Guia 1 ASM

Guia 1

Ejercicio 1

```
.text
    # lui: Load Upper Imm
    # lui rd, imm
    # rd: register
    # imm: numero de hasta 20 bits (max 1048576 o 0x100000)

    # Cargar 0x12345 en t0
    lui t0, 0x12345
    # Cargar 201 en t1
    lui t1, 201
    # Cargar 0xABCDE en t2
    lui t2, 0xABCDE
```

1.a Que diferencia se visualiza entre las instrucciones del codigo en Source y Basic?

Basic	Source
`lui x5, 0x00012345`	`lui t0, 0x12345`
`lui x6, 0x000000c9`	`lui t1, 201`
`lui x7, 0x000abcde`	`lui t2, 0xABCDE`

El codigo **Source** son basicamente las instrucciones que nosotros escribimos como las escribimos.
El codigo **Basic** son las instrucciones que se guardan en la **Memoria de Instrucciones**, con los inmediatos en hexadecimal y los registros sin alias.

1.b Cual es la direccion de comienzo del programa y que longitud tiene cada instruccion?

La direccion de comienzo (program counter inicial) es `0x00400000`, esto podemos verlo en la columna Address. Y como vemos tambien en esa columna, cada instruccion tiene una longitud de 4.

1.c Escribir el codigo objeto (Code) de cada instruccion en binario

Es lo que aparece en la columna **Code**

1. `0x123452b7`

2. `0x000c9337`
3. `0xabcde3b7`

1.d Que valores inicialmente tienen los registros t0, t1, t2 y pc?

t0: `0x00000000`
t1: `0x00000000`
t2: `0x00000000`
pc: `0x00400000`

1.e En que direccion comienza el segmento de datos?

Ventanita **Data Segment**
`0x10010000`

1.f Presionar F7 para ejecutar la primera instruccion, que valor toma t0?

`0x12345000`

1.g En cuanto y por que cambia el valor del registro pc? Cual es su funcion?

El valor del registro pc cambia en 4 bits porque es la longitud de cada instruccion, y justamente su funcion es "apuntar" a la proxima instruccion en la Memoria de Instrucciones

1.h Seguir ejecutando el programa y verificar los valores de t1, t2 y pc

instruccion	pc	t0	t1	t2
---	`0x00400000`	`0x00000000`	`0x00000000`	`0x00000000`
`lui t0, 0x12345`	`0x00400004`	`0x12345000`	`0x00000000`	`0x00000000`
`lui t1, 201`	`0x00400008`	`0x12345000`	`0x000c9000`	`0x00000000`
`lui t2, 0xABCDE`	`0x0040000c`	`0x12345000`	`0x000c9000`	`0xabcde000`

Ejercicio 2

```
.text
    # Cargar 0x10010 en el registro a1
    lui a1, 0x10010

    # Load Word
    # lw rd, rs1, imm
    # rd: registro destino
    # rs1: registro fuente (source)
    # imm: bits de desplazamiento desde el valor guardado en rs1
```

```
    # cargar en t0 el valor guardado en 0x10010000=(a1)
    lw t0, 0(a1)
    # cargar en t1 el valor guardado en 0x10010004=(a1)+4
    lw t1, 4(a1)
    # cargar en t2 el valor guardado en 0x10010004=(a1)+8
    lw t2, 8(a1)
```

2.a Cargar manualmente los valores de las siguientes palabras a partir de la direccion 0x1001000 (del Segmento de Datos), 0x12345678, 1000, 0x12ab34cd

Estos datos se cargan en la ventana llamada **Data Segment** (Segmento de Datos).

2.b Cual es la funcion de `lui a1, 0x10010`?

Esta instruccion carga la primer direccion del Segmento de Datos en el registro a1 (`0x10010000`)

2.c Que hace la instruccion `lw t1, 4(a1)`? Describir cuales son sus operandos. Si a1 fuera 0x10010004, se obtendria el mismo resultado al hacer `lw t1, 0(a1)`? por que?

La instruccion `lw t1, 4(a1)` carga en el registro t1 los 32 bits guardados en el segmento de datos a partir de la direccion de memoria guardada en `a1 + 4 bits=0x10010004`. Si a1 fuera 0x10010004, obtendriamos el resultado que tuvieramos `lw t1, 8(a1)`, es decir desplazado. Estariamos leyendo la direccion `a1+4=0x1001008`.

2.d Ejecutar paso a paso el programa y responda que valores carga en t0, t1 y t2

Instruccion	pc	t0	t1	t2	a1
---	`0x00400000`	`0x00000000`	`0x00000000`	`0x00000000`	`0x00000000`
`lui a1, 0x10010`	`0x00400004`	`0x00000000`	`0x00000000`	`0x00000000`	`0x10010000`
`lw t0, 0(a1)`	`0x00400008`	`0x12345678`	`0x00000000`	`0x00000000`	`0x10010000`
`lw t1, 4(a1)`	`0x0040000c`	`0x12345678`	`0x000003e8`	`0x00000000`	`0x10010000`
`lw t2, 8(a1)`	`0x00400010`	`0x12345678`	`0x000003e8`	`0x12ab34cd`	`0x10010000`

Ejercicio 3

```
# Escribimos en el Data Segment
.data
# Referenciamos una palabra (4 bytes)
valor: .word 0
```

```
.text
    # Cargamos 0x12345, 0x345 y 0x5 en t0, t1 y t2
    lui t0, 0x12345
    lui t1, 0x345
    lui t2, 0x5
    # Guardamos el valor de la direcc
    sw t0, valor, t6
    # Guardamos la direccion de 'valor' en a7
    la a7, valor
    # Guardamos en t1 el valor almacenado en la direccion (a7)+4
    sw t1, 4(a7)
    # Guardamos en t2 el valor almacenado en la direccion (a7)+8
    sw t2, 8(a7)
```

3.a Que diferencia se visualiza entre las instrucciones del codigo en Source y Basic?

Source	Basic
`sw t0, valor, t6`	`auipc x31, 0x0000fc10`
	`sw x5, 0xfffffff4(x31)`
`la a7, valor`	`auipc x17, 0x0000ffc10`
	`addi x17, x17, 0xffffffec`

La primer instruccion sw (store word) pretende guardar en la direccion de valor, con un offset almacenado en t6 (en este caso 0), la palabra almacenada en t0 (x5). La instruccion sw normalmente se usa como `sw rs1, offset(rs2)`, por lo que esto se traduce en:

1. Add Upper Immediate to PC (`auipc rd,imm`): suma un inmediato de 20 bits con el PC y lo guarda en un registro destino. En este caso sumamos el PC y `0x0000fc10` como upper immediate (se cargan los 20 bits en la parte "alta" del registro, seguid de ceros), que es igual a `0x0fc10000`+`0x0040000c`=`0x1001000c`, y lo guardamos en x31.
2. Store Word (`sw rs1,imm(rs2)`): guarda la palabra almacenada en rs1 en la direccion `rs2+imm`. En este caso guardamos la palabra guardada en x5 (`0x12345000`) en la direccion `x31+0xfffffff4`=`0x10010000`. El numero `0xfffffff4` es un offset negativo de 4 bytes de la direccion `0x1001000c`.

`la a7, valor` es una pseudoinstruccion que carga la direccion de `valor` en el registro a7. Se llama pseudo instruccion ya que el ISA no tiene una instruccion que cargue directamente una direccion de 32 o 64 bits, por lo que se usa combinacion de instrucciones:

1. `auipc rs1, imm`: almacena en el registro rs1 la suma del inmediato superior `imm` y el program counter PC. En este caso `auipc x17, 0x0000fc10` guarda en x17 la suma `0x0fc10000`+`0x00400014`=`0x10010014`

2. `addi rs1, rs2, imm`: almacena en el registro rs1 la suma del registro rs2 y el inmediato `imm`. En este caso almacenamo en x17 la suma de x17 y `0xffffffec` que es igual a `0x10010014`-`0x000000ec`=`0x10010000`, que es igual a la direccion de `valor`.

3.b Pasar el codigo generado en la instruccion de la linea 4 (en Code), indicando los campos que la conforman.

El codigo de la instruccion es `0x0fc10f97`, en binario `00001111110000010000111110010111`. Segun la documentacion de risc-v, auipc es una instruccion tipo U, por lo que tiene una estructura tal que los primero 7 bits son el opcode, los siguientes 5 bits indican el registro destino y por ultimo tenemos 20 bits correspondientes al inmediato. Entonces:
Imm: `00001111110000010000`
rd: `11111`
opcode: `0010111`

3.c Hacer un seguimiento paso a paso y verificar que registros intervienen en las operaciones

Pregunta boluda, con la a ya se entiende.

3.d Que valor tenia el registro t6 antes de ejecutar el programa y en cuanto quedo?

Tenia valor `0x00000000` y termino con `0x1001000c` por lo que explicamos en a

3.e Verificar que valor se almaceno en memoria y en que direccion

Se guardo el valor `0x12345000` en la direccion `0x10010000`

3.f Editar el programa y cambiar la instruccion Basic por las que surgieron en Source

*Cambiar Source (las originales) por las Basic.

3.g Verificar el funcionamiento del programa con los cambios realizados en el punto (f)

Funciona igual

Ejercicio 4

```
.text
    # Guardar 0x12345000 en t0
    lui t0, 0x12345
    # Almacenar en la direccion del stack pointer el valor guardado en t0
(0x12345000)
    sw t0, (sp)
    # Guardar 0x10010000 en a1
    lui a1, 0x10010
```

```
    # Guardar en t1 el valor almacenado en el stack pointer
    lw t1, (sp)
    # Almacenar en la direccion almacenada en a1 el valor almacenado
    # en t1 (que era el valor al que apuntaba el stack pointer)
    sw t1, (a1)

    # Esto guarda 10 (0x0000000a) en a7
    # Se suele usar para indicar la salida del programa.
    li a7, 10
    ecall
```

4.a Que hace el programa?

Almacena una palabra en el Stack y luego almacena en el Data Segment un valor del Stack (el que almacenamos antes).
Esto seria masomenos equivalente a:

```
int t0, t1;
int* sp = new int, a1;

t0 = 0x12345;     // lui t0, 0x12345
*sp = t0;         // sw t0, (sp)
// si suponemos que 0x10010000 es una direccion de memoria
// accesible por nuestor programa...
a1 = 0x10010000; // lui a1, 0x10010000
t1 = *sp;         // lw t1, (sp)
*a1 = t1;         // sw t1, (a1)

return 0;         // li a7, 10 ; ecall
```

4.b En donde se almacena el valor de t0?

Se almacena en el primer valor del Stack, referenciado por el Stack Pointer (sp).

4.c Por que se asigna 0x1001 a a1?

No se asigna `0x1001`, se asigna `0x10010000`. Se hace para usarlo como referencia al primer valor del Data Segment.

4.d Que accion hace `sw t0,(sp)` y cual es la direccion en donde se almacenara el dato? Que valor habia antes en esa direccion y que valor hay despues de ejecutar la instruccion?

Hace lo que dijimos en 4.b, almacena el valor de t0 en el Stack. El Stack Pointer (sp) es, justamente, un puntero al Stack, que al inicio del programa apunta a la primer posicion, o Head.
El valor que habia antes era 0, y el valor es, como ya dijimos, el de t0. `0x12345000`.

4.e Al ejecutar `lw t1,(sp)`, que valor toma t1?

Toma el valor almacenado en la direccion del Stack apuntada por (sp), que era la primer posicion del Stack. El valor que habia alli almacenado era `0x12345000`.

4.f Con `sw t1,(a1)`, en que direccion se almacena el valor de t1?

En la direccion almacenada en a1, `0x10010000`.

Ejercicio 5

```
.data
elemento: .word 0x4E1C532D
.text
    lw t1, elemento
```

5.a que instrucciones se generan?

```
auipc x6, 0x0000fc10
lw x1, 0(x6)
```

5.b En las instrucciones generadas, cuales son los registros intervinientes?

Solo t1.

5.c Por que valor es reemplazada la etiqueta elemento?

la etiqueta elemento es reemplazada por la direccion de memoria donde esta la palabra a la que hace referencia.

5.d Cual es la ventaja de utilizar etiquetas en lugar de direcciones?

Simplicidad de codigo, mejor lectura y no corremos peligro de "perder" la referencia, que en caso de tenerla en un registro podemos "pisarla".

5.e Verificar el valor almacenado en la direccion de memoria apuntada por elemento

Es `0xE1C532D`, exactamente el que definimos en `.data`.

5.f Al ejecutar el programa, que valor queda en t1?

El valor guardado en la direccion que referencia `elemento`.

Ejercicio 6

```
.data
valor: .word 0x805215C9
.text
```

```
    # Load Address
    la s0, valor
    # Load Byte Unsigned
    lbu a1, 0(s0)
    lbu a2, 1(s0)
    lbu a3, 2(s0)
    lbu a4, 3(s0)

.text
    la s0, valor
    # Load Byte
    lb a1, 0(s0)
    lb a2, 1(s0)
    lb a3, 2(s0)
    lb a4, 3(s0)
```

6.a Como se almacena el valor en el segmento de datos? La arquitectura es big endian o little endian?

En el Data Segment `0x805215C9` se almacena desde el byte menos significativo al menos significativo, es decir `C9`, `15`, `52`, `80`. Esta arquitectura se llama Little Endian

6.b Que valores se almacenan en cada caso en a1, a2, a3 y a4? Por que?

Los valores almacenados en cada caso son

rd	lbu	lu
a1	`0x000000C9`	`0xffffffC9`
a2	`0x00000015`	`0x00000015`
a3	`0x00000052`	`0x00000015`
a4	`0x00000080`	`0xffffff80`

En el caso de `lbu` leemos sin signo, por lo que ignoramos el primer bit. En cambio, con lb queremos el signo, por lo que, al usarse complemento 2, cargamos el mismo numero pero rellenando con 1's en caso de que sea negativo, o 0's en caso de ser positivo.
Entonces, `0xC9` y `0x80` son negativos por empezar con 1
`0xC9`: `1100 1001` -> `-55`
`0x80`: `1000 0000` -> `-128`
Y `0x52` y `0x15` positivos:
`0x52`: `0101 0010`
`0x15`: `0001 0101`

Ejercicio 7

```
.data
valor: .word 0x805215C9
.text
    # Load Address
    la s0, valor
    # Load Half-word Unsigned
    lhu a1, 0(s0)
    lhu a2, 2(s0)

.text
    la s0, valor
    # Load Half-word
    lh a1, 0(s0)
    lh a2, 2(s0)
```

7.a Analizar las instrucciones generadas. Identificar los registros intervinientes para poder extraer un valor de la memoria

Los registros que intervienen son s0 para almacenar la direccion de memoria y a1 y a2 para almacenas los valores.

7.b Que valores se almacenan en cada caso en a1 y a2? Por que?

rd	lhu	lh
a1	`0x000015C9`	`0x000015C9`
a2	`0x00008052`	`0xffff8052`

Es el mismo caso que en el ejercicio 6, pero esta vez lo que se almacenan son medias palabras, o 2 bytes, en lugar de bytes. Y la extension de signo funciona de la misma forma.

Ejercicio 8

Realizar un programa que asigne las palabras `0xABCD0000` en la primera direccion del segmento de datos y 0x12340000 en la siguiente

```
.text
    # Guardamos la direccion del Data Segment
    lui a1, 0x10010
    # Guardamos 0xABCD0000 en t0
    lui t0, 0xABCD0
    # Almacenamos el valor de t0 en (a1)
```

```
    sw t0, 0(a1)
    # Guardamos 0x12340000 en t0
    lui t0, 0x12340000
    # Almacenamos el valor de t0 en (a1)+4
    sw t0, 4(a1)
```

Ejercicio 9

Agregar las lineas necesarias al programa del Ejercicio 8 para intercambiar los valores en memoria.

```
.text
    # Guardamos la direccion del Data Segment
    lui a1, 0x10010
    # Recuperamos el valor de cada direccion
    lw t0, 0(a1)
    lw t1, 4(a1)
    # Guardamos en las direcciones opuestas
    sw t0, 4(a1)
    sw t1, 0(a1)
```