Maia Alonso, Gonzalo Benzaquen y Julieta Kovalivker

Trabajo Práctico I - Tecnología Digital II

Ejercicio 2:

a) Previamente a ejecutar el programa, describir con palabras el comportamiento esperado del mismo. No se debe explicar instrucción por instrucción, la idea es entender que hace el programa y que resultado genera.

La parte principal del programa funciona como un bucle. Mientras que el valor guardado en R3 y R2 no sean iguales, se suma de a uno a R2 (que comienza siendo 0x50) hasta que alcanza el valor de R3 (0x70). Por cada iteración del bucle, se guarda en la memoria desde la posición 0x50, la resta de 0xFF y el valor en R2. Así, quedan almacenados en la memoria los números enteros desde 0xAF hasta 0x90, es decir [(0xFF - 0x50); (0xFF - 0x6F)]. El bucle itera 32 veces.
Cuando R2 llega a 0x70, se realiza la comparación que activa la flag de Zero y hace un jump a la etiqueta *fin*. Aquí, se vuelve a hacer un jump y se regresa a la primera instrucción. Es así que el programa se ejecuta infinitamente.

b) Identificar la dirección de memoria de cada una de las etiquetas del programa.

- main = 0x00
- aca = 0x0A
- coso2 = 0x14
- fin = 0x20
- halt = 0x22

c) Ejecutar e identificar de ser posible cuantos ciclos de clock son necesarios para que el programa llegue a la instrucción JMP halt.

Justo antes de llegar a la instrucción “JMP halt” se declaran “DB 0xA0” y “DB 0x00” que, al buscar la instrucción de 16 bits asociada en la datasheet, equivale a “JMP 0x00”. Por ende, el programa nunca llega a la instrucción “JMP halt”.

d) ¿Cuántas microinstrucciones son necesarias para ejecutar la instrucción ADD? ¿Cuántas para la instrucción JZ? ¿Cuántas para la instrucción JMP?

Para ejecutar la instrucción ADD son necesarias 5 microinstrucciones, para ejecutar JZ son necesarias 4 y para ejecutar JMP son necesarias 2.

e) ¿El programa utiliza la pila?, ¿Qué datos son almacenados en la pila?

Si. Podemos ver que el programa claramente utiliza la pila ya que utiliza las instrucciones PUSH, POP, CALL y RET con |R7| (el puntero de pila) las cuales interactúan con ella.
Durante la ejecución del programa se guardan en la pila:

- Al hacer “CALL |R7|, coso2” se guarda en la dirección *0xFF* el valor *0x0E,* la dirección actual del Program Counter.
- Al hacer “PUSH |R7|, R0” se guarda en la dirección *0xFE* el valor *0x00* (El valor actual del registro R0)

f) Describir detalladamente el funcionamiento de las instrucciones PUSH, POP, CALL y RET.

PUSH |Rx|, Ry
La instrucción PUSH sirve para agregar el valor de un registro a la pila. Recibe dos parámetros: el primero (Rx) es el registro que posee el puntero de tope de la pila, el segundo (Ry) es el registro con el valor que se desea apilar.
Para lograr esto, cada vez que se agrega un valor a la pila, se le resta uno a Rx para que así, ahora el puntero de la pila corresponda al lugar en memoria siguiente al ya utilizado.
Los pasos para realizar esto son:

1. Se guarda el valor del registro Rx en ADDR
2. Se escribe el valor de Ry en |Rx|, o sea, en la pila (la posición en memoria guardada en ADDR)
3. Se guarda en el A de entrada de la ALU el valor de Rx
4. Se guarda en el B de entrada de la ALU un 1
5. Se escribe en RX la resta de la ALU (RX - 1)
6. Finaliza la instrucción

POP |Rx|, Ry

La instrucción POP sirve para quitar el último valor de la pila y guardarlo en otro registro. Recibe dos parámetros: el primero (Rx) es el registro que posee el puntero de tope de la pila, el segundo (Ry) es el registro en el que se desea guardar el valor quitado de la pila.
Para lograr esto, cada vez que se desea quitar un valor de la pila, se le suma uno a Rx para que así, ahora el puntero de la pila corresponda al último valor agregado, o sea, el que se desea quitar.
Los pasos para realizar esto son:

1. Se guarda en el A de entrada de la ALU el valor de Rx
2. Se guarda en el B de entrada de la ALU un 0x01
3. Se guarda en Rx la suma de la ALU (RX + 0x01)
4. Se guarda el valor del registro Rx en ADDR
5. Se guarda el valor de la dirección |Rx| (la pila) en Ry
6. Finaliza la instrucción

CALL |Rx|, Ry ó |Rx|, M
La instrucción CALL sirve para poder saltar de una instrucción a otra guardando en la pila el valor del PC donde fue hecho el salto, para luego, poder volver a esa instrucción. Recibe dos parámetros: el primero (Rx) es el registro que posee el puntero de tope de la pila, el segundo (Ry o M) es a donde se quiere que apunte el PC.
Como cada vez que se llama a CALL se debe agregar a la pila el valor del PC, se le suma uno a Rx (como fue explicado en el PUSH).
Los pasos para realizar esto son:

1. Se guarda el valor del registro Rx en ADDR
2. Se escribe el valor de PC (Ry o M) en |Rx|, o sea, en la pila (la posición en memoria guardada en ADDR)
3. Se guarda en el A de entrada de la ALU el valor de Rx
4. Se guarda en el B de entrada de la ALU un 0x01
5. Se escribe en RX la suma de la ALU (RX + 0x01)
6. Se guarda en el PC el valor del Ry
7. Finaliza la instrucción

RET |Rx|

La instrucción RET sirve para que, una vez ya usada la instrucción CALL, se pueda volver a donde fue utilizado el CALL. Recibe un solo parámetro, Rx, que es el registro que posee el puntero de tope de la pila. Es importante que el último valor de la pila sea la posición en memoria de la instrucción a la que queremos volver.
Los pasos para realizar esto son:

1. Se guarda en el A de entrada de la ALU el valor de Rx
2. Se guarda en el B de entrada de la ALU un 0x01
3. Se escribe en RX la suma de la ALU (RX + 0x01)
4. Se guarda el valor del registro Rx en ADDR
5. Se guarda el valor de la dirección |Rx| (la pila) en el PC
6. Finaliza la instrucción

Ejercicio 3:

a. Agregar la instrucción ADDINMEM

- Ver la operación con código **01110** del archivo *./Ejercicio3/microCode.ops*

b. Agregar la instrucción ADDE10S

- Ver la operación con código **01111** del archivo *./Ejercicio3/microCode.ops*
- Ver la operación 14 de la ALU implementada en el archivo *Ejercicio3/orgaSmallModificado.circ*

Ejercicio 4:

a. Utilizar la instrucción ADDINMEM

- Ver implementación en el archivo *./Ejercicio4/IncisoA/ej4a.asm*

b. Utilizar la instrucción ADDE10S

- Ver implementación en el archivo *./Ejercicio4/IncisoB/ej4b.asm*

ACLARACIONES ADICIONALES:

- Para poder construir programas con las instrucciones agregadas, modificamos el archivo *common.py* dando soporte a ADDINMEM y ADDE10S. (Puede verse en el archivo *./Ejercicio4/tools/common.py*)

- El directorio *./Ejercicio3/* contiene una carpeta *Tests* con un archivo utilizado para probar la implementación de ADDINMEM. Este archivo puede ser ignorado.
- Al intentar compilar las implementaciones de los ejercicios 4a y 4b, nos surgían errores por causa de los espaciados e indentados. Por eso, también incluimos una versión de ambos archivos sin comentarios, espacios e indentados. Ambos directorios (*IncisoA* e *IncisoB*) contienen una carpeta *PRUEBA* con dicha implementación.