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

IIC2343 - Arquitectura de Computadores (I/2026)

Interrogación 2 - Parte 1 - Sección 2

Pauta de evaluación

Pregunta 1: Computador básico con subrutinas (6 ptos.)

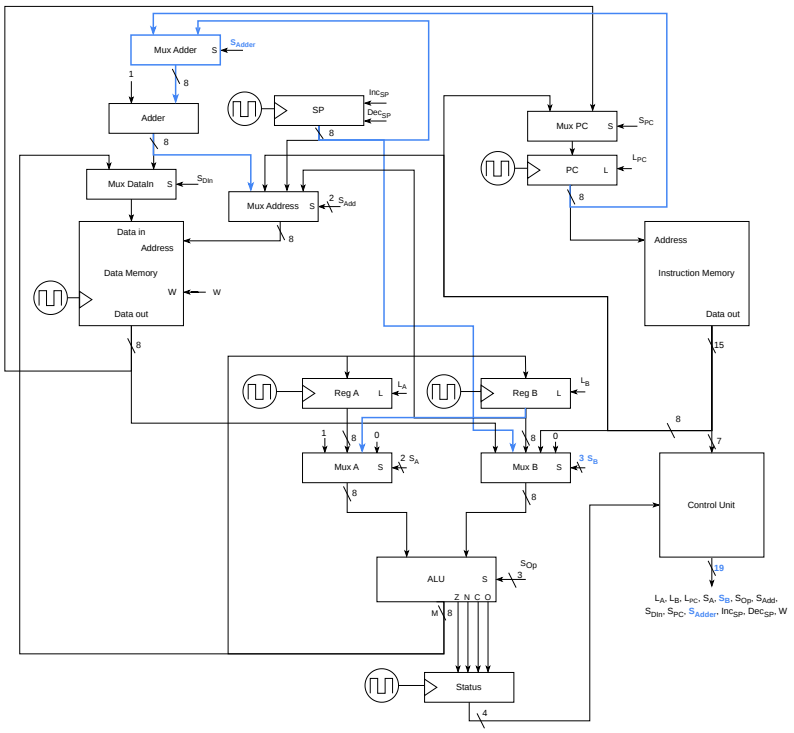
En los ítems **(a)**, **(b)** y **(c)**, se le describirán instrucciones que deberá implementar en el computador básico con subrutinas a partir del diagrama adjunto. Para cada una de ellas, debe incluir la combinación **completa** de señales que las ejecutan. En las señales de carga/escritura/incremento/decremento, debe indicar si se activan (1) o no (0); en las señales de selección, debe indicar el **nombre** de la entrada escogida ("-" si no afecta). Debe realizar todas las modificaciones de *hardware* necesarias en un solo diagrama. Además, puede implementar instrucciones sin modificar el *hardware* si es que la arquitectura se lo permite, indicando la forma de hacerlo a partir de las señales de control. **Cada instrucción debe ejecutarse en un ciclo**.

(a) (1 pto.) `PUSHRW A`. Sobreescribe el tope del *stack* con el valor del registro `A` sin modificar el *stack pointer*.

(b) (1 pto.) `CMP B, SP`. Computa la operación `B - SP` sin actualizar los registros.

(c) (1 pto.) `RET1`. Salta a la dirección de retorno almacenada en el tope del *stack* **en un solo ciclo**. En otras palabras, es la versión óptima de `RET` al realizar la misma operación en un ciclo y no en dos.

Solución: A continuación, se muestra el diagrama con las conexiones necesarias para ejecutar todas las instrucciones solicitadas, incluyendo una descripción de cada conexión añadida y lo que habilita.



- Se modifica la entrada del *Adder* que computa `PC+1` para añadir un multiplexor `Mux Adder` con señal de selección S_{Adder}. Este selecciona entre el valor de `PC` y el de `SP` para realizar la suma, lo que da la posibilidad de computar `PC+1` o `SP+1`.
- Se conecta la salida del *Adder* con la entrada restante del `Mux Address`, permitiendo seleccionar `PC+1` o `SP+1` (el que nos interesa) como dirección de la memoria de datos.
- Se conecta la salida de `B` con una entrada de `Mux A` y la salida de `SP` con una entrada de `Mux B`, de forma que puedan seleccionarse como primer y segundo operando de la ALU respectivamente. Esto requiere incrementar a 3 bits la señal S_B.

A continuación, la tabla de señales resultante:

Instrucción	L_A	L_B	L_{PC}	Inc_{SP}	Dec_{SP}	W	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	S_{Adder}
PUSHRW A	0	0	0	0	0	1	A	ZERO	ADD	ADDER	ALU	-	SP
CMP B, SP	0	0	0	0	0	0	B	SP	SUB	-	-	-	-
RET1	0	0	1	1	0	0	-	-	-	ADDER	-	DOUT	SP

Por cada instrucción, se otorgan:

- **0.5 ptos.** por *hardware* correctamente modificado; **0.3 ptos.** si existe **un solo error** de implementación; **0.1 ptos.** si existen **solo dos errores**; y **0 ptos.** en otro caso.
- **0.5 ptos.** por entregar una combinación de señales correcta; **0.3 ptos.** si existe **un solo error**; **0.1 ptos.** si existen **solo dos errores**; y **0 ptos.** en otro caso. Si una instrucción se define en más de un ciclo, tampoco se otorga puntaje en este criterio. Si la instrucción se implementa correctamente sin modificación de *hardware*, se otorga el puntaje completo.

(d) (3 ptos.) A continuación, se adjuntan dos fragmentos de código **(1)** y **(2)** escritos en el Assembly del computador básico, con la diferencia de que **(2)** incorpora las instrucciones implementadas en los ítems anteriores. Para cada fragmento, indique los valores de los registros `A` y `B` al finalizar la ejecución del código. Si no incluye desarrollo o no justifica cómo llegó a los valores obtenidos, **no se otorgará puntaje**.

(1)

```
DATA:
CODE:
  MOV A,2          ; Dir. memoria 0x00
  MOV B,5          ; Dir. memoria 0x01
  PUSH A           ; Dir. memoria 0x02
  PUSH B           ; Dir. memoria 0x03
  CALL mystery     ; Dir. memoria 0x04
  POP B            ; Dir. memoria 0x05-0x06
  POP A            ; Dir. memoria 0x07-0x08
  JMP end          ; Dir. memoria 0x09
  mystery:
    MOV A,0        ; Dir. memoria 0x0A
    NOT A,A        ; Dir. memoria 0x0B
    SUB A,2        ; Dir. memoria 0x0C
    MOV B,A        ; Dir. memoria 0x0D
    MOV A,(B)      ; Dir. memoria 0x0E
    ADD A,4        ; Dir. memoria 0x0F
    MOV (B),A      ; Dir. memoria 0x10
    RET            ; Dir. memoria 0x11-0x12
  end:
```

(2)

```
DATA:
  n    5
  res  0
CODE:
  MOV A,(n)          ; Dir. memoria 0x00
  CALL fun           ; Dir. memoria 0x01
  JMP end            ; Dir. memoria 0x02
  fun:
    CMP A,0          ; Dir. memoria 0x03
    JLE fun_end      ; Dir. memoria 0x04
    PUSH A           ; Dir. memoria 0x05
    MOV B,A          ; Dir. memoria 0x06
    AND A,1          ; Dir. memoria 0x07
    JLE rewrite      ; Dir. memoria 0x08
    continue:
      MOV A,B        ; Dir. memoria 0x09
      SUB A,1        ; Dir. memoria 0x0A
      JMP fun        ; Dir. memoria 0x0B
    rewrite:
      MOV A,0        ; Dir. memoria 0x0C
      PUSHRW A       ; Dir. memoria 0x0D
      JMP continue   ; Dir. memoria 0x0E
    fun_end:
      MOV B,254      ; Dir. memoria 0x0F
      loop:
        CMP B,SP     ; Dir. memoria 0x10
        JEQ end_loop ; Dir. memoria 0x11
        POP A        ; Dir. memoria 0x12-0x13
        ADD A,(res)  ; Dir. memoria 0x14
        MOV (res),A  ; Dir. memoria 0x15
        JMP loop     ; Dir. memoria 0x16
      end_loop:
        RET1         ; Dir. memoria 0x17
  end:
```

Solución: El fragmento de código **(1)**, a través de la subrutina `mystery`, **sobreescribe la dirección de retorno de su llamado**. Si bien este valor era 0x05 (`POP B`), este se lee con direccionamiento indirecto –`MOV A,(B)`–, se incrementa en 4 unidades –`ADD A,4`–, y luego se sobreescribe de vuelta –`MOV (B), A`–, generando que `RET` salte directamente a 0x09 (`JMP end`). Esto implica que el valor final del registro `A` sea igual a 9 (computado en 0x0F), y que el de `B` sea igual a 253 o -3, según interpretación (computado en 0x0D).

El fragmento de código **(2)**, por otro lado, hace lo siguiente: (i) llama a la subrutina `fun` y *pushea* los valores `n`, `n-1`, `n-2`, ..., 1; (ii) al *pushear*, revisa si el valor escrito es par –`AND A,1` < 1– y, en dicho caso, lo **sobreescribe por cero** –`PUSHRW A` en 0x0D–; (iii) al terminar de *pushear* y sobreescribir los pares, descarta el contenido del *stack* y acumula la suma de los elementos en `res` a través de `A`; (iv) una vez que los revisa todos –`SP` $= 254$, siendo el tope del *stack* la dirección de retorno de `fun`–, retorna y termina el programa. Por ende, `A` es igual a 9 ($5 + 3 + 1$, los impares del rango $[1, 5]$), mientras que `B` es igual a 254, ya que la última vez que actualiza su valor es con `MOV B,254` en 0x0F.

A continuación, se muestra el resultado de la ejecución para ambos fragmentos:

(1)

```
DATA:
CODE:
  MOV A,2          ; A = 2
  MOV B,5          ; B = 5
  PUSH A           ; MEM[255] = 2, SP = 254
  PUSH B           ; MEM[254] = 5, SP = 253
  CALL mystery     ; MEM[253] = 0x05, SP = 252
  POP B            ; No se ejecuta
  POP A            ; No se ejecuta
  JMP end          ; RET salta aquí -> Termina programa
  mystery:
    MOV A,0        ; A = 0
    NOT A,A        ; A = 11111111b = 255
    SUB A,2        ; A = 253
    MOV B,A        ; B = 253
    MOV A,(B)      ; A = MEM[253] = 5
    ADD A,4        ; A = 5 + 4 = 9
    MOV (B),A      ; MEM[253] = 9
    RET            ; SP = 253
                   ; PC = MEM[253] = 9 = 0x09
  end:             ; A = 9, B = 253
```

(2)

```
DATA:
  n    5
  res  0
CODE:
  MOV A,(n)           ; A = 5
  CALL fun            ; MEM[255] = 0x02
  JMP end             ; Termina el programa
  fun:
    CMP A,0           ; if A <= 0
    JLE fun_end       ;   -> Salta a fun_end
    PUSH A            ; MEM[254] = 5, MEM[253] = 4, ..., MEM[250] = 1 y SP = 249
    MOV B,A           ; B = 5, 4, 3, 2, 1
    AND A,1           ; 5 AND 1 = 1, 4 AND 1 = 0, 3 AND 1 = 1, 2 AND 1 = 0, 1 AND 1 = 1
    JLE rewrite       ; Si A es par, salta a rewrite
    continue:         ; e.o.c. es impar y continua sin reescribir
      MOV A,B         ; A = 5, 4, 3, 2, 1
      SUB A,1         ; A = 4, 3, 2, 1, 0
      JMP fun         ; Repetir hasta que A == 0
    rewrite:
      MOV A,0         ; A = 0
      PUSHRW A        ; MEM[253] = 0, MEM[251] = 0
      JMP continue    ; Salta a continue
    fun_end:
      MOV B,254       ; B = 254
      loop:
        CMP B,SP      ; if SP == B (se revisaron los elementos pusheados)
        JEQ end_loop  ;   -> Salta a end_loop
        POP A         ; A = MEM[250] y SP = 250, ..., A = MEM[254] y SP = 254
        ADD A,(res)   ; A += res (acumula suma en res)
        MOV (res),A   ; res = A (actualiza res)
        JMP loop      ; Repetir hasta que SP == 254
      end_loop:
        RET1          ; PC = MEM[255] = 0x02 (retorna a JMP end en un ciclo)
  end:                ; A = res = 5 + 3 + 1 = 9; B = 254 (última actualización con MOV B,254)
```

Para cada fragmento, se otorgan **0.75 ptos.** por registro correctamente computado siempre que su cómputo esté respaldado por desarrollo o justificación. Si no se llega a ningún resultado final correcto por **error de arrastre**, se otorgan **0.5 ptos.** en el fragmento.