DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

**IIC2343-2 - Arquitectura de Computadores (I/2022)**

# Assembler de Proyecto

**Ensamblador para su computador básico**

## 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 `.txt`

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:

```
# python3 assembler.py code.txt
```

## Funcionamiento

El programa debe recibir la ubicación de un archivo de formato `.txt`, dentro del cual hay un programa en el lenguaje *assembly* de su proyecto, y debe traducirlo a al código de máquina para su ROM. Luego por medio del puerto serial de la Basys3 debe programar la ROM. Para este último paso sus ayudantes les prepararon una librería publicada en PyPI que pueden instalar a través de consola con:

```
# pip install iic2343
```

Esta librería tiene el código necesario para comunicarse por medio del puerto serial con el componente Proggramer que se encuentra dentro de sus proyectos, el cual se encargará de detener su CPU, programar ROM y, al terminar, reiniciar su CPU.

Ejemplo:

```python
from iic2343 import Basys3

rom_programmer = Basys3()

if __name__ == '__main__':

    # Begin serial proggraming (stops the CPU and enables proggraming)
    rom_programmer.begin()

    # 12 bits address: 0, 36 bits word: 0x000301601
    rom_programmer.write(0, bytearray([0x0, 0x00, 0x30, 0x16, 0x01]))

    # 12 bits address: 1, 36 bits word: 0x000301803
    rom_programmer.write(1, bytearray([0x0, 0x00, 0x00, 0x18, 0x03]))

    # 12 bits address: 2, 36 bits word: 0x000201803
    rom_programmer.write(2, bytearray([0x0, 0x00, 0x20, 0x18, 0x03]))

    # 12 bits address: 3, 36 bits word: 0x000002000
    rom_programmer.write(3, bytearray([0x0, 0x00, 0x00, 0x20, 0x00]))

    # End serial proggraming (restarts CPU)
    rom_programmer.end()
```

Alternativamente, si lo necesitan, pueden escribir también un archivo `ROM.vhd` probar si su assembler está traduciendo correctamente las instrucciones. A continuación un ejemplo de la ROM de 4096 palabras de 36 bits que tiene solo 9 instrucciones:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

entity ROM is
    Port (
        clk       : in  std_logic;
        write     : in  std_logic;
        disable   : in  std_logic;
        address   : in  std_logic_vector(11 downto 0);
        datain    : in  std_logic_vector(35 downto 0);
        dataout   : out std_logic_vector(35 downto 0)
          );
end ROM;

architecture Behavioral of ROM is

type memory_array is array (0 to ((2 ** 12) - 1) ) of std_logic_vector (35 downto 0);

signal memory : memory_array:= (
        "000000000000000000010000111000000010",    -- instruccion 1
        "000000000000000000000000011000000011",    -- instruccion 2
        "000000000000000001000000111000000010",    -- instruccion 3
        "000000000000000001000000011000000011",    -- instruccion 4
        "000000000000000001100000111000000010",    -- instruccion 5
        "000000000000000001000000011000000011",    -- instruccion 6
        "000000000000000010000000111000000010",    -- instruccion 7
        "000000000000000011000000011000000011",    -- instruccion 8
        "000000000000000000000000111000000010",    -- instruccion 9
        "000000000000000000000000000000000000",    -- el resto de las
        "000000000000000000000000000000000000",    -- instrucciones estan
        "000000000000000000000000000000000000",    -- en blanco
        "000000000000000000000000000000000000",
```

```
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",    -- instruccion 4095
        "00000000000000000000000000000000"     -- instruccion 4096
        );

begin

process (clk)
    begin
       if (rising_edge(clk)) then
            if(write = '1') then
                memory(to_integer(unsigned(address))) <= datain;
            end if;
       end if;
end process;

with disable select
    dataout <= memory(to_integer(unsigned(address)))  when '0',
            (others => '0') when others;

end Behavioral;
```

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 en la palabra de 36 bits queda a criterio de cada grupo.

## 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 responsabilidad del *assembler* asignarles direcciones en la RAM agregando las instrucciones que sean necesarias. 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 reemplazos en las instrucciones según corresponda.

```
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.

### Etapa 2

Para la etapa 2 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 instrucciones.

```
// Esto es un comentario
       DATA:
// Linea en blanco
 v1     10              // 10 se asume decimal
        v2      10d     // 10 en decimal
 v3     10b             // 10 en binario
 v4     10h             // 16 en hexadecimal

  CODE:
 MOV B, ( v4    )       // B = Mem[3] = 16
 MOV    A, ( 10b        )       // A = Mem[2] = 2

        label1:
        MOV (v1),B      // Mem[0] = 16
        JMP label2      // Salta a label2
1end:

  label2:

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.

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

## Etapa 3

Para la etapa 3 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.

```
DATA:
 letrac 'c'     // 99
 string "ho la" // ['h', 'o', ' ', 'l', 'a', 0] = [104, 111, 32, 108, 97, 0]
CODE:
 MOV A,(letrac) // A = 99
 SUB A,'a'      // A = A - 97 = 2
```

## Assembly

| | | |
|---|---|---|
| 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 |
| | B | 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] |
| 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 | guarda A en Mem[SP] y decrementa SP |
| | B | guarda B en Mem[SP] y decrementa SP |
| POP | A | incrementa SP y luego guarda Mem[SP] en A |
| | B | 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 |