Trabajo Práctico 1 Tecnología Digital II

Lara Fainburg, Solana Belinky, Catalina Lavallen

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.

Al principio se asignan valores a los registros R7-R0 a cada uno respectivamente.
Luego, se comparan los registros R3 y R2, y en el caso de que sean iguales, se salta a fin y termina. En caso contrario, se pasa al siguiente (call), que lo que hace es saltar a "coso2", en donde se actualiza el valor de R2 sumándole R1, y así sucesivamente creando un ciclo hasta que ambos sean iguales, cuando eso se cumple, pasa a fin. También guarda la dirección de retorno en la pila (usando R7 como puntero).
Es por eso que el ciclo se repite hasta que R2 sea igual a R3, que este es el momento en el que el programa salta a fin y entra en un ciclo infinito en halt, por eso decimos que se "traba" ahí.
En caso de que R2 sea mayor que R3, hace lo mismo, pero restando para que sean iguales, para llegar al resultado esperado (R2=R3).
En coso2 se inicia guardando la dirección de la memoria de R7 en R0, para después almacenar el valor de R7 (0xff). luego de esto, al contenido de R0 (0xff) se le resta el contenido de R2 (0x50), lo que nos da la operación (205) que se va a guardar en R2. para terminar se recupera el valor de la pila (que se había almacenado anteriormente), esto nos regresa al dónde empezamos.

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

main: **|00|**
    |00| SET R7 , 0xFF
    |02| SET R2 , 0x50
    |04| SET R3 , 0x70
    |06| SET R1 , 0x01
    |08| SET R0 , 0x00

aca: **|0a|**
    |0a| CMP R3 , R2
    |0c| JZ fin
    |0e| CALL | R7 | , coso2
    |10| ADD R2 , R1
    |12| JMP aca

coso2: **|14|**
    |14| PUSH | R7 | , R0
    |16| SET R0 , 0xFF
    |18| SUB R0 , R2

|1a| STR [R2] , R0
|1c| POP | R7 | , R0
|1e| RET | R7 |

fin: **|20|**
|20| DB 0xA0
|21| DB 0x00

halt: **|22|**
|22| JMP halt

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

```
11110: ; SET
1era RB_enIn RB_selectIndexIn=0 DE_enOutImm ; Rx <- M
2da reset_microOp
```

Los ciclos que se van a ejecutar para llegar a la instrucción JMP halt va a ser infinitos lo que impide que lleguen a JMP halt, ya que se queda “trabado” ahí como dijimos en el punto a.

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

→ Para llegar a JZ son necesarias 4 microinstrucciones
(no estamos contando el fetch)

```
 10110: ; JZ
 1era JZ_microOp load_microOp ; if Z then microOp+2 else microOp+1
 2da  reset_microOp
 3ra  PC_load    DE_enOutImm  ; PC <- M
 4ta  reset_microOp
```

ADD —> 5 microinstrucciones
→ Para llegar a ADD son necesarias 5 microinstrucciones
(no estamos contando el fetch)

```
00001: ; ADD
 1ra  ALU_enA RB_enOut RB_selectIndexOut=0 ; A <- Rx
 2da  ALU_enB RB_enOut RB_selectIndexOut=1 ; B <- Ry
 3ra  ALU_OP=ADD ALU_opW
 4ta  RB_enIn RB_selectIndexIn=0 ALU_enOut ; Rx <- Rx + Ry
 5ta  reset_microOp
```

JMP → 2 microinstrucciones
→ Para llegar a JMP son necesarias 2 microinstrucciones
(no estamos contando el fetch)

```
10100: ; JMP
 1ra  PC_load    DE_enOutImm   ; PC <- M
 2da  reset_microOp
```

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

El programa utiliza la pila para guardar y recuperar datos, esto lo podemos ver más que nada en la parte de coso2 con las instrucciones de PUSH, POP, y RET. En este caso, los datos que son almacenados en la pila son R0, R7.

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

Las instrucciones PUSH, POP, CALL, RET son las únicas que operan con la pila, esta es direccionada desde un registro pasado por parámetro**.**

Las instrucciones PUSH y POP van de la mano.
→ **PUSH:** La instrucción PUSH se utiliza para guardar un valor en la pila. Para hacer esto ocurren dos cosas en simultáneo. Por un lado se guarda el valor en la memoria, para eso la instrucción toma el valor de un registro y lo guarda en la dirección de memoria apuntada por el puntero de pila. Al mismo tiempo se resta 1 al puntero de la pila lo que se guarda en el registro, que no es lo mismo que la memoria.
Rx y Ry nunca cambian de valor, lo que cambia es su posición de memoria.

PUSH |Rx|, Ry	Mem[Rx] ← Ry ; Rx ← Rx-1

→ **POP:** La instrucción POP se utiliza para recuperar el valor más recientemente almacenado en la pila (los valores se recuperan en “sentido contrario” del que se guardaron en un principio con el PUSH). Para hacer esto, al igual que en la instrucción PUSH tienen que ocurrir dos cosas en simultáneo. Por un lado se incrementa en 1 el puntero de la pila, para que apunte al primer lugar ocupado, lo que apunta a Ry. A su vez se guarda el valor de la memoria en un registro (Rx)

POP |Rx|, Ry	Rx ← Rx+1 ; Ry ← Mem[Rx]

Las instrucciones CALL y RET van de la mano.
→ **CALL:** Guarda el valor actual del Contador de Programa (PC). Guarda la dirección actual en la pila y salta a una nueva dirección de memoria. El valor actual del contador de programa (PC) contiene la dirección de la siguiente instrucción a ejecutar. Osea, estamos poniendo el valor de PC (que es un registro más) en Rx.
Cuando ya se almacenó el valor de PC en la pila, se le resta 1 a Rx. Esto se hace para que el puntero cambie y apunte a la siguiente posición.

CALL |Rx|, M	Mem[Rx] ← PC ; Rx ← Rx-1 ; PC ← M

→ **RET:** Lo que hace esta instrucción es recuperar la dirección de memoria guardada anteriormente en la instrucción CALL para reanudar la ejecución del programa desde donde se dejó y que los registros terminen con el mismo valor con el que arrancaron. Lo que hace es aumentar el puntero de la pila para encontrar la dirección de memoria donde se guardó el valor original del pc. A su vez se vuelve a restaurar la dirección que tienen la instrucción

siguiente a la call, lo que hace que la ejecución del programa continúe justo después de la llamada a la función.

Comentarios extra del trabajo:

Ejercicio 3B (circuito que implementamos en la ALU):

Al operador a 14 (cte 0x02) le entran dos variables, A y B, y da una salida que está conectada al multiplexor de la ALU.
Lo que hace este operador es sumar dos números en exceso-10. El circuito lo que hace es al comienzo, tomar las dos entradas y a cada una por separado restarle diez (por estar en notación exceso-10). Al resultado de ambas restas lo sumo, para después sumarle 10 devuelta, lo que nos da la suma de esos dos números. Para cumplir las condiciones, lo que hacemos es usar multiplexores para decidir qué es lo que va a hacer frente a diferentes situaciones en donde la suma pase el máximo representable o cero dependiendo si es negativo o positivo el resultado.

Foto del circuito agregado a la ALU (el archivo con el circuito completo está adjuntado en la entrega):