Guia 2 ASM

Ejercicio 1

```
addi t0, x0, 0x512
```

1.a

Copiar el codigo generado y pasarlo a binario

`0x51200293`=`0101 0001 0010 0000 0000 0010 1001 0011`

1.b

Agrupar los bits de acuerdo al tipo de instruccion e identificar los operandos y elementos

Addi es una instruccion tipo I con la siguiente estructura

- `addi rd, rs1, imm`
- `opcode`: 7 bits
- `rd`: 5 bits
- `funct3`: 3 bits
- `rs1`: 5 bits
- `imm`: 12 bits
  Por lo que en nuestra instruccion cada elemento tiene los siguientes valores
- `opcode` = `0010011`
- `rd` = `00101`
- `funct3` = `000`
- `rs1` = `00000`
- `imm` = `010100010010`

Ejercicio 2

```
.text
    # almacenar 0x12345000 en s1
    lui s1, 0x12345
    # sumarle 0x678 (esto seria 0x00000678)
    addi s1, s1, 0x678
```

Ejercicio 3

Codigo 1:

```
.text
    # Guardar inmediato 0x111117FF en a0
    li a0, 0x111117FF
    # Guardar inmediato 0x11111CAB en a1
    li a1, 0x11111CAB
```

Codigo 2:

```
.text
    # Guardar 0x11111000 en a2
    lui a2, 0x11111
    # Hacer una operacion or bit a bit y almacenarla en a2
    # 0001 0001 0001 0001 0001 0000 0000 0000 or
    # 0000 0000 0000 0000 0000 0111 1111 1111 = 0x111117FF
    ori a2, a2, 0x7FF
    # Guardar 0x11112000 en a3
    lui a3, 0x11112
    # Hacer una operacion or bit a bit
    # 0001 0001 0001 0001 0010 0000 0000 0000 or
    # 1111 1111 1111 1111 1111 0011 0101 0101
    ori a3, a3, -0x355
```

Codigo 3:

```
.text
    # Guardar 0x11111000 en a4
    lui a4, 0x11111
```

```
    # Sumarle 0x000007FF
    addi a4, a4, 0x7FF
    # Guardar 0x11112000 en a5
    lui a5, 0x11112
    # Sumarle 0xfffff355
    addi a5, a5, -0x355
```

3.a

Que codigo se genera? A que conclusion llega? Investigue por que

Para el codigo 1, que consta de pseudo instrucciones `li`, se generan basicamente basicamente las mismas instrucciones que en el codigo 3 y obtenemos el mismo resultado en ambos (pero en registros distintos). Tanto en el codigo 2 como en el 3, las instrucciones Basic y Source son iguales, porque no estamos usando pseudo instrucciones.
En el codigo 2 no obtenemos el mismo resultado que en los otros, siendo que la unica diferencia con el codigo 3 es que usamos `ori` en lugar de `addi`. Esto se debe a que en el primer caso estamos haciendo una operacion or bit a bit, mientras que en el segundo estamos sumando. Ademas, la representacion con signo extendido que usamos hace que `-0x355` sea igual a `0xfffff355` (recordemos que `f` = `1111`)

3.b

Que resultado almacenan los registros cuando lo ejecuta paso a paso?

En el codigo 1, a0 y a1 pasan de `0x00000000` a `0x111117FF` y `0x11111CAB` directamente, si ignoramos las instrucciones generadas en Basic, si no es exactamente igual que en el codigo 3.
En el codigo 2 y 3 tenemos

- `a2=0x00000000` -> `0x11111000`->`0x111117FF`
- `a3=0x00000000` -> `0x11112000`->`0xFFFFFCAB`
- `a4=0x00000000` -> `0x11111000`->`0x111117FF`
- `a5=0x00000000` -> `0x11112000`->`0x11111CAB`

3.c

Por que hay que agregar el valor con el signo '-'? Note que para llegar al valor `a1` tiene que estar incrementado en 1. Siempre?

Uno podria pensar que, siguiendo la misma logica que con `0x111117FF` podemos hacer `lui a5, 0x11111` y luego sumamos `0xcab`. El problema con esto (y lo podemos probar) es que, como recordamos de 1.a , el inmediato es de 12 bits **con signo**.
Entonces, si queremos llegar al valor `0x111117FF` necesitaremos restar de un valor mayor.

Para esto guardamos `0x11112000` y a este numero podemos restarle `0x355` y obtenemos el valor deseado.
Generalizando, para lograr valores entre `0x11111000` y `0x111117FF`, sumamos a `0x11111000`. Y para valores entre `0x11111800` y `0x11111FFF`, restamos a `0x11112000`.

3.d

Que valor resulta de hacer complemento a 2 del valor `0xCAB`?

En binario, `0xcab` es igual a `110010101011` y en complemento a 2, el bit mas significativo representa el bit de signo. Para saber que valor representa, invertimos todos los bits (porque el bit de signo es 1, ojo) y obtenemos `001101010100` que es igual a `0x354`

Ejercicio 4

Supongo que se refiere a comparar un programa usando addi y otro usando sub, no esta claro.

```
.text
    # Codigo 1:
    # Guardamos 20 en t0
    addi t0, x0, 20
    # Restamos 5 a 20 con addi
    addi t0, t0, -5

    # Codigo 2:
    # Guardamos 20 en t0
    addi t0, x0, 20
    # Guardamos 5 en t1
    addi t1, x0, 5
    # Restamos 5 a 20
    sub t0, t0, t1
```

4.a

Identifique los codigos generados en ambos casos. Como se representan los valores, en particular si hay negativos?

Si hay negativos, en el caso de `sub` no importa, pero para el caso de `addi` se representan como ya vimos, como "sign extended" o signo extendido. Esto significa que, si es negativo, todos los bits "restantes" son `1`.
En este caso, `-5` se representa como `0xfffffffb`.

Ejercicio 5

```
.text
    lui a0, 0x10010
    addi t0, x0, 0x234
    sw t0, 0(a0)
```

Ejercicio 6

Tiene sentido esta instruccion? Que hace?

```
    add t0, t0, t0
```

Si tiene sentido, es equivalente a duplicar el valor en t0. `t0 = t0 + t0`.

Ejercicio 7

Indique para que sirve esta instruccion

```
    add t0, t0, zero
```

La instruccion no hace nada, suma 0 al valor ya existente en `t0`. `t0 = t0 + 0`.

Ejercicio 8

Representaremos las variable como:

- `a` -> `t0`
- `b` -> `t1`
- `c` -> `t2`
- `d` -> `t3`
- `e` -> `t4`

8.a

`a = b;`

```
add t0, t1, x0
```

8.b

`a = b + c;`

```
add t0, t1, t2
```

8.c

`a = a + 1;`

```
addi t0, t0, 1
```

8.d

`a = c + 2;`

```
addi t0, t2, 2
```

8.e

`a = b + c + d + e;`

```
add t0, t1, t2
add t0, t0, t3
add t0, t0, t4
```

8.f

`a = b - c;`

```
sub t0, t1, t2
```

8.g

`a = c + (b - d);`

```
sub t0, t1, t3
add t0, t0, t2
```

8.h

`a = (b + c) - (d + e);`

```
add t0, t1, t2
sub t0, t0, t3
sub t0, t0, t4
```

8.i

`a = b * c;`

```
mul t0, t1, t2
```

8.j

`a = b / c;`

```
div t0, t1, t2
```

8.k

`a = 3 * e`

```
addi a0, x0, 3  # auxiliar
mul t0, a0, t4
```

8.l

`a = (b - c) * (d - e);`

```
sub a0, t3, t4  # auxiliar
sub t0, t1, t2
mul t0, t0, a0
```

8.m

`a = b * c * d;`

```
mul t0, t1, t2
mul t0, t0, t3
```

8.n

`a = (b + c) * (d / e);`

```
div a0, t3, t4  # Auxiliar
add t0, t1, t2
mul t0, t0, a0
```

Ejercicio 9

```
.text
    # Guarda 0x465 en t0
    ori t0, zero, 0x465
    # Guarda 0x0FF en t1
    ori t1, zero, 0x0FF
    # Guarda 0x123 en t2
    ori t2, zero, 0x123
    # 0100 0110 0101 and
    # 0001 0010 0011 = 0000 0010 0001
    and a0, t0, t2
    # 1111 1111 and
    # 0011 0101 = 0011 0101
    andi a1, t1, 0x35
    # 0000 1111 1111 or
```

```
    # 0100 0110 0101 = 0100 1111 1111
    or a2, t1, t0
    # 1111 1111 or
    # 0000 0001 = 1111 1111
    ori a3, t1, 1
    # 0x0FF = 0x100
    addi a4, t1, 1
    # 0100 0110 0101 xor
    # 0100 0110 0101 = 0
    xor a5, t0, t0
    # 0100 0110 0101 xor
    # 0011 0111 0001 = 0111 0001 0100
    xori a6, t0, 0x371
    # 0111 0001 0100 xor
    # 0011 0111 0001 = 0100 0110 0101
    xori a7, a6, 0x371
    # not 0000 0010 0001 = 1111 1101 1110
    not s1, a0
```

Basicamente:

- `t0` = `0x465`
- `t1` = `0x0FF`
- `t2` = `0x123`
- `a0` = `0x021`
- `a1` = `0x035`
- `a2` = `0x4FF`
- `a3` = `0x0FF`
- `a4` = `0x100`
- `a5` = `0x000`
- `a6` = `0x714`
- `a7` = `0x465`
- `s1` = `0xFFFFFFDE`

Ejercicio 10

```
.text
    # t0 = 0x1DC
    ori t0, zero, 476
```

```
    # t1 = 0xFF
    ori t1, zero, 0xFF
    # t2 = 0x00001000
    lui t2, 1
    # t3 = 0x4
    ori t3, zero, 4
    # t4 = 0xC3010000
    lui t4, 0xC3010
    # s0 =  0001 1101 1100 << 16
    # = 0001 1101 1100 0000 0000 0000 0000 = 0x1DC0000
    slli s0, t0, 16
    # s1 = 1111 1111 >> 1
    # = 0111 1111 = 0x7F
    srai s1, t1, 1
    #
    srl s2, t4, t3
    slli s4, t3, 1
    add s4, s4, t3
    sll s5, t0, s4
```