

Ejercicios

32-bit instruction format

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
R	func							rs2					rs1					func			rd					opcode											
I	immediate												rs1					func			rd					opcode											
SB	immediate							rs2					rs1					func			immediate					opcode											
UJ	immediate																									rd					opcode						

Ejercicio 1:

00: 00700293

- **Binario:** 0000 0000 0111 | 00000 | 000 | 00101 | 0010011.
- **Operación:** addi 00101(t0), 00000(zero), 0x007(t2).

04: 00100313

- **Binario:** 0000 0000 0001 | 00000 | 000 | 00110 | 0010011
- **Operación:** addi 00110(t1), 00000(zero), 0x001(ra).

08: 0062f333

- **Binario:** 0000 000 | 00110 | 00101 | 111 | 00110 | 0110011
- **Operación:** and 00110(t1), 00000(t0), 0x001(t1).

0c: 00030463

- **Binario:** 0000 000 | 00000 | 00110 | 000 | 01000 | 1100011
- **Operación:** beq 00110(t1), 00000(zero), 0x008(S0).

10: fff28293

- **Binario:** 1111 1111 1111 | 00101 | 000 | 00101 | 0010011
- **Operación:** addi 00101(t0), 00101(t0), 0xfff(-1).

14: 4012d293

- **Binario:** 0100 0000 0001 0010 1101 0010 1001 0011
- **Operación:** srai 00101(t0), 00101 (t0), 0x401

Ejercicio 2:

```
li a0,4228
li a1,2114
jal ra, resta
fin:
    beq zero, zero, fin
resta:
prologo:
    addi sp, sp, -4
    sw ra, 0(sp)
    sub a0, a0, a1
    beq a0, zero, epilogo
sigo:
    jal ra, resta
epilogo:
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

A. Indicar en qué posiciones de memoria se encuentra cada etiqueta.

li a0, 4228 (0x0 a 0x4 y 0x4 a 0x8): Como 4228 es mucho más grande que 4096 (más que 12 bits el inmediato), realiza un LUI y un addi, por lo tanto son dos posiciones de memoria diferentes. Como es una pseudoinstrucción, realiza dos instrucciones

li a1, 2114 (0x8 a 0xC y 0xC a 0x10): Como es una pseudoinstrucción, realiza dos instrucciones

jal ra, resta: (0x10)

beq zero, zero, fin: (0x14)

addi sp, sp, -4: (0x18)

sw ra, 0(sp): (0x1C)

sub a0, a0, a1: (0x20)

beq a0, zero, epilogo: (0x24)

jal ra, resta: (0x28)

lw ra, 0(sp): (0x2C)

addi sp, sp, 4: (0x30)

ret: (0x34)

```

0:      00001537      lui x10 0x1
4:      08450513      addi x10 x10 132
8:      000015b7      lui x11 0x1
c:      84258593      addi x11 x11 -1982
10:     008000ef      jal x1 8 <resta>

)0000014 <fin>:
14:     00000063      beq x0 x0 0 <fin>

)0000018 <resta>:
18:     ffc10113      addi x2 x2 -4
1c:     00112023      sw x1 0 x2
20:     40b50533      sub x10 x10 x11
24:     00050463      beq x10 x0 8 <epilogo>

)0000028 <sigo>:
28:     ff1ff0ef      jal x1 -16 <resta>

)000002c <epilogo>:
2c:     00012083      lw x1 0 x2
30:     00410113      addi x2 x2 4
34:     00008067      jalr x0 x1 0

```

B. Indicar el desplazamiento de las llamadas a etiquetas.

Nota: 0x10 extiende con 0's hasta 32 bits.

Llega a jal que está en 0x10, salta a la diferencia con resta.

→ resta: 0x18 hasta 0x24 (guarda).

→sigo: 0x28(false) → resta: 0x18

→ epilogo: 0x2C(true) → fin: 0x14

Nota: En 0x24, hay dos posibles casos, si la guarda se cumple, entonces se desplaza hasta 0x2C (epílogo), caso contrario, se desplaza a 0x28 (jal ra, resta) y vuelve a 0x18 (resta).

C. Indique el rango de constantes, en decimal y binario que pueden utilizarse en la instrucción li. ¿Coinciden con el rango del imm de la instrucción ADDI?

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I	immediate												rs1			func			rd				opcode									
U	immediate																		rd				opcode									

La instrucción Li es una pseudoinstrucción que carga dos operaciones base:

- lui (instrucción U) → Tiene un rango de 12 bits [-2048, 2047].
- addi (instrucción I) → Tiene un rango de 17 bits [-65536, 65535].

Para el li a0 4228 procede a hacer el desplazamiento de 12 bits

E. ¿Cuál es el valor final de a1? El valor final de a1 sería 2114, pues en ningún momento se cambia su valor ya que se necesita para llevar a0 a 0.

F. ¿Cuál es el valor final del PC? El último valor del PC es la última instrucción que ejecutó, sería 0x0014 (fin).

G. Listar la secuencia descrita por el PC. 0x0000 0x0004 0x0008 0x0010 0x0018 0x001c 0x0020 0x0024 0x0028 (vuelve a entrar a prólogo) 0x0018 0x001c 0x0020 0x0024 (evalua guarda, true va directo a epílogo) 0x002c 0x0030 0x0034 (el return nos manda de vuelta al jal) 0x0010 0x0014 (fin)

H. Indique qué valores toman los registros ra y sp: al inicio, durante y al finalizar la ejecución.

ra: 0x0010 antes de saltar a resta, dentro del prólogo, ra se guarda en la primera posición del sp ocupando los primeros 4 bytes, finalmente en epílogo, a ra se le coloca la posición del elemento 0 del stack pointer. De igual manera, siempre tiene el mismo valor, nunca cambia.

sp: Se desconoce que tiene inicialmente pero tiene cosas de memoria dinámica, variables globales, otras instrucciones, etc. Pero viendo solo el programa, el sp se mueve hacia abajo 4 bytes. Luego, en el espacio que reservamos se guarda el valor de ra. Por último, los primeros 4 bytes del stack pointer se almacenan en ra y se liberan los 4 bytes que habíamos reservado.

I: Reemplazar la segunda instrucción li a1, 2114 de modo que a1 sea a0 dividido 2 con una única instrucción

Lo que necesitamos es que a1 sea a0 dividido dos. Para esto podemos hacer un desplazamiento lógico hacia la derecha 1 bit.

li a0,4228

srli a1, a0, 1

```
li a0,4228
srli a1, a0, 1
jal ra, resta
fin:
beq zero, zero, fin
resta:
prologo:
addi sp, sp, -4
sw ra, 0(sp)
sub a0, a0, a1
beq a0, zero, epilogo
sigo:
jal ra, resta
epilogo:
lw ra, 0(sp)
addi sp, sp, 4
ret
```

3.3. Ejercicio 3

- 3.a Realizar el seguimiento del siguiente programa por al menos 12 ciclos de instrucción, qué comportamiento presenta? Asumir que el PC arranca en 0x08 y que toda dirección de memoria con un valor de memoria no explicitado vale 0.

```
00000008 <main>:
    08:      00400593      addi x11 x0 4
    0c:      0005a603      lw x12 0 x11
    10:      00400693      addi x13 x0 4
    14:      0006a683      lw x13 0 x13
    18:      0006a683      lw x13 0 x13
    1c:      fed606e3      beq x12 x13 -20 <main>

00000020 <guardar>:
    20:      fffa6737      lui x14 0xfffa6
    24:      9fd70713      addi x14 x14 -1539
    28:      00c70633      add x12 x14 x12
    2c:      02b62423      sw x11 40 x12

00000030 <fin_programa>:
    30:      00000513      addi x10 x0 0
    34:      05d00893      addi x17 x0 93
    38:      00000073      ecall
```

Nuestro procesador hace un ciclo por cada instrucción, entonces 12 ciclos de instrucción serían 12 instrucciones.

typo: el lw x13 0 x13 no va porque hace dos veces lo mismo.

Veamos qué hace cada línea para poder saber su comportamiento

1. addi x11, x0, 4 => Almacena el valor de 4 en el registro x11.
2. lw x12, 0(x11) => Busca en la memoria principal, el registro en la posición 0x00000004 y lo almacena en x12.
3. addi x13, x0, 4 => Almacena el valor de 4 en el registro x13.
4. lw x13, 0(x13) => Busca en la memoria principal, el registro en la posición 0x00000004 y lo almacena en x13.
5. beq x12, x13, -20 => Si x12 es igual a x13, vuelve 5 palabras hacia atrás. En este caso, desconocemos qué tiene x13 y x12 pero sabemos que el valor lo fue a buscar en la misma posición de memoria en ambos casos. Por lo tanto, se cumple la igualdad y salta a la primera instrucción
6. Repite 1, 2, 3, 4 y 5 exactamente igual.
7. Acá termina -> No pasa nunca a guardar ni fin programa.

El motivo del por qué se cuelga es debido a que cuando el PC comienza en 0x8, la instrucción del `addi x11, x0, 4` y `lw x12, 0(x11)` almacena en x12 la primera palabra que se encuentra en la posición de memoria 0x4, en este caso, es el valor 0.

Luego el `addi x13, x0, 4`, `lw x13, 0(x13)` va a buscar a la posición de memoria de 0x4 la primera palabra, en este caso el valor es 0, como esto lo hace dos veces, ahora hace `lw x13, 0(0x0)` y el valor que tiene 0x0 es 0.

Entonces como x13 y x12 tienen el valor de 0 se cuelga infinitamente.

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x00000034	0x05f00893	0x93	0x08	0xf0	0x05
0x00000030	0x00000513	0x13	0x05	0x00	0x00
0x0000002c	0x02b62423	0x23	0x24	0xb6	0x02
0x00000028	0x00c70633	0x33	0x06	0xc7	0x00
0x00000024	0x9fd70713	0x13	0x07	0xd7	0x9f
0x00000020	0xffffa6737	0x37	0x67	0xfa	0xff
0x0000001c	0xfed606e3	0xe3	0x06	0xd6	0xfe
0x00000018	0x0006a683	0x83	0xa6	0x06	0x00
0x00000014	0x0006a683	0x83	0xa6	0x06	0x00
0x00000010	0x00400693	0x93	0x06	0x40	0x00
0x0000000c	0x0005a603	0x03	0xa6	0x05	0x00
0x00000008	0x00400593	0x93	0x05	0x40	0x00
0x00000004	0x00000000	0x00	0x00	0x00	0x00
0x00000000	0x00000000	0x00	0x00	0x00	0x00

Valores a la hora de comparar x13 con x12 (como siempre el valor en la posición 4)

x12	a2	0x00000000
x13	a3	0x00000000

3.b Suponga que el programa hubiese sido cargado en la posición 0x0000 y el PC comienza con ese valor. ¿Cambia la ejecución del programa? ¿De qué manera? ¿Por qué?

```

00000000 <main>:
00:      00400593      addi x11 x0 4
04:      0005a603      lw x12 0 x11
08:      00400693      addi x13 x0 4
0c:      0006a683      lw x13 0 x13
10:      0006a683      lw x13 0 x13
14:      fed606e3      beq x12 x13 -20 <main>

00000018 <guardar>:
18:      fffa6737      lui x14 0xffffa6
1c:      9fd70713      addi x14 x14 -1539
20:      00c70633      add x12 x14 x12
24:      02b62423      sw x11 40 x12

00000028 <fin_programa>:
28:      00000513      addi x10 x0 0
2c:      05d00893      addi x17 x0 93
30:      00000073      ecall

```

Nota: para cambiar el inicio del PC se necesita un Ripes instalado localmente, Edit -> Settings -> Compiler y abajo de todo en .text section start address se pone el PC que se quiere. Para que esto de efecto, hay que reiniciar el programa.

Sí, la ejecución del programa cambia. En este caso termina como corresponde y no se queda colgado infinitamente haciendo un loop en el beq. Las 12 instrucciones que hace el procesador llega a terminar el programa dejando el registro de x17 con el valor de 93.

El motivo del por qué se cuelga es debido a que cuando el PC comienza en 0x0, la instrucción del addi x11, x0, 4 y lw x12, 0(x11) almacena en x12 la primera palabra que se encuentra en la posición de memoria 0x4, en este caso, es el valor 0x0005a603. Luego el addi x13, x0, 4, lw x13, 0(x13) va a buscar a la posición de memoria de 0x4 la primera palabra, en este caso el valor es 0x0005a603, como esto lo hace dos veces, ahora busca la primera palabra en la posición de la memoria 0x0005a603 pero como no tiene absolutamente nada, x13 pasa a tener el valor de 0.

Luego, como 0 (x13) es diferente a 0x0005a603 (x12) no se cumple el beq y sigue el programa.

Address		Word	Byte 0	Byte 1	Byte 2	Byte 3
0x0000002c	0x05f00893	0x93	0x08	0xf0		0x05
0x00000028	0x00000513	0x13	0x05	0x00		0x00
0x00000024	0x02b62423	0x23	0x24	0xb6		0x02
0x00000020	0x00c70633	0x33	0x06	0xc7		0x00
0x0000001c	0x9fd70713	0x13	0x07	0xd7		0x9f
0x00000018	0xffffa6737	0x37	0x67	0xfa		0xff
0x00000014	0xfed606e3	0xe3	0x06	0xd6		0xfe
0x00000010	0x0006a683	0x83	0xa6	0x06		0x00
0x0000000c	0x0006a683	0x83	0xa6	0x06		0x00
0x00000008	0x00400693	0x93	0x06	0x40		0x00
0x00000004	0x0005a603	0x03	0xa6	0x05		0x00
0x00000000	0x00400593	0x93	0x05	0x40		0x00

Valores a la hora de comparar x13 con x12.

x12	a2	0x0005a603
x13	a3	0x00000000

El problema principal acá es que el mismo código funciona diferente según donde se cargue el PC. Esto es malo, muy malo pues indica que el código NO es independiente de la posición. Por lo tanto, el programa está mal hecho y se lo debe corregir para que funcione siempre sin importar donde comience.

Lo que deja en claro el por qué no cumple el Independent Position Code es que a la hora de hacer un lw, está haciendo un lw con una posición prácticamente hardcodeda (0x4) que corresponde a una posición que es cargada para el PC.