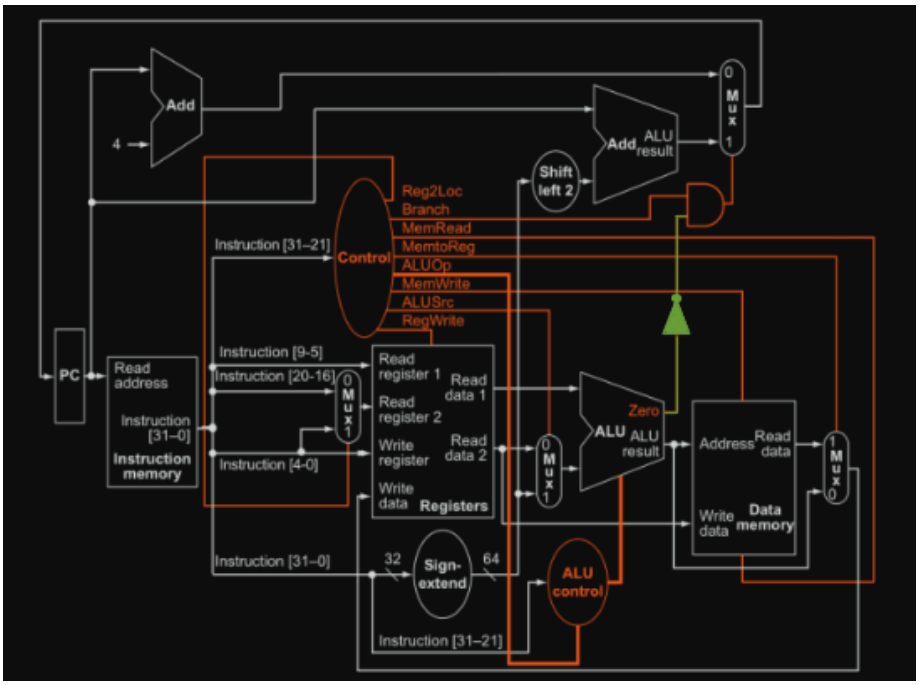
- Las instrucciones que acceden a memoria de datos son Ldur y Stur.Por lo tanto, el 35% de las instrucciones utilizan el recurso data memory.
- Todas las instrucciones deben acceder a la memoria de instrucción
- Los tipos de instrucción que utilizan el bloque de extensión de signo son tipo I, CB, B, D. Por lo tanto, el 76% de las instrucciones utilizan el recurso sign extend.

3. Complete la tabla con el estado de las señales sin mirar el libro. Indique con X las condiciones no-importa.

Instr	Reg2loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	AluOp1	AluOp0
R-type	0	0	0	1	0	0	0	1	0
Ldur	x	1	1	1	1	0	0	0	0
Stur	1	1	x	0	0	1	0	0	0
Cbz	1	0	x	0	0	0	1	0	1

4. Cuando se fabrican los chips de silicio, defectos en los materiales y errores en la fabricación pueden generar circuitos defectuosos. Un defecto común es que un cable de señal se rompa y siempre registre un '0' lógico. Esto se conoce comúnmente como "stuck-at-0 fault".
    - ¿Qué instrucciones operarían de forma incorrecta si el cable MemToReg está atascado en '0'? → Ldur
    - ¿Qué instrucciones operarían de forma incorrecta si el cable ALUSrc está atascado en '0'? → Ldur, Stur
    - ¿Qué instrucciones operarían de forma incorrecta si el cable Reg2Loc está atascado en '0'? → Stur, Cbz

5. Agregando una compuerta en diagrama del data path & control, cambie la implementación de la instrucción CBZ a CBNZ.

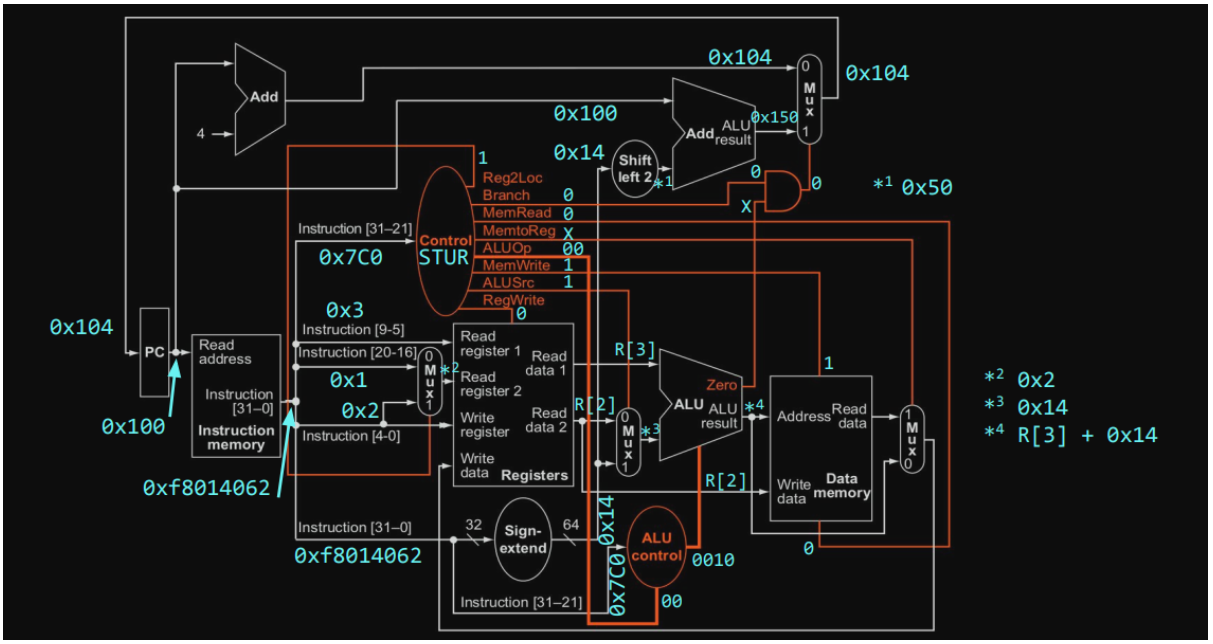


6. En este ejercicio analizaremos en detalle cómo se ejecuta una instrucción en el single-cycle datapath, asumiendo que la palabra de instrucción que ejecuta el procesador es: 0xf8014062, dado que PC = 0x100.
    - ¿Cuáles son las salidas de los bloques Sign-extend y Shift left 2 para esta palabra de instrucción?
    - ¿Cuáles son los valores de entrada a la unidad ALU control para esta palabra de instrucción?
    - ¿Cuál es la nueva dirección en el PC después de ejecutar esta instrucción?
    - Mostrar los valores de las entradas y salidas de cada Mux durante la ejecución de esta instrucción. Para los valores que son salidas de Registers, utilizar “Reg [Xn]”. ¿Cuáles son los valores de entrada de la ALU y las dos unidades Add?
    - ¿Cuáles son los valores de todas las entradas del bloque Registers?

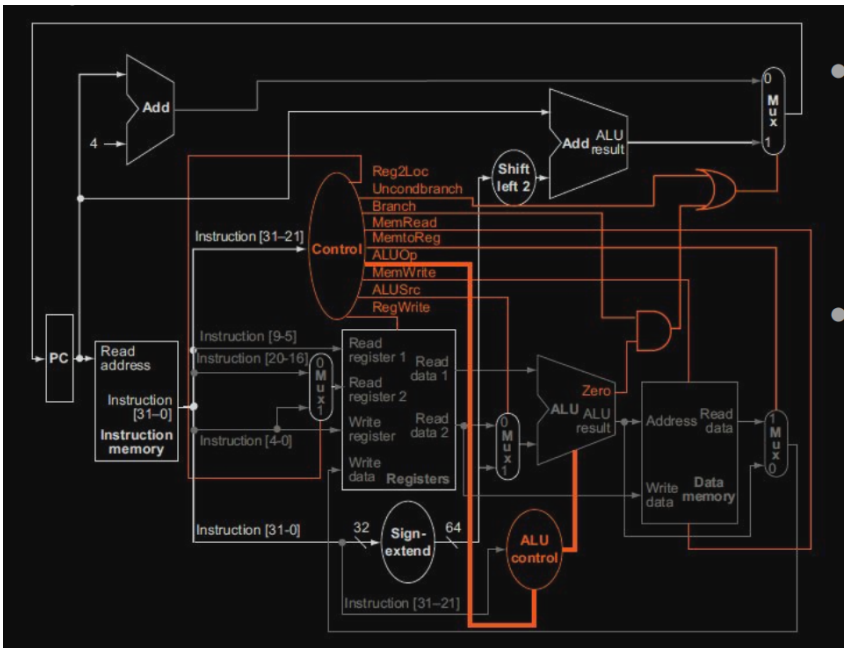
La instrucción en binario es : 1111 1000 0000 0001 0100 0000 0110 0010. Luego tomamos sus 11 bits más significativos y obtenemos 0x7C0 que se corresponde al opcode de STUR. Es de tipo D asi que obtenemos los siguientes campos :

- opcode → 11 bits
- DT address → 9 bits
- op → 2 bits
- Rn → 5 bits
- Rt → 5 bits

- Lo que tendríamos en sign extend es … 0000 0000 0000 0000 0001 0100, luego en hexa tendríamos 0x0000000000000014.
- En Shift left tendríamos el número multiplicado por 4 para que el PC pueda trabajar correctamente con la memoria, osea : .. 0000 0000 0000 0000 0101 0000 → 0x0000000000000050
- La entrada a la unidad ALU control es 00, debido a que es una instrucción Stur.
- La nueva dirección del PC después de ejecutar esta instrucción es 0x100 + 0x004 → 0x104



7. Agregar a la implementación de la ISA la instrucción de salto incondicional B, a partir de una nueva señal que sale de Control, denominada UncondBranch.



8. Suponiendo que los diferentes bloques dentro del procesador tienen las siguientes latencias:

I-mem/ D-mem	Register File	Mux	Alu	Adder	Single gate	Register read	Register Setup	Sign Extend	Control	S|2
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps	25 ps

- ¿Cuál es la latencia si lo único que tuviera que hacer el procesador es fetch de instrucciones consecutivas?
- Hay un modo alternativo de generar la señal Reg2Loc fácilmente de la instrucción sin tener que esperar la latencia de Control. Explique cómo sería. 8.3) ¿Cuál es la latencia si solo hubiera instrucciones de tipo R?
- ¿Cuál es la latencia para LDUR?
- ¿Cuál es la latencia para STUR?
- ¿Cuál es la latencia para CBZ?
- ¿Cuál es el mínimo periodo de reloj para esta implementación de la ISA? Indicar también la frecuencia de reloj correspondiente (f = 1/t).

- Latencia de fetch = RegRead + Imem = 30+250 = 280ps
- Latencia de tipo R = 30 + 250 + 50 + 25 + 150 + 25 + 200 + 25 + 20 = 775ps
- Latencia de Ldur = 30 + 250 + 150 + 200 + 250 + 25 + 20 = 925ps
- Latencia de Stur = 30 + 250 + 50 + 25 + 150 + 200 + 250 = 955ps
- Latencia CBZ = 30 + 250 + 50 + 25 + 150 + 25 + 200 + 5 + 25 + 20 = 780ps
- Para que todas las instrucciones puedan ejecutarse completamente el periodo de reloj debe ser mayor o igual a la latencia de instrucción más grande. En este caso, corresponde a la latencia de STUR. Mínimo periodo de reloj para esta implementación de la ISA (t) = 955ps = 955e-12 s
- Se calcula la frecuencia como la inversa del periodo. Máxima frecuencia de reloj (f) = 1/t ≈ 1.047 GHz.

```
                MOVZ X0, 0X100, LSL 0
                LSL X1, X0, 1
                ORRI X2, X1, 0X100          0X100: 0X1
LOOP :          LDUR X3, [X0, #0]           0X108: 0X2
                ADDI X3, X3, 1              0X110: 0X0
                CBZ X3. END                 0X118: 0XFFFFFFFFFFFFFFFF
                SUBI X3, X3, 1              …
                LSL X3, X3, 3               0X200: 0XCAFECAFE
                ADD X3, X1, X3              0X208: 0XC0CAC0LA
                LDUR X3, [X3, #0]           0X210: 0XDEADBEEF
                STUR X3, [X2, #0]           …
                ADDI X0, X0, 8              0X300: 0XC0CAC0LA
                ADDI X2, X2, 8              0X308: DEADBEEF
                B LOOP                      0X310: 0XCAFECAFE
```

```
x0 = 0x100

x1 = 0x200
x2 = 0x300

x3 = 0x1
x3 = 0x2
x3 = 0x1
x3 = 0x8
x3 = 0x208
x3 = 0XC0CAC0LA
0x300 = 0XC0CAC0LA

x0 = 0x108
x2 = 0x308

x3 = 0X2
x3 = 0x3
x3 = 0x2
x3 = 0x10
x3 = 0x210
x3 = DEADBEEF
0x308 = DEADBEEF

x0 = 0x110
x2 = 0x310

x3 =  0X0
x3 = 0x1
x3 = 0x0
x3 = 0x0
```

```
x0 = 0x118
x2 = 0x318
x3 = 0XFFFFFFFFFFFFFFFF
x3 = 0
```

```
x3 = 0x200
x3 = 0XCAFECAFE
0x310 = 0XCAFECAFE

x0 = 0x100
x1 = 0x200
x2 = 0x300
x7 = 0x1
x7 = 0x2
x7 = 0x1
x7 = 0xFEEDCAFE
0x300 = 0xFEEDCAFE
x0 = 0x108
x1 = 0x208
x2 = 0x308

x7 = 0x0
x7 = 0x1
x7 = 0x0
x0 = 0x110
x1 = 0x210
x2 = 0x310
x7 = 0x1
x7 = 0x2
x7 = 0x1
x7 = 0xDABAABAD
0x310 = 0xDABAABAD

x0 = 0x118
x1 = 0x218
x2 = 0x318

x7 = 0xFFFFFFFFFFF
x7 = 0x0
```