# Ensamblado y desensamblado de LEGv8

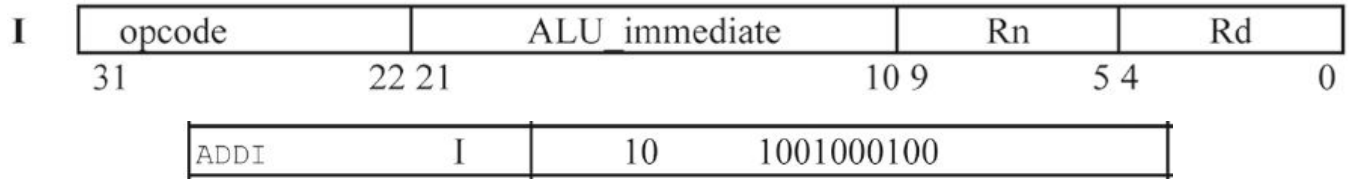
## Algunos ejercicios resueltos

OdC - 2025

# Ejercicio 2 - ADDI X9, X9, #0

| NAME, MNEMONIC | | FOR-MAT | OPCODE (9) (Hex) | OPERATION (in Verilog) |
|---|---|---|---|---|
| ADD | ADD | R | 458 | R[Rd] = R[Rn] + R[Rm] |
| ADD Immediate | ADDI | I | 488-489 | R[Rd] = R[Rn] + ALUImm |

2.1) Formato de instrucción: tipo I




2.2)
- opcode (10 bits): 1001000100
- ALU_immediate (12 bits): 000000000000
- Rn (5 bits) = X9: 01001
- Rd (5 bits) = X9: 01001

# Ejercicio 2 - ADDI X9, X9, #0

2.2)
- opcode (10 bits): 1001000100
- ALU_immediate (12 bits): 000000000000
- Rn  (5 bits) = X9: 01001
- Rd  (5 bits) = X9: 01001

Instrucción ensamblada en binario:

0b **1001 0001 0000 0000 0000 0001 0010 1001**

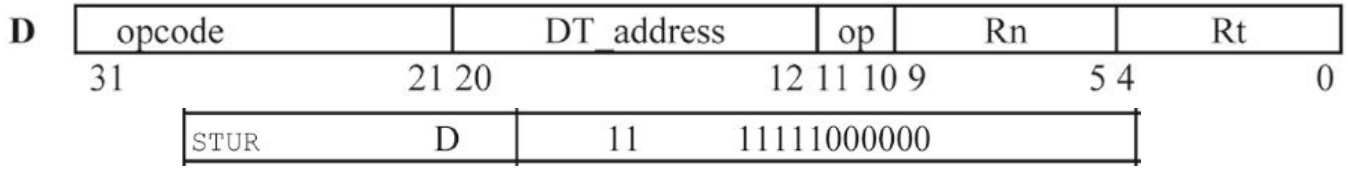
Instrucción ensamblada en hexadecimal:

0x **91000129**

# Ejercicio 2 - STUR X10, [X11,#32]

| | | | | |
|---|---|---|---|---|
| STore Register Unscaled offset | STUR | D | 7C0 | M[R[Rn] + DTAddr] = R[Rt] |

2.1) Formato de instrucción: tipo D




2.2)
- opcode (11 bits): 11111000000
- DT_address (9 bits): 000100000
- op (2 bits): 00
- Rn (5 bits) = X11: 01011
- Rt (5 bits) = X10: 01010

# Ejercicio 2 - STUR X10, [X11,#32]

2.2)
- opcode (11 bits): 11111000000
- DT_address (9 bits): 000100000
- op (2 bits): 00
- Rn  (5 bits) = X11: 01011
- Rt  (5 bits) = X10: 01010

Instrucción ensamblada en binario:

0b **1111 1000 0000 0010 0000 0001 0110 1010**

Instrucción ensamblada en hexadecimal:

0x **F802016A**

# Ejercicio 3.1

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

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

| | | | | |
|---|---|---|---|---|
| SUB | R | 11 | 11001011000 | 658 |

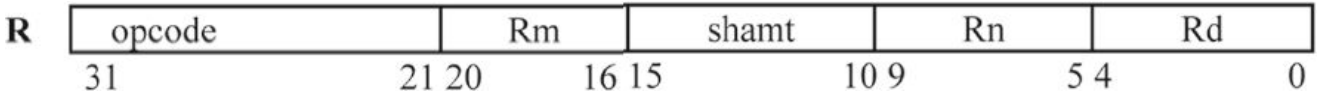
1) Con el opcode identificamos la instrucción: SUB y el **tipo: R**.

| | | | | |
|---|---|---|---|---|
| SUBtract | SUB | R | 658 | R[Rd] = R[Rn] – R[Rm] |

2) Instrucción en assembler: **SUB X17, X15, X13**

# Ejercicio 3.1

3) Sabiendo el tipo de instrucción identificamos el orden de los campos.




4)
- opcode (11 bits) = 0x658 = 0b11001011000
- Rm (5 bits) = 13 = 0b01101
- shamt (6 bits) = 0 = 0b000000
- Rn (5 bits) = 15 = 0b01111
- Rd (5 bits) = 17 = 0b10001

5) Representación binaria de la instrucción:

0b **1100 1011 0000 1101 0000 0001 1111 0001**

# Ejercicio 4.2

Dado el número en binario: 1101 0010 1011 1111 1111 1111 1110 0010

a) Transformar de binario a hexadecimal.

b) ¿Qué instrucciones LEGv8 representan en memoria?

a) 0x **D2BFFFE2**

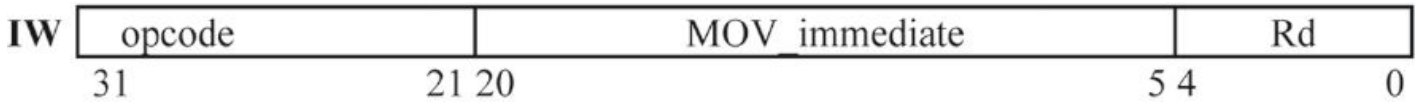
b) 1101 0010 1011 1111 1111 1111 1110 0010

- Campo de opcode de 11 bits = 11010010101 = 0x695

| | | | | | |
|---|---|---|---|---|---|
| MOVZ | IM | 9 | 110100101 | 694 | 697 |

- Opcode de la instrucción MOVZ = 110100101

# Ejercicio 4.2




1101 0010 1011 1111 1111 1111 1110 0010

- opcode: 110100101 -> MOVZ
- LSL: 01 -> LSL 16
- MOV_immediate: 1111111111111111 -> 0xFFFF
- Rd: 00010 -> X2

| bit 22 | bit 21 | LSL |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 16 |
| 1 | 0 | 32 |
| 1 | 1 | 48 |

Instrucción en assembler: **MOVZ X2, #0xFFFF, LSL 16**

# Ejercicio 5

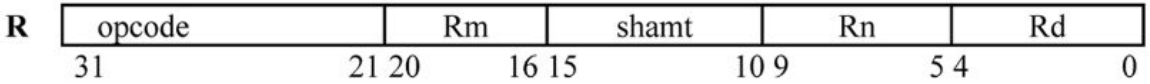
Ejecutar el siguiente código assembler que está en memoria para dar el valor final del registro X1. El contenido de la memoria se dá como una lista de pares, dirección de memoria: contenido, suponiendo alineamiento de memoria del tipo big endian. Describa sintéticamente que hace el programa.

0x10010000: 0x8B010029

0x10010004: 0x8B010121

```
0x10010000: 0x8B
0x10010001: 0x01
0x10010002: 0x00
0x10010003: 0x29

0x10010004: 0x8B
0x10010005: 0x01
0x10010006: 0x01
0x10010007: 0x21
```




# Ejercicio 5

0x10010000: 0x8B010029 -> **10001011000** 00001 000000 00001 01001
op: **10001011000** -> 0x458 -> ADD
Rm: 00 001 -> X1
sh: 000000 -> 0
Rn: 00 001 -> X1
Rd: 01 001 -> X9
**ADD X9, X1, X1**

0x10010004: 0x8B010121 -> **10001011000** 00001 000000 01001 00001
op: **10001011000** -> 0x458 -> ADD
Rm: 00 001 -> X1
sh: 000000 -> 0
Rn: 01 001 -> X9
Rd: 00 001 -> X1
**ADD X1, X9, X1**

**Resultado final: 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 (si)
2. ADDI X1, X2, -1 (no -1)
3. ADDI X1, X2, 4096 (no 4095)
4. EOR X32, X31, X30 (no X32)
5. ORRI X24, X24, 0x1FFF (no 0x1FFF)
6. STUR X9, [XZR,#-129] (si)
7. LDURB XZR, [XZR,#-1] (si)
8. LSR X16, X17, #68 (no 68)
9. MOVZ X0, 0x1010, LSL #12 (no 12)
10. MOVZ XZR, 0xFFFF, LSL #48 (si)

# Ejercicio 7-a

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

Usamos el main.list a modo demostrativo, pero el proceso de ensamblado es el que vimos en clases

De **main.list**:

```
Disassembly of section .text:

0000000000000000 <L1-0x4>:
   0:    d2e00020      mov  x0, #0x1000000000000

0000000000000004 <L1>:
   4:    d1000400      sub  x0, x0, #0x1
   8:    b5ffffe0      cbnz x0, 4 <L1>
```

| opcode | n° instrucciones | Rt |
|---|---|---|
| 10110101 | 1111111111111111111 ($-1_{10}$) | 00000 |
| 8 bits | 19 bits | 5 bits |

0b10110101111111111111111111100000 = 0xb5ffffe0

# Ejercicio 7-b

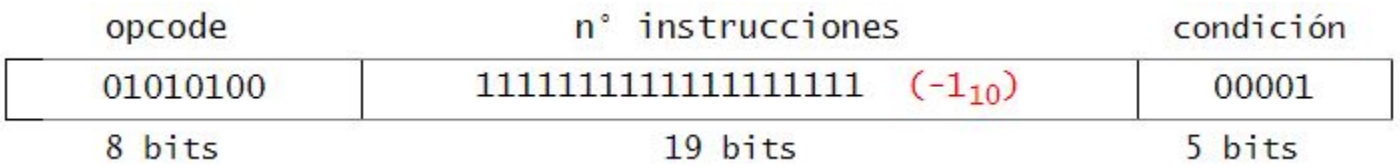
```
    MOVZ X0, 0xFFFF, LSL #32
L1: SUBIS X0,X0,#1
    B.NE L1
```

De **main.list**:

```
Disassembly of section .text:

0000000000000000 <L1-0x4>:
   0:    d2dfffe0      mov  x0, #0xffff00000000

0000000000000004 <L1>:
   4:    f1000400      subs x0, x0, #0x1
   8:    54ffffe1      b.ne 4 <L1>
```




0b01010100111111111111111111100001 = 0x54ffffe1

# Ejercicio 7-c

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

De **main.list**:

```
Disassembly of section .text:

0000000000000000 <L1-0x4>:
   0:    d2a00040      mov  x0, #0x20000

0000000000000004 <L1>:
   4:    f100001f      cmp  x0, #0x0
   8:    54000060      b.eq 14 <EXIT>
   c:    d1000400      sub  x0, x0, #0x1
  10:    17fffffd      b    4 <L1>
  14:    <EXIT>
```

# Ejercicio 7-c

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

```
8:    54000060      b.eq 14 <EXIT>
```

| opcode | n° instrucciones | condición |
|---|---|---|
| 01010100 | 0000000000000000011 ($3_{10}$) | 00000 |
| 8 bits | 19 bits | 5 bits |

0b01010100000000000000000001100000 = 0x54000060

```
10:   17fffffd      b    4 <L1>
```

| opcode | n° instrucciones |
|---|---|
| 000101 | 11111111111111111111111101 ($-3_{10}$) |
| 6 bits | 26 bits |

0b00010111111111111111111111111101 = 0x17fffffd

# Ejercicio 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?

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

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).

# Ejercicio 8.2

- Campo de inmediato de branch = 26 bits.

(3) BranchAddr = { 36{BR_address [25]}, BR_address, 2'b0 }

- Máxima cantidad de instrucciones "hacia adelante" = $2^{25}$-1 = 0x1FF FFFF
- Máxima cantidad de posiciones de memoria "hacia adelante" = 0x7FF FFFC

Branch B B 0A0-0BF PC = PC + BranchAddr (3,9)

- Partiendo de PC = 0x00000600 se alcanza la dirección:

PC = 0x0000 0600 + 0x7FF FFFC = **0x0800 05FC**

➔ 0x0001 4000 es posible llegar con una sola instrucción
➔ 0x0011 4524 es posible llegar con una sola instrucción
➔ 0x0F00 0200 NO es posible llegar con una sola instrucción

# Ejercicio 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).

```
MOVZ X0, #0x4000, LSL 16
BR X0
```

- Si el PC es distinto de `0x0` la respuesta es la misma!!
- Notar que BR se ensambla como instrucción tipo R (ver erratas).

**Branching Far Away**

Given a branch on register X19 being equal to register zero,

```
CBZ    X19, L1
```

replace it by a pair of instructions that offers a much greater branching distance. These instructions replace the short-address conditional branch:

```
       CBNZ   X19, L2
       B      L1
L2:
```

# Ejercicio 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 `0xFFFF FFFC`, ¿Cuántas instrucciones B son necesarias? (no se puede usar BR).

Máxima cantidad de posiciones de memoria "hacia adelante" de B = `0x7FF FFFC`

1) Partiendo de PC = `0x0` se alcanza la dirección = `0x7FF FFFC`
2) `PC = 0x7FF FFFC + 0x7FF FFFC = 0xFFF FFF8`
3) `PC = 0xFFF FFF8 + 0x7FF FFFC = 17FF FFF4`
4) …

Cantidad de instrucciones B = `0xFFFF FFFC / 0x7FF FFFC`
`= 4294967292 / 134217724 = 32,00000092 =` **33 instrucciones**

# Ejercicio 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]
```

# Ejercicio 10 - Self-modifying code

```
 0: MOVZ X0, 0x0400, LSL #0    // X0 = 0x0000 0000 0000 0400
 4: MOVK X0, 0x9100, LSL #16   // X0 = 0x0000 0000 9100 0400
 8: STURW X0, [XZR,#12]        // Sobre-escribe la siguiente instrucción con (*)
12: STURW X0, [XZR,#12]
```

(*) - 0x91000400 = 0b10010001000000000000010000000000

0b10010001000 = 0x488 = ADDI, Tipo I: Rd = Rn + Inmediato

- 0x91000400 = 0b 1001000100 000000000001 00000 00000

Opcode= 1001000100 (10 bits) = ADDI,

Inmediato= 000000000001 (12 bits), Rn= 00000 = X0, Rd= 00000 = X0

```
12: ADDI X0, X0, #1   // X0 = 0x0000000091000401
```