

IIC2343 - Arquitectura de Computadores (II/2019)

Assembler

Ensamblador para el computador básico

1. Objetivos

Para facilitar la programación de su computador basico deberán desarollar 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:

```
library IEEE:
  use IEEE.STD_LOGIC_1164.ALL;
  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
                  : out std_logic_vector (35 downto 0)
10
            );
11
  end ROM;
12
```

```
13 architecture Behavioral of ROM is
14
15
  type memory_array is array (0 to ((2 ** 12) - 1)) of std_logic_vector (35 downto 0);
16
17
  signal memory: memory_array:= (
18
         "0000000000000000000100000111000000010".

    instruccion 1

19
         "000000000000000000000000011000000011"
                                                   instruccion 2
         "0000000000000000001000000111000000010"
20
                                                -- instruccion 3
         "0000000000000000000100000011000000011".
21
                                                   instruccion 4
22
         "0000000000000000001100000111000000010"
                                                   instruccion 5
         "00000000000000000100000011000000011"
23
                                                   instruccion 6
         "0000000000000000010000000111000000010"
24
                                                   instruccion 7
25
         "00000000000000001100000011000000011"
                                                -- instruccion 8
         "000000000000000000000000111000000010"
26
                                                -- instruccion 9
27
         -- el resto de las
         28
                                                — instrucciones estan
29
         en blanco
```

De este modo podrán probar el código con tan solo copiar el resultado de su assembler a la ROM de su proyecto. Recuerden que la estructura de cada una de las instrucciones de 36 bits queda a criterio de cada grupo.

3. Requisitos

Su assembler debe reconocer variables solo al comienzo del programa en la zona llamada *DATA*:. Estas variables deben almacenarse en la RAM al comenzar el programa, por lo que es responzabilidad del *assembler* asignarles direcciones en la RAM agregando las instrucciones que sean nesesarias. Cada variable declarada en una linea es del formato *nombre valor*. Su *assembler* debe recordar estos nombres y sus direcciones en la RAM y reemplazarlos en las instrucciones según corresponda.

Luego, la linea *CODE*: delimita el fin de la *DATA*: y el comienzo de las instrucciones del programa. Según corresponda cada instrucción en assembly debe ser traducida a una o más instrucciones en código de máquina. Además, en esta zona se debe poder definir *labels* como una palabra seguida inmediatamente por dos puntos en una linea aparte. Su *assembler* debe recordar los nombres y las direcciones de los labels para hacer los remplazos en las instrucciones según corresponda.

Ejemplo:

```
DATA:
variable1 2
variable2 3
CODE:
MOV A,(variable1)
JMP fin
MOV A,(variable2)
fin:
```

Su compilador **NO** puede necesitar elementos adicionales como una línea *END* al final del código para poder compilar. Y recuerde que 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 linea usando // como delimitador.
- Espacios y tabulaciones en distintas partes del código, además de lineas en blanco entre intrucciones.

Ejemplo:

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

```
DATA:
2
                    0
   а
3
   arreglo
                    10
4
                    101b
5
                    2h
6
                    7d
  CODE:
                   // B = 1
  MOV B, arreglo
                    // B = B + 1 = 2
9
   INC B
  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:

```
DATA:
letrac 'c' // 99
string "houla" // ['h', 'o', '', 'l', 'a', 0] = [104, 111, 20, 108, 97, 0]

CODE:
MOV A,(letrac) // A = 99
SUB A,'a' // A = A - 97 = 2
```

4. Assembly

Esta es la lista de instrucciones separadas por entrega.

Entrega 3		
MOV	A,B	guarda B en A
	B,A	guarda A en B
	A,Lit	guarda un literal en A
	B,Lit	guarda un literal en B
	A,(Dir)	guarda Mem[Dir] en A
	B,(Dir)	guarda Mem[Dir] en B
	(Dir),A	guarda A en Mem[Dir]
	(Dir),B	guarda B en Mem[Dir]
	A,(B)	guarda Mem[B] en A
	B,(B)	guarda Mem[B] en B
	(B),A	guarda A en Mem[B]
	(B),Lit	guarda Lit en Mem[B]
ADD SUB AND OR XOR	A,B	guarda A op B en A
	B,A	guarda A op B en B
	A,Lit	guarda A op literal en A
	B,Lit	guarda A op literal en B
	A,(Dir)	guarda A op Mem[Dir] en A
	B,(Dir)	guarda A op Mem[Dir] en B
	(Dir)	guarda A op B en Mem[Dir]
	A,(B)	guarda A op Mem[B] en A
	B,(B)	guarda A op Mem[B] en B
NOT SHL SHR	A	guarda op A en A
	B,A	guarda op A en B
	(Dir),A	guarda op A en Mem[Dir]
	(B),A	guarda op A en Mem[B]
INC	A	incrementa A en una unidad
	В	incrementa B en una unidad
	(Dir)	incrementa Mem[Dir] en una unidad
	(B)	incrementa Mem[B] en una unidad
DEC	A	decrementa A en una unidad
CMP	A,B	hace A-B
	A,Lit	hace A-Lit
	A,(Dir)	hace A-Mem[Dir]
	A,(B)	hace A-Mem[B]
$_{ m JMP}$	Ins	carga Ins en PC
$_{ m JEQ}$	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 $Z=0$
$_{ m JGE}$	Ins	carga Ins en PC si en el status $N=0$
$_{ m JLT}$	Ins	carga Ins en PC si en el status $N=1$
$_{ m JLE}$	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	guarda A en Mem[SP] y decrementa SP
	В	guarda B en Mem[SP] y decrementa SP
POP	A	incrementa SP y luego guarda Mem[SP] en A
	В	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	guarda Input[Lit] en A
	B,Lit	guarda Input[Lit] en B
	(B),Lit	guarda Input[Lit] en Mem[B]
OUT	A,B	envia A a Output[B]
	A,(B)	envia A a Output[Mem[B]]
	A,(Dir)	envia A a Output[Mem[Dir]]
	A,Lit	envia A a Output[Lit]