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

IIC2343 - Arquitectura de Computadores (II/2024)

Assembler de Proyecto

Ensamblador para su computador básico

Objetivos

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

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

De no realizar un archivo ejecutable, el *assembler* puede estar escrito en *Python3*, y se deberá poder ejecutar a través de la 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 que habrá un programa en el lenguaje *assembly* de su proyecto. Luego, debe traducirlo al código de máquina para su *ROM*. Posteriormente, por medio del puerto serial de la *Basys3*, debe programar la *ROM*. Para este último paso, sus ayudantes les prepararon una librería publicada y documentada en PyPI, que pueden instalar a través de consola mediante del siguiente comando:

```
# pip install iic2343
```

Esta librería tiene el código necesario para comunicarse con el componente *Programmer* que se encuentra dentro de sus proyectos por medio de un puerto serial en el *USB* de la placa. Este componente se encargará de detener su *CPU*, programar la *ROM*, y al terminar, reiniciar todos los registros/contadores e iniciar 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, se recomienda también escribir un archivo `ROM.vhd` para probar si su *assembler* está traduciendo correctamente las instrucciones. A continuación, se muestra 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",
```

```
        "000000000000000000000000000000000000",
        "000000000000000000000000000000000000",    -- instruccion 4095
        "000000000000000000000000000000000000"     -- 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* al archivo `ROM.vhd` 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 una sección *DATA:* al comienzo del código. Esta sección será la única en donde se definirán las variables de sus programas. Estas variables deben almacenarse en la RAM en direcciones asignadas por el *assembler* al comenzar el programa, utilizando las instrucciones necesarias para poder lograrlo. Cada variable declarada en una línea sigue el formato *nombre valor*. Su *assembler* debe recordar estos nombres y sus direcciones en la RAM y reemplazarlos en las instrucciones según corresponda.

Luego, la línea *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 deben poder definir *labels* como una palabra seguida inmediatamente por dos puntos en una línea 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 *assembler* **NO** puede necesitar elementos adicionales como una línea *END* al final del código para poder ensamblar. Recuerde que la CPU solo opera con números positivos de 16 bits, por lo que no debe soportar números negativos.

Etapa 1

Para la etapa 1, usted deberá implementar una primera versión de su *assembler*, con menos capacidades y menos restricciones. En esta etapa, es necesario que su *assembler* reconozca solo las instrucciones que fueron implementadas en la etapa 1. Específicamente, su *assembler* debe cumplir las siguientes características:

- Se debe aceptar **a lo menos** números en base 10.
- **No debe** interpretar una sección DATA.
- Debe interpretar correctamente la primera línea del archivo, que siempre será "CODE:".
- **No debe** reconocer *labels*, es decir, todas las direcciones de memoria que se entregarán serán números decimales.
- Interpretar un .txt con todo "bonito", es decir, solo contendrá espacios, no tendrá comentarios y las instrucciones se escribirán sin espacios entre sus argumentos. Es decir, los archivos se verán de la siguiente manera:

```
CODE:
MOV A,5
MOV (0),A
MOV B,(0)
CMP A,B
JEQ 2
```

Cabe mencionar que, a pesar de que las instrucciones sean que **no se debe** implementar algo, usted es libre de hacerlo si quisiera adelantar trabajo de la etapa 2. Dicho esto, es su responsabilidad si su *assembler* no funciona correctamente por intentar implementar algo extra a lo que se le pide.

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 línea usando // como delimitador.
- Espacios y tabulaciones en distintas partes del código, además de líneas en blanco entre instrucciones.

```
// Esto es un comentario
        DATA:
// Línea 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
```

- Aceptar el nombre de una variable como literal de su dirección en la RAM.
- Aceptar el uso explícito de literales en instrucciones de direccionamiento directo.

```
DATA:
  var1 1   ; Se almacena en 0x00 = 0
  var2 2   ; Se almacena en 0x01 = 1
CODE:
  MOV A,(var1)   ; A = Mem[var1] = 1
  MOV B,(1)      ; B = Mem[1] = Mem[var2] = 2
```

- Definición de arreglos de variables declarando una lista de valores nombrando solo al 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

En esta etapa, es crucial que hayan logrado todo lo pedido en las etapas 1 y 2, ya que se correrá un test para evaluar específicamente su *assembler*. En este test, se evaluará que su *assembler* cumpla con todas las características pedidas tanto en la etapa 1 como en la etapa 2 y será publicado en su debido momento para que lo puedan revisar y verificar que cumplen lo pedido.

Opcionalmente, si para esta etapa van a usar una pantalla LCD, su *assembler* además 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 el valor de B en A
	B,A	Guarda el valor de A en B
	A,Lit	Guarda un literal Lit en A
	B,Lit	Guarda un literal Lit en B
	A,(Dir)	Guarda el valor de Mem[Dir] en A
	B,(Dir)	Guarda el valor de Mem[Dir] en B
	(Dir),A	Guarda el valor de A en Mem[Dir]
	(Dir),B	Guarda el valor de B en Mem[Dir]
	A,(B)	Guarda el valor de Mem[B] en A
	B,(B)	Guarda el valor de Mem[B] en B
	(B),A	Guarda el valor de A en Mem[B]
	(B),Lit	Guarda un literal Lit en Mem[B]
ADD SUB AND OR XOR	A,B	Guarda el resultado de A op B en A
	B,A	Guarda el resultado de A op B en B
	A,Lit	Guarda el resultado de A op Lit en A
	B,Lit	Guarda el resultado de A op Lit en B
	A,(Dir)	Guarda el resultado de A op Mem[Dir] en A
	B,(Dir)	Guarda el resultado de A op Mem[Dir] en B
	(Dir)	Guarda el resultado de A op B en Mem[Dir]
	A,(B)	Guarda el resultado de A op Mem[B] en A
	B,(B)	Guarda el resultado de A op Mem[B] en B
NOT SHL SHR	A	Guarda el resultado de op A en A
	B,A	Guarda el resultado de op A en B
	(Dir),A	Guarda el resultado de op A en Mem[Dir]
	(B),A	Guarda el resultado de op A en Mem[B]
INC	A	Incrementa el valor de A en una unidad
	B	Incrementa el valor de B en una unidad
	(Dir)	Incrementa el valor de Mem[Dir] en una unidad
	(B)	Incrementa el valor de Mem[B] en una unidad
DEC	A	Decrementa el valor de A en una unidad
CMP	A,B	Ejecuta la instrucción SUB A,B sin actualizar el valor de A
	A,Lit	Ejecuta la instrucción SUB A,Lit sin actualizar el valor de A
	A,(Dir)	Ejecuta la instrucción SUB A,(Dir) sin actualizar el valor de A
	A,(B)	Ejecuta la instrucción SUB A,(B) sin actualizar el valor de A
JMP	Ins	Carga la dirección de la instrucción Ins en PC
JEQ	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple Z = 1
JNE	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple Z = 0
JGT	Ins	Carga la dirección de la instrucción Ins en PC ssi en Status se cumple N = 0 y Z = 0
JGE	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple N = 0
JLT	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple N = 1
JLE	Ins	Carga la dirección de la instrucción Ins en PC Ins si en Status se cumple N = 1 o Z = 1
JCR	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple C = 1
NOP		No hace cambios
PUSH	A	Guarda el valor de A en Mem[SP] y decrementa SP en una unidad
	B	Guarda el valor de B en Mem[SP] y decrementa SP en una unidad
POP	A	IncrementaSP en una unidad y luego guarda el valor de Mem[SP] en A
	B	Incrementa SP en una unidad y luego guarda el valor de Mem[SP] en B
CALL	Ins	Guarda PC+1 en Mem[SP], carga la dirección de la instrucción Ins en PC y decrementa SP en una unidad
RET		Incrementa SP en una unidad y luego carga el valor de Mem[SP] en PC