Tecnología Digital II: Sistemas de Computación TP I

Integrantes: Belinky Solana, Fainburg Lara, Leis Guadalupe.

Ejercicio 2)

a. El programa hace una secuencia de sumas acumulativas desde el número uno hasta el diez. Su funcionamiento empieza con inicializar registros que definen el contador de iteraciones, el valor a sumar en cada paso, el límite del ciclo y otras variables auxiliares. Luego entra en un ciclo que se repite donde, en cada iteración, se va sumando uno al contador y se llama a una etiqueta que actualiza un valor acumulado.

Esta etiqueta lee el valor acumulado actual desde una dirección de memoria, le suma el valor correspondiente al número de iteración y guarda el nuevo resultado tanto en esa misma dirección como en otra dirección variable que va cambiando con cada paso. Así se va construyendo una lista en memoria con los resultados que fue acumulando en cada suma.

El ciclo se repite hasta completar nueve pasos, y ahí es donde el programa finaliza (9 iteraciones). Como resultado, al final de la ejecución la memoria va a tener en una dirección el total de la suma de los números del uno al nueve y, a partir de otra dirección, una secuencia con todos los resultados parciales obtenidos en cada vuelta del ciclo.

b. Como vemos en la siguiente imagen, cada etiqueta del código contiene una dirección de memoria correspondiente al lugar que ocupa en ella. La etiqueta *main* se encuentra en la posición 0x00 (hexadecimal). A su vez, *ciclo* se encuentra en la posición 0x0A (hexadecimal) y *completar* está ubicada en la posición 0x14. Por último, *fin* se encuentra posicionada en la dirección de memoria 0x2A y la etiqueta *halt* en la dirección de memoria 0x2C

```
     main |00| SET R7 , 0xFF
          |02| SET R3 , 0x01
          |04| SET R2 , 0x0A
          |06| SET R1 , 0x01
          |08| SET R0 , 0x00
    ciclo |0a| ADD R3 , R1
          |0c| CALL | R7 | , completar
          |0e| CMP R3 , R2
          |10| JZ fin
          |12| JMP ciclo
completar |14| PUSH | R7 | , R0
          |16| PUSH | R7 | , R1
          |18| LOAD R0 , [ 0xA0 ]
          |1a| ADD R0 , R1
          |1c| STR [ 0xA0 ] , R0
          |1e| SET R1 , 0xB0
          |20| ADD R1 , R3
          |22| STR [ R1 ] , R0
          |24| POP | R7 | , R1
          |26| POP | R7 | , R0
          |28| RET | R7 |
      fin |2a| DB 0xB8
          |2b| DB 0x00
     halt |2c| JMP halt
```

c. Teniendo en cuenta que cada clock son dos clicks (flanco ascendente y descendente), contamos alrededor de 2400 clicks, lo que representan aproximadamente 1200 ciclos hasta llegar a la instrucción JMP halt. En el momento que se llega a esta instrucción, el programa entra en un bucle infinito que mantiene detenido el programa, evitando que continúe con su ejecución.

d.

```
00001: ; ADD
    ALU_enA RB_enOut RB_selectIndexOut=0 ; A <- Rx
    ALU_enB RB_enOut RB_selectIndexOut=1 ; B <- Ry
    ALU_OP=ADD ALU_opW
    RB_enIn RB_selectIndexIn=0 ALU_enOut ; Rx <- Rx + Ry
    reset_microOp
```

La operación ***ADD*** requiere de 5 microinstrucciones para realizar su función, teniendo en cuenta la microinstrucción reset.

```
10110: ; JZ
    JZ_microOp load_microOp ; if Z then microOp+2 else microOp+1
    reset_microOp
    PC_load    DE_enOutImm  ; PC <- M
    reset_microOp
```

La operación ***JZ*** requiere de 4 microinstrucciones contando ambos reset, para realizar su función.

```
10100: ; JMP
    PC_load    DE_enOutImm   ; PC <- M
    reset_microOp
```

La operación ***JMP*** requiere de 2 microinstrucciones incluyendo el reset para realizar su función.

e. Sí, el programa utiliza la pila. Esto lo podemos ver en las instrucciones *CALL, PUSH, POP* y *RET*. A través de las instrucciones mencionadas, la pila usa el registro R7 como Stack Pointer (puntero). Cuando se ejecuta CALL, se guarda en la pila la dirección de retorno para poder volver desde la subrutina con RET. Asimismo, se almacenan los valores de los registros R0 y R1 mediante PUSH, y luego se recuperan con POP. El uso de la pila nos permite conservar los valores iniciales de los registros mientras se usan temporalmente dentro de la subrutina.

f.

→ CALL: Guarda el valor actual del Contador de Programa (PC) en la pila. 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

→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, sino que lo que cambia es su posición de memoria.

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

→ JC: La instrucción JC realiza un salto condicional a una subrutina (etiqueta) solo sí el flag de carry está encendido (en 1), entonces el programa carga una nueva dirección en el contador del programa PC. Sí el flag C está apagado (en 0) continua con la instrucción siguiente.

JC M	Si flag_C=1 entonces PC ← M

→ LOADF: La instrucción LOADF toma como parámetro el contenido de un registro y copia el valor de los registros en binario en las flags correspondientes. Por ejemplo, sí hacemos la instrucción LOADF Rx, seteando a Rx en 0x0001, el resultado de la operación pondrá la flag Z en 1, y la flag C en 0. (Esto lo hará tomando el valor de Rx como una combinación binaria)

LOADF Rx	Flags ← Rx

---

3) a.

```
01110: ; CLEAN3

; [Rx] <- 0
   MM_enAddr      RB_enOut RB_selectIndexOut=0       ; addr <- Rx
   ALU_enA        ALU_OP=cte0x00                     ; A <- 0
   MM_load        ALU_enOut                          ; [Rx] <- 0


; [Rx+1] <- 0

   ALU_enA                RB_enOut RB_selectIndexOut=0   ; A <- Rx
   ALU_enB                ALU_enOut ALU_OP=cte0x01       ; B <- 1
   MM_enAddr                ALU_enOut ALU_OP=ADD         ; addr <- RX + 1
   ALU_enA                ALU_OP=cte0x00                 ; A <- 0
   MM_load                ALU_enOut                      ; [Rx+1] <- 0

; [Rx+2] <- 0

   ALU_enA                RB_enOut RB_selectIndexOut=0   ; A <- Rx
   ALU_enB                ALU_enOut ALU_OP=cte0x02       ; B <- 2
   MM_enAddr                ALU_enOut ALU_OP=ADD         ; addr <- RX + 2
   ALU_enA                ALU_OP=cte0x00                 ; A <- 0
   MM_load                ALU_enOut                      ; [Rx+2] <- 0
   reset_microOp
```

b)

01111: ; COLLATZ Rx, Ry

```
ALU_enA                 RB_enOut RB_selectIndexOut=0   ; A <- Rx (parte alta)
ALU_enB                 RB_enOut RB_selectIndexOut=1   ; B <- Ry (parte baja)
ALU_OP=11 ALU_opW
RB_enIn RB_selectIndexIn=0      ALU_enOut               ; Rx <- parte alta
ALU_OP=cte0xFF ALU_opW
RB_enIn RB_selectIndexIn=1      ALU_enOut               ; Ry <- parte baja
reset_microOp
```