



IIC2343 – Arquitectura de Computadores (II/2019)

Assembler

Ensamblador para el computador básico

1. Objetivos

Para facilitar la programación de su computador básico deberán desarrollar un *assembler*. Este debe ser capaz de traducir un programa escrito en el assembly del proyecto al código de máquina de su proyecto. De este modo se podrá probar y evaluar el correcto funcionamiento de su arquitectura.

Como solo se evaluará su existencia y el correcto funcionamiento, no se revisará el código fuente. Cada grupo tendrá la libertad de programar su *assembler* en el lenguaje que prefieran, con la condición de debe quedar a disposición de los ayudantes como un archivo ejecutable que pida el *path* a un archivo *.asm*

De no realizar un archivo ejecutable, el assembler puede estar escrito en Python3, y se podrá ejecutar a través de consola de comandos de la siguiente forma:

```
1 $> python3 assembler.py as1.asm
```

Para ambos casos, debe estar informado en la bitácora del proyecto cual método realizó.

2. Funcionamiento

El programa debe recibir la ubicación de un archivo de formato *.asm* y escribir el archivo un archivo *ROM.vhd*.

El código que va a leer es un programa en el lenguaje *assembly* de su proyecto y el resultado debe ser el código correspondiente a la ROM, pero con las instrucciones del programa en su código máquina. A continuación, un ejemplo de la ROM de 4096 palabras de 36 bits que tiene solo 9 instrucciones:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4 USE IEEE.NUMERIC_STD.ALL;
5
6 entity ROM is
7     Port (
8         address : in  std_logic_vector (11 downto 0);
9         dataout  : out std_logic_vector (35 downto 0)
10    );
11 end ROM;
```


Ejemplo:

```
1 DATA:
2 variable1 2
3 variable2 3
4 CODE:
5 MOV A,(variable1)
6 JMP fin
7 MOV A,(variable2)
8 fin:
```

Su compilador **NO** puede necesitar elementos adicionales como una línea *END* al final del código para poder compilar. Y recuerde que la CPU solo opera con números positivos de 16 bits, por lo que no debe soportar números negativos.

3.1. Entrega 3

Para la entrega 3 su assembler debe:

- Aceptar literales como decimal en el formato *102d* y *102*, binario en el formato *1010b* y hexadecimal en el formato *AAh*.
- Comentarios en una línea usando *//* como delimitador.
- Espacios y tabulaciones en distintas partes del código, además de líneas en blanco entre instrucciones.

Ejemplo:

```
1 // Esto es un comentario
2     DATA:
3 // Línea en blanco
4 v1    10           // 10 se asume decimal
5     v2    10d      // 10 en decimal
6 v3    10b          // 10 en binario
7 v4    10h          // 16 en hexadecimal
8
9     CODE:
10 MOV B, ( v4      ) // B = Mem[3] = 16
11 MOV  A, ( 10b    ) // A = Mem[2] = 2
12
13     label1:
14     MOV (v1),B     // Mem[0] = 16
15     JMP label2     // Salta a label2
16 1end:
17
18     label2:
19
20 JMP    1end        // Salta a 1end
```

- Soportar tanto valores como punteros, tal que la variable entre paréntesis es el valor y que sin ellos es su lugar en la memoria. O sea, si tenemos una variable *var*, usar (*var*) trae su valor y *var* trae su dirección en memoria. Por lo tanto, el assembler debe tener una forma de recordar en qué parte de la memoria se guardan las variables.
- Aceptar el nombre de una variable como literal de su dirección en la RAM.
- Definición de arreglos de variables declarando una lista de valores pero nombrando solo el primero.

Ejemplo:

```

1 DATA:
2   a           0
3   arreglo     10
4               101b
5               2h
6               7d
7 CODE:
8   MOV B,arreglo // B = 1
9   INC B         // B = B + 1 = 2
10  MOV A,(B)     // A = Mem[B] = 5

```

3.2. Entrega 4

Para la entrega 4 su assembler debe:

- Aceptar literales como caracteres desde el 32 al 126 de la tabla ASCII, en el formato '*c*'.
- Definición de *strings* en el formato "*ho la*" como arreglo de caracteres seguido por un 0.

Ejemplo:

```

1 DATA:
2   letrac 'c'    // 99
3   string "ho la" // ['h', 'o', ' ', 'l', 'a', 0] = [104, 111, 20, 108, 97, 0]
4 CODE:
5   MOV A,(letrac) // A = 99
6   SUB A,'a'      // A = A - 97 = 2

```

4. Assembly

Esta es la lista de instrucciones separadas por entrega.

Entrega 3		
MOV	A,B B,A A,Lit B,Lit A,(Dir) B,(Dir) (Dir),A (Dir),B A,(B) B,(B) (B),A (B),Lit	guarda B en A guarda A en B guarda un literal en A guarda un literal en B guarda Mem[Dir] en A guarda Mem[Dir] en B guarda A en Mem[Dir] guarda B en Mem[Dir] guarda Mem[B] en A guarda Mem[B] en B guarda A en Mem[B] guarda Lit en Mem[B]
ADD SUB AND OR XOR	A,B B,A A,Lit B,Lit A,(Dir) B,(Dir) (Dir) A,(B) B,(B)	guarda A op B en A guarda A op B en B guarda A op literal en A guarda A op literal en B guarda A op Mem[Dir] en A guarda A op Mem[Dir] en B guarda A op B en Mem[Dir] guarda A op Mem[B] en A guarda A op Mem[B] en B
NOT SHL SHR	A B,A (Dir),A (B),A	guarda op A en A guarda op A en B guarda op A en Mem[Dir] guarda op A en Mem[B]
INC	A B (Dir) (B)	incrementa A en una unidad incrementa B en una unidad incrementa Mem[Dir] en una unidad incrementa Mem[B] en una unidad
DEC	A	decrementa A en una unidad
CMP	A,B A,Lit A,(Dir) A,(B)	hace A-B hace A-Lit hace A-Mem[Dir] hace A-Mem[B]
JMP	Ins	carga Ins en PC
JEQ	Ins	carga Ins en PC si en el status Z = 1
JNE	Ins	carga Ins en PC si en el status Z = 0
JGT	Ins	carga Ins en PC si en el status N = 0 y Z = 0
JGE	Ins	carga Ins en PC si en el status N = 0
JLT	Ins	carga Ins en PC si en el status N = 1
JLE	Ins	carga Ins en PC Ins si en el status N = 1 o Z = 1
JCR	Ins	carga Ins en PC Ins si en el status C = 1
NOP		no hace cambios
PUSH	A B	guarda A en Mem[SP] y decrementa SP guarda B en Mem[SP] y decrementa SP
POP	A B	incrementa SP y luego guarda Mem[SP] en A incrementa SP y luego guarda Mem[SP] en B
CALL	Ins	guarda PC+1 en Mem[SP], carga Ins en PC y decrementa SP
RET		incrementa SP y luego carga Mem[SP] en PC
Entrega 4		
IN	A,Lit B,Lit (B),Lit	guarda Input[Lit] en A guarda Input[Lit] en B guarda Input[Lit] en Mem[B]
OUT	A,B A,(B) A,(Dir) A,Lit	envia A a Output[B] envia A a Output[Mem[B]] envia A a Output[Mem[Dir]] envia A a Output[Lit]