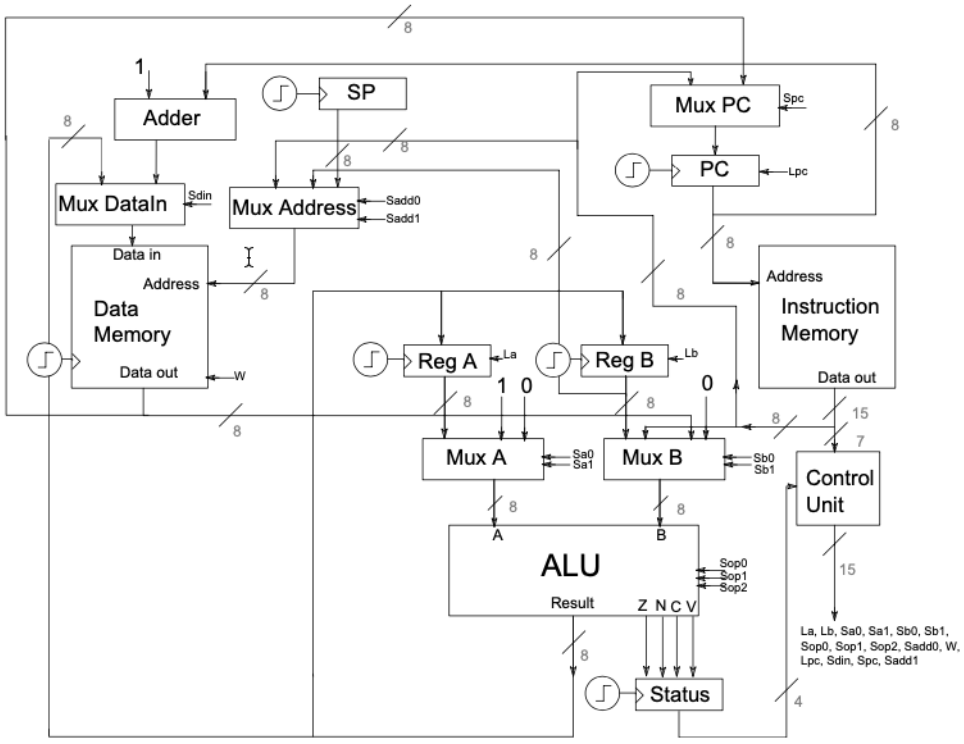
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

IIC2343 - Arquitectura de Computadores (II/2025)

Guía de ejercicios: Subrutinas

Ayudantes: Daniela Ríos (danielaarp@uc.cl), Alberto Maturana (alberto.maturana@uc.cl), Mario Rojas (mario.denzel@estudiante.uc.cl)

Computador básico con subrutinas



Pregunta 1: Computador básico (P4-I1-2024-1)

Asuma que se encuentra trabajando como programador(a) en la oficina del famoso juego desarrollado completamente en Assembly: *Rollercoaster Tycoon*. Luego de unas horas, se da cuenta que la letra T del teclado de su computador ha dejado de funcionar, la que es de suma importancia al momento de escribir subrutinas por el uso de la instrucción `RET`. Adicionalmente, no cuenta con la opción de copiar y pegar texto, por lo que deberá desistir de su uso. Por el motivo anterior, deberá diseñar tres nuevas instrucciones que le permitan simular parte del comportamiento de la instrucción `RET`.

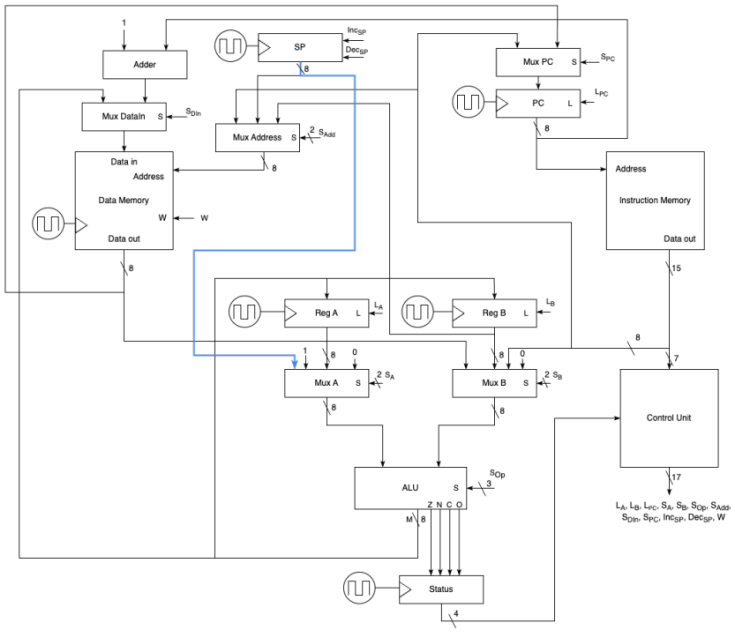
1. `MOV B, SP`: Almacena el valor del registro `SP` en el registro `B`.
2. `JMP B`: Salta a la instrucción con dirección igual al valor almacenado en el registro `B`.
3. `JMP (B)`: Salta a la instrucción con dirección igual al valor almacenado en la memoria de datos en la dirección `B`.

Deberá modificar la microarquitectura del computador básico para implementar las nuevas instrucciones y su funcionamiento. Para cada instrucción nueva, deberá incluir la combinación completa de señales que la ejecutan. Por cada señal de carga/escritura/incremento/decremento, deberá indicar si se activan (1) o no (0); en las señales de selección, deberá indicar el nombre de la entrada escogida ("-"si no afecta). Puede realizar todas las modificaciones en un solo diagrama.

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}
`MOV B, SP`												
`JMP B`												
`JMP (B)`												

1. `MOV B, SP`. Guarda en `B` el valor de `SP`.

Solución: Esta instrucción no se puede habilitar sin modificaciones de hardware. Una opción consiste en conectar la salida del contador SP con la entrada restante del componente Mux A, como se muestra a continuación:



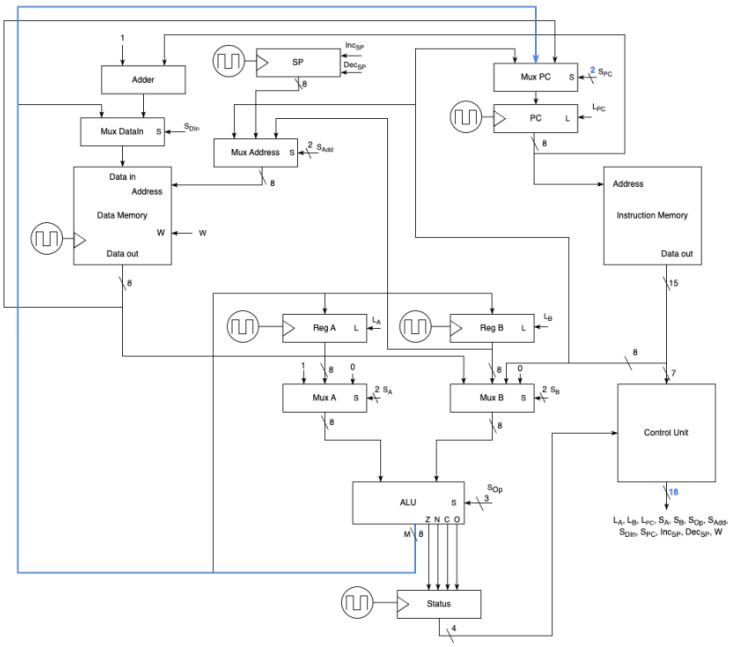
Para ejecutar la instrucción, se utiliza la siguiente combinación de señales:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}
`MOV B, SP`	0	1	0	0	0	0	SP	ZERO	ADD	-	-	-

Se otorgan **0.75 ptos.** por la correctitud de la modificación y **0.75 ptos.** por entregar una combinación de señales correcta para la instrucción. En caso de existir como máximo un error de implementación en uno de los criterios antes descritos, se otorga la mitad del puntaje. Si existe más de un error, no se otorga puntaje.

2. `JMP B`. Salta a la instrucción con dirección igual al valor `B`.

Solución: Esta instrucción no se puede habilitar sin modificaciones de hardware. Una opción consiste en conectar la salida de la ALU con una de las entradas del componente Mux PC (lo que implica agregar un bit a su señal de selección), como se muestra a continuación:



Para ejecutar la instrucción, se utiliza la siguiente combinación de señales:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}
JMP B	0	0	1	0	0	0	ZERO	B	ADD	-	-	ALU

Se otorgan **0.75 ptos.** por la correctitud de la modificación y **0.75 ptos.** por entregar una combinación de señales correcta para la instrucción. En caso de existir como máximo un error de implementación en uno de los criterios antes descritos, se otorga la mitad del puntaje. Si existe más de un error, no se otorga puntaje.

3. `JMP (B)`. Salta a la instrucción con dirección igual al valor `Mem[B]`.

Solución: Esta instrucción se puede implementar sin modificaciones de hardware, como se muestra a continuación:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}
JMP (B)	0	0	1	0	0	0	-	-	-	B	-	DOUT

Se otorgan **0.75 ptos.** por la correctitud de la modificación y **0.75 ptos.** por entregar una combinación de señales correcta para la instrucción. En caso de existir como máximo un error de implementación en uno de los criterios antes descritos, se otorga la mitad del puntaje. Si existe más de un error, no se otorga puntaje.

- Si no se modifica el diagrama: Se otorgan **1.5 ptos.** por entregar una combinación de señales correcta para la instrucción. En caso de existir como máximo un error de implementación, se otorga la mitad del puntaje. Si existe más de un error, no se otorga puntaje.
- Si se modifica el diagrama: Se otorgan **0.75 ptos.** por la correctitud de la modificación y **0.75 ptos.** por entregar una combinación de señales correcta para la instrucción. En caso de existir como máximo un error de implementación en uno de los criterios antes descritos, se otorga la mitad del puntaje. Si existe más de un error, no se otorga puntaje.

4. Asuma que se implementan las instrucciones anteriores y modifique el siguiente fragmento de código de forma que no utilice la instrucción `RET`, pero que se llegue al mismo resultado. Luego, comente sobre su resultado: ¿Cumple la misma función que el original? ¿Existe alguna consideración a tener en cuenta?

```
DATA:
    number_of_rides 0

CODE:
    JMP main

    // Incrementa la cantidad de veces a la que se sube a la montaña rusa
    increase_number_of_rides:
        MOV A, (number_of_rides)
        ADD A, 1
        MOV (number_of_rides), A
        RET

    main:
        PUSH A
        CALL increase_number_of_rides
        POP A
```

Solución: Para simular el comportamiento de `RET` con las instrucciones nuevas, existen dos alternativas:

a) Reemplazar `RET` por:

1) `POP B`
2) `JMP B`

En este caso, el código cumple la función original y no se requiere ningún arreglo adicional.

b) Reemplazar `RET` por:

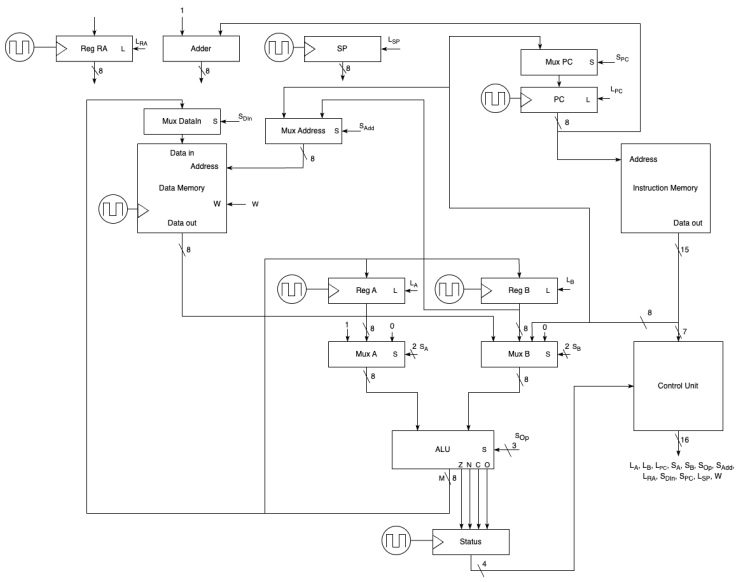
1) `MOV B, SP`
2) `INC B`
3) `JMP (B)`

En este caso, la ejecución no es exactamente igual a la original, dado que si bien se accede correctamente a la dirección de retorno, el valor de `SP` no es ajustado mediante un incremento. Esto genera que la ejecución de `POP A` almacene la dirección de retorno de la subrutina en `A` y no el valor respaldado al comienzo del programa.

Se otorgan **0.75 ptos.** por reemplazar correctamente el uso de `RET` con las instrucciones implementadas; y **0.75 ptos.** por señalar correctamente si su implementación cumple la misma función o no. También se aceptan como correctas alternativas donde se hagan modificaciones adicionales aparte del reemplazo de `RET` para almacenar correctamente el valor de `A`. En caso de existir como máximo un error de implementación en la implementación de código, se otorga la mitad del puntaje. Si existe más de un error, no se otorga puntaje.

Pregunta 2: Computador básico (P2-I2-2024-2)

En arquitecturas RISC, las direcciones de retorno de una subrutina se almacenan en un **registro de enlace** (`RA`), a diferencia del computador básico donde se guardan en el *stack*. Asimismo, estas arquitecturas no poseen `PUSH`/`POP`, sino que acceden al *stack* directamente con el *stack pointer* (`SP`). Deberá contestar las preguntas de esta sección a partir de este contexto y el siguiente diagrama:

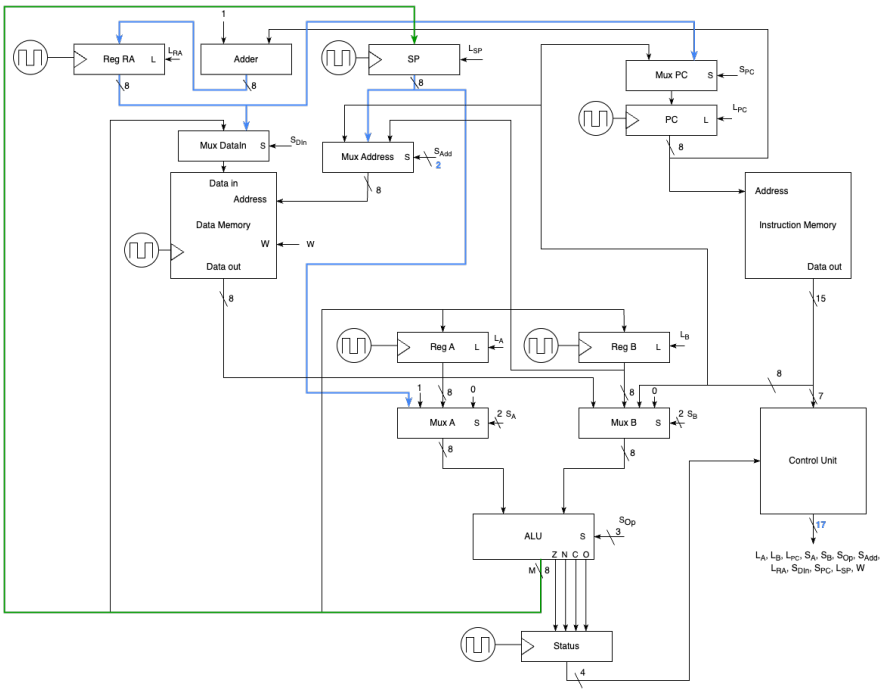


(a) **(5 ptos.)** A partir del diagrama de base adjunto, y asumiendo que se eliminan las instrucciones `CALL`, `RET`, `PUSH` y `POP`, realice las modificaciones de *hardware* necesarias e indique la combinación de señales completa para ejecutar las siguientes instrucciones en un ciclo:

- **(0.5 ptos.)** `MOV A,(SP)`. Guarda en `A` el valor `Mem[SP]`.
- **(0.5 ptos.)** `MOV (SP), A`. Guarda en `Mem[SP]` el valor `A`.
- **(1 pto.)** `ADD SP,Lit`. Guarda en `SP` el valor `SP + Lit`. `Lit` puede ser negativo.
- **(1 pto.)** `MOV (SP), RA`. Guarda en `Mem[SP]` el valor `RA`.
- **(1 pto.)** `JAL RA, label`. Guarda `PC+1` en `RA` y salta a la dirección asociada a `label`.
- **(1 pto.)** `JALR RA`. Salta a la dirección almacenada en `RA`.

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}
`MOV A,(SP)`												
`MOV (SP), A`												
`ADD SP, Lit`												
`MOV (SP), RA`												
`JAL RA, label`												
`JALR RA`												

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.



- La conexión entre `SP` y `Mux Address` permite el uso de `SP` como dirección de memoria con la señal S_{Add}, la que ahora debe ser de 2 bits.
- La conexión entre `SP` y `Mux A` permite su uso como primer operando en la `ALU`.
- La conexión entre la salida de la `ALU` y `SP` permite cargar en este último los resultados de las operaciones con la señal L_{SP}.
- La conexión entre `RA` y `Mux DataIn` permite la escritura de `RA` en memoria.
- La conexión entre `RA` y `PC+1` permite escribir este valor en `RA` con la señal L_{RA}.
- La conexión entre `RA` y `Mux PC` permite realizar saltos hacia direcciones almacenadas en `RA`.

A continuación, la tabla de señales resultante:

Instrucción	L_A	L_B	L_{PC}	L_{SP}	L_{RA}	W	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}
`MOV A, (SP)`	1	0	0	0	0	0	ZERO	DOUT	ADD	SP	-	-
`MOV (SP), A`	0	0	0	0	0	1	A	ZERO	ADD	SP	ALU	-
`ADD SP, Lit`	0	0	0	1	0	0	SP	LIT	ADD	-	-	-
`MOV (SP), RA`	0	0	0	0	0	1	-	-	-	SP	RA	-
`JAL RA, label`	0	0	1	0	1	0	-	-	-	-	-	LIT
`JALR RA`	0	0	1	0	0	0	-	-	-	-	-	RA

Por cada instrucción, se otorga **la mitad** del puntaje por *hardware* y **la otra mitad** por la combinación de señales. En cada criterio, se descuenta la mitad del puntaje en caso de existir, como máximo, un error de implementación o valor en la combinación de señales. En otro caso, no se otorga puntaje.

(b) **(1 pto.)** Modifique el siguiente fragmento de código para que haga uso exclusivo de las instrucciones anteriores. Debe mantener su funcionamiento original y terminar sin errores.

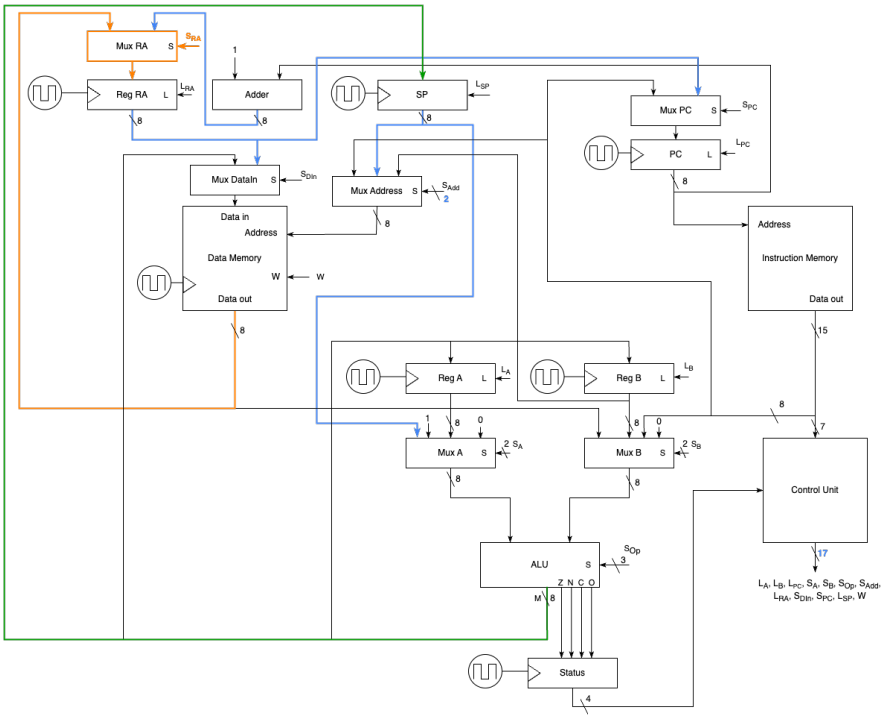
```
MOV A,1
PUSH A
CALL func
POP A
JMP end
func:
  SHL A,A
  CMP A,8
  JEQ end_func
  CALL func
  end_func:
    RET
end:
```

Solución: Es importante señalar que, con las instrucciones entregadas, **no es posible** entregar un fragmento de código que mantenga el funcionamiento original **sin errores**. Esto se debe al hecho de que no existe una instrucción que permita recuperar una dirección de retorno respaldada en memoria (por ejemplo, `MOV RA, (SP)`. Por este motivo, se consideran correctas respuestas que solo caigan en ese caso. A continuación, una solución posible.

```
MOV A,1
MOV (SP),A        ; Reemplazo por PUSH A
ADD SP,-1         ; Reemplazo por PUSH A
JAL RA,func       ; Reemplazo por CALL func
ADD SP,1          ; Reemplazo por POP A
MOV A,(SP)        ; Reemplazo por POP A
JMP end
func:
  SHL A,A
  CMP A,8
  JEQ end_func
  MOV (SP),RA     ; Respaldo de RA para evitar error en llamado recursivo
  ADD SP,-1       ; Respaldo de RA para evitar error en llamado recursivo
  JAL RA,func     ; Reemplazo por CALL func
  ADD SP,1        ; De todas formas, se restituye el valor de SP
  MOV RA,(SP)     ; Si existiera, podríamos recuperar el valor respaldado de RA
  end_func:
    JALR RA       ; Reemplazo por RET
end:
```

Se otorgan **0.25 ptos.** por reemplazar correctamente la instrucción `PUSH A`; **0.25 ptos.** por reemplazar correctamente la instrucción `CALL label`; **0.25 ptos.** por reemplazar correctamente la instrucción `POP A`; y **0.25 ptos.** por reemplazar correctamente la instrucción `RET`. Adicionalmente, si la respuesta alude a la falta de una instrucción para recuperar la dirección de retorno; o bien se usa de forma hipotética una instrucción que permita resolver el problema, **se otorgarán 0.5 ptos. de bonus al total**.

A modo de completitud, se incluye tanto el diagrama como la tabla de señales de la arquitectura que habilita la ejecución de la instrucción `MOV RA,(SP)`.



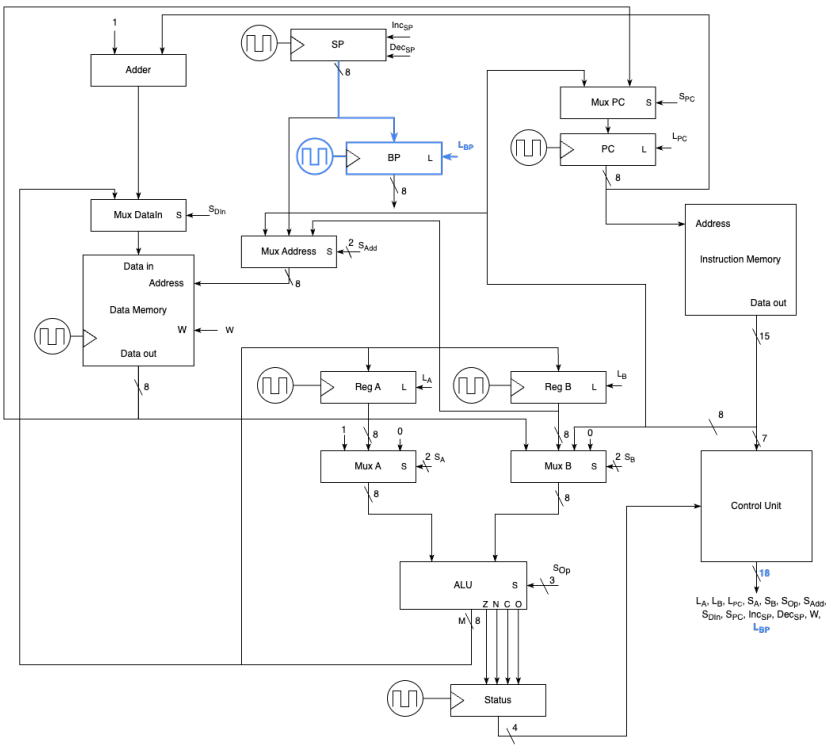
Instrucción	L_A	L_B	L_{PC}	L_{SP}	L_{RA}	W	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	S_{RA}
`MOV A, (SP)`	1	0	0	0	0	0	ZERO	DOUT	ADD	SP	-	-	-
`MOV (SP), A`	0	0	0	0	0	1	A	ZERO	ADD	SP	ALU	-	-
`ADD SP, Lit`	0	0	0	1	0	0	SP	LIT	ADD	-	-	-	-
`MOV (SP), RA`	0	0	0	0	0	1	-	-	-	SP	RA	-	-
`MOV RA, (SP)`	0	0	0	0	1	0	-	-	-	SP	-	-	DOUT
`JAL RA, label`	0	0	1	0	1	0	-	-	-	-	-	LIT	PC+1
`JALR RA`	0	0	1	0	0	0	-	-	-	-	-	RA	-

Pregunta 3: Computador básico (P4-I1-2023-2)

En varias arquitecturas se hace uso del registro *base pointer* (`BP`). Este funciona como punto de referencia para obtener los parámetros y dirección de retorno de cada llamada de una subrutina, lo que facilita el manejo de llamados anidados. Deberá modificar la microarquitectura del computador básico para implementar el registro `BP` de 8 bits y su funcionamiento. Se le listarán las instrucciones que deben ser implementadas. 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). Puede realizar todas las modificaciones en un solo diagrama.

(a) (1 pto.) `MOV BP,SP`. Almacena en el registro `BP` el valor del registro `SP`.

Solución: Para esta modificación, agregamos el registro `BP` a nuestra arquitectura con una nueva señal de carga L_{BP} y con valor de entrada igual a la salida de `SP`. De momento, no conectamos la salida de `BP` con ninguna compuerta (se asume que va a tierra).



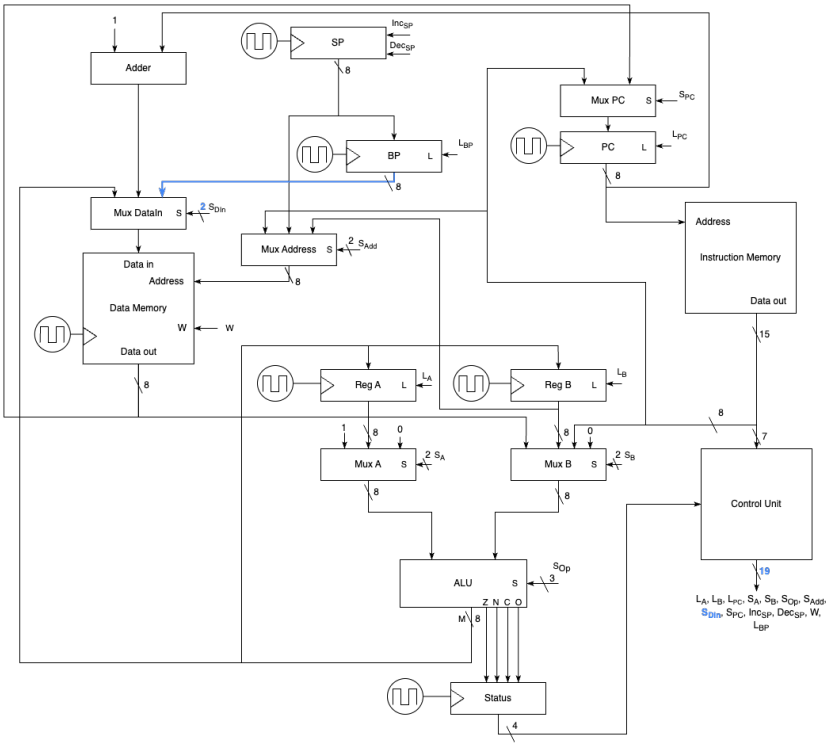
Para ejecutar la instrucción, se utiliza la siguiente combinación de señales:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	L_{BP}
`MOV BP,SP`	0	0	0	0	0	0	-	-	-	-	-	-	1

Se otorgan **0.5 ptos.** por la correctitud de la modificación y **0.5 ptos.** por entregar una combinación de señales correcta para la instrucción.

(b) (1 pto.) `PUSH BP`. Almacena en `Mem[SP]` el valor de `BP` y decrementa `SP` en una unidad.

Solución: Para esta modificación, conectamos la salida del registro `BP` con una de las entradas del componente `Mux DataIn`. Para ello, es necesario incrementar la cantidad de bits de la señal S_{DIn} a dos.



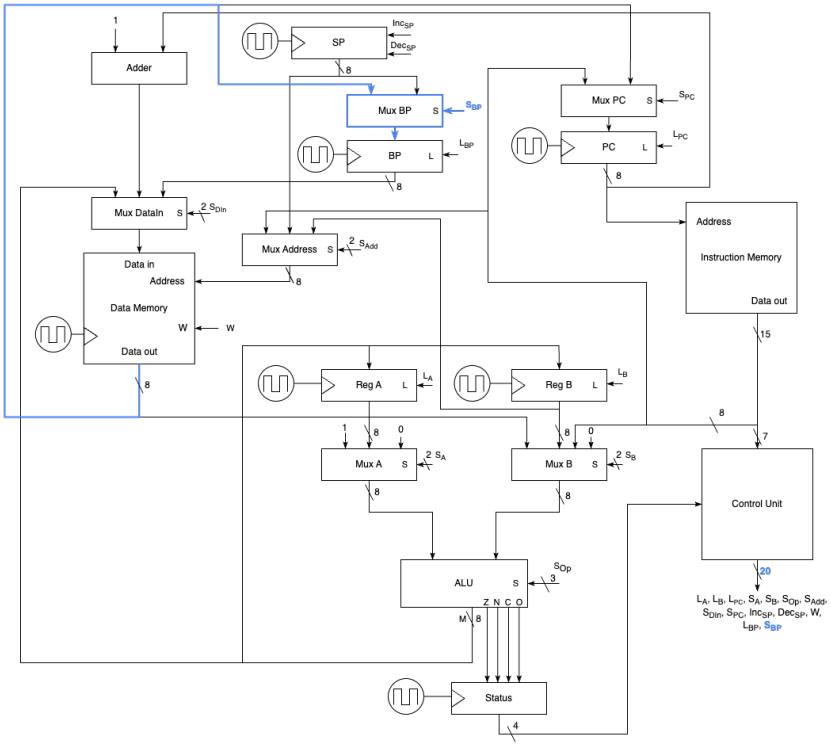
Para ejecutar la instrucción, se utiliza la siguiente combinación de señales:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	L_{BP}
PUSH BP	0	0	0	1	0	1	-	-	-	SP	BP	-	0

Se otorgan **0.5 ptos.** por la correctitud de la modificación y **0.5 ptos.** por entregar una combinación de señales correcta para la instrucción.

(c) (1 pto.) `POP BP`. Incrementa `SP` en una unidad y almacena en `BP` el valor `Mem[SP]`.

Solución: Para esta modificación, conectamos la salida de la memoria de datos con la entrada de `BP`. Dado que ya existe una entrada para `SP`, se agrega un nuevo componente `Mux BP` con señal de selección S_{BP} para escoger el valor de carga entre `SP` y `DOUT`.



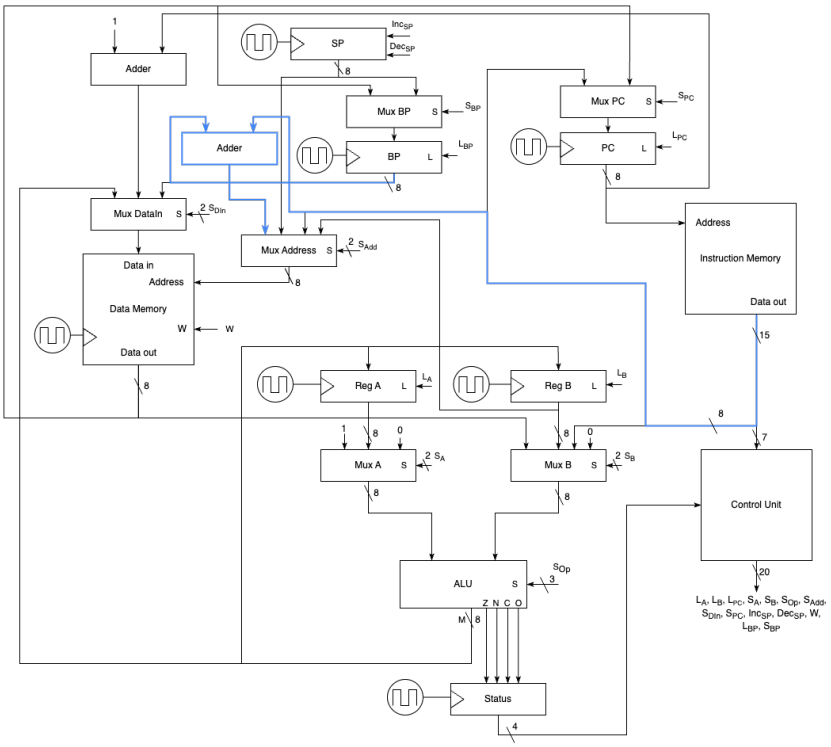
Para ejecutar la instrucción, se utiliza la siguiente combinación de señales:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	L_{BP}	S_{BP}
POP BP	0	0	0	0	1	0	-	-	-	-	-	-	0	-
	0	0	0	0	0	0	-	-	-	SP	-	-	1	DOUT

Se otorgan **0.5 ptos.** por la correctitud de la modificación y **0.5 ptos.** por entregar una combinación de señales correcta para la instrucción.

(d) (2 ptos.) `MOV Reg,(BP + Lit)`. Almacena en `Reg` (`A` o `B`) el valor `Mem[BP + Lit]`.

Solución: Para esta modificación, agregamos un `Adder` que recibirá como entradas el valor del registro `BP` y el literal `LIT` de la memoria de instrucciones, para luego conectar su salida al componente `Mux Address`.

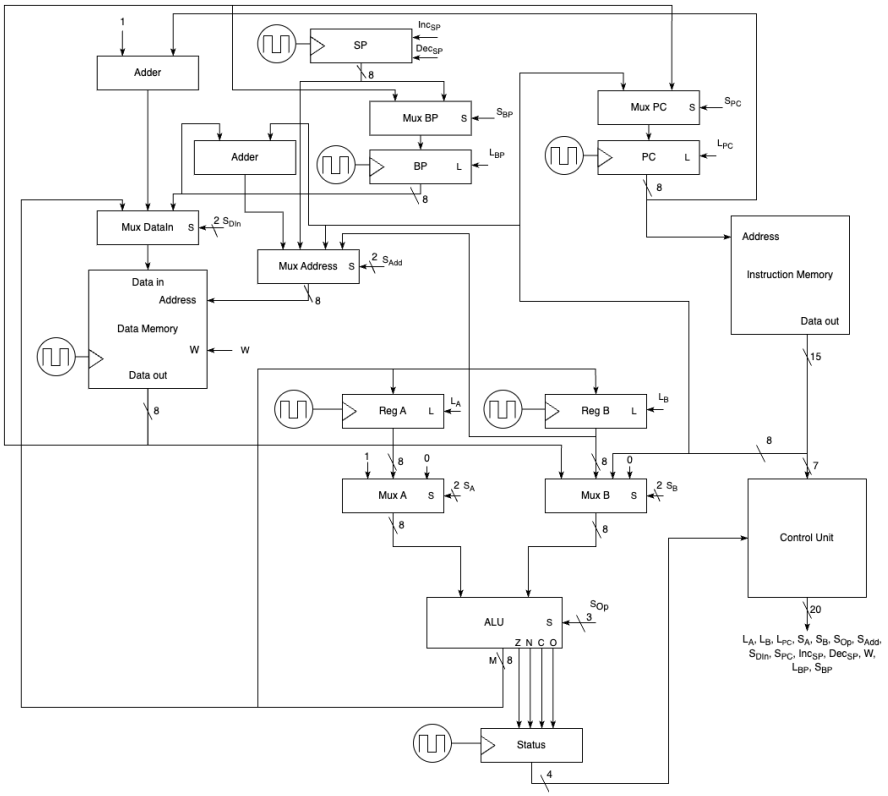


Para ejecutar las instrucciones, se utilizan la siguientes combinaciones de señales:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	L_{BP}	S_{BP}
`MOV A,(BP + Lit)`	1	0	0	0	0	0	0	DOUT	ADD	BP+LIT	-	-	0	-
`MOV B,(BP + Lit)`	0	1	0	0	0	0	0	DOUT	ADD	BP+LIT	-	-	0	-

Se otorgan **1.5 ptos.** por la correctitud de la modificación y **0.5 ptos.** por entregar una combinación de señales correcta para una o ambas instrucciones. Se otorgan **0.75 ptos.** por la modificación si esta presenta como máximo un error de implementación.

En la siguiente página se muestra el diagrama completo de la arquitectura esperada y su tabla de instrucciones, asumiendo que se implementan todas las modificaciones juntas.



Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	L_{BP}	S_{BP}
MOV BP,SP	0	0	0	0	0	0	-	-	-	-	-	-	1	SP
PUSH BP	0	0	0	1	0	1	-	-	-	SP	BP	-	0	-
POP BP	0	0	0	0	1	0	-	-	-	-	-	-	0	-
	0	0	0	0	0	0	-	-	-	SP	-	-	1	DOUT
MOV A,(BP + Lit)	1	0	0	0	0	0	0	DOUT	ADD	BP+LIT	-	-	0	-
MOV B,(BP + Lit)	0	1	0	0	0	0	0	DOUT	ADD	BP+LIT	-	-	0	-

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}	L_{BP}	S_{BP}
MOV BP,SP														
PUSH BP														
POP BP														
MOV A,(BP+Lit)														
MOV B,(BP+Lit)														

(e) (3 ptos.) Asuma que se agregan las instrucciones anteriores a la ISA del computador básico y que se ejecuta el programa adjunto. Explique, a grandes rasgos, lo que realiza la subrutina `func` y señale el valor final de la variable `result`. Puede asumir que `BP` parte en 255.

```
DATA:
  n      3
  result 0
CODE:
  MOV A,(n)          // Dirección Mem. Instr.: 0x00
  PUSH A             // Dirección Mem. Instr.: 0x01
  CALL func          // Dirección Mem. Instr.: 0x02
  POP A              // Dirección Mem. Instr.: 0x03-0x04
  JMP end            // Dirección Mem. Instr.: 0x05
  func:
    PUSH BP          // Dirección Mem. Instr.: 0x06
    MOV BP,SP        // Dirección Mem. Instr.: 0x07
    MOV A,(BP + 3)   // Dirección Mem. Instr.: 0x08
    CMP A,0          // Dirección Mem. Instr.: 0x09
    JEQ base_end_0   // Dirección Mem. Instr.: 0x0A
    CMP A,1          // Dirección Mem. Instr.: 0x0B
    JEQ base_end_1   // Dirección Mem. Instr.: 0x0C
    SUB A,1          // Dirección Mem. Instr.: 0x0D
    PUSH A           // Dirección Mem. Instr.: 0x0E
    CALL func        // Dirección Mem. Instr.: 0x0F
    POP A            // Dirección Mem. Instr.: 0x10-0x11
    MOV A,(BP + 3)   // Dirección Mem. Instr.: 0x12
    SUB A,2          // Dirección Mem. Instr.: 0x13
    PUSH A           // Dirección Mem. Instr.: 0x14
    CALL func        // Dirección Mem. Instr.: 0x15
    POP A            // Dirección Mem. Instr.: 0x16-0x17
    JMP end_func     // Dirección Mem. Instr.: 0x18
    base_end_1:
      INC (result)   // Dirección Mem. Instr.: 0x19
    base_end_0:
      NOP            // Dirección Mem. Instr.: 0x1A
    end_func:
      POP BP         // Dirección Mem. Instr.: 0x1B-0x1C
      RET            // Dirección Mem. Instr.: 0x1D-0x1E
  end:
```

Solución: La subrutina `func` corresponde a la función de recurrencia de Fibonacci, *i.e.*:

$$F_N = \begin{cases} F_{N-1} + F_{N-2} & N > 1 \\ N & N \leq 1 \end{cases}$$

Esto se deduce de la siguiente forma:

- Todo `PUSH A` se realiza antes de un llamado a `func`, siendo este el parámetro `n`.
- Mediante `MOV A,(BP + 3)` se recupera el argumento respaldado. Por cada llamado se guardan en el *stack* los valores `A` (parámetro), `PC+1` (retorno) y `BP`. Como `SP` apunta siempre una dirección sobre el tope del *stack* y `BP` tiene el mismo valor por `MOV BP,SP`, entonces `BP + 3` apunta a la dirección en la que se guardó el parámetro.
- Se realizan dos llamados recursivos al interior de `func`, siendo uno de ellos realizado con el parámetro `n - 1` (proveniente de `SUB A,1`) y el otro con el parámetro `n - 2` (proveniente de `SUB A,2`).

- `base_end_0` y `base_end_1` corresponden a los casos base de la recursión y se llega a ellos si, y solo si `A` es 0 o 1 respectivamente. Interesa sobre todo `base_end_1`, dado que en dicho caso se incrementa el valor de la variable `result`.
- Al terminar un llamado de la subrutina en `end_func`, se reestablece el *stack* al ejecutar `POP BP`, `RET` y `POP A` luego de retornar. Esto asegura que siempre se utilice el valor correcto de `BP` para acceder al parámetro de cada llamado recursivo.

A partir de lo anterior, se deduce entonces que el valor de `result` es igual a:

$$\begin{aligned} F_3 &= F_2 + F_1 \\ &= F_1 + F_0 + F_1 \\ &= 1 + 0 + 1 \\ &= 2 \end{aligned}$$

Se otorgan **2 ptos.** por señalar correctamente el resultado de la ejecución correctamente y **1 pto.** por describir `func` a partir de Fibonacci. Si no se cumple ninguno de los dos criterios anteriores, se otorga **1 pto.** si se muestra un desarrollo parcial pero correcto de la ejecución de `func`.

Pregunta 4: Assembly con subrutinas (P3-I1-2024-1)

(a) Indique los valores de los registros `A` y `B` al finalizar la ejecución del código **(a)**.

(b) Indique los valores de los registros `A` y `B` al finalizar la ejecución del código **(b)**.

(c) El código **(c)** *debería* computar la multiplicación entre las variables `X` e `Y`, pero cuenta con errores que generan un resultado erróneo. Indique el valor de los registros `A`, `B` y la variable `res` al finalizar la ejecución del código. Luego, señale dónde se encuentra el o los errores y cómo se resuelven.

(a)

```
DATA:
    res     0

CODE:
    JMP _start

    func_1:
        SHR A, A
        JCR func_2
        CMP A, 0
        JNE func_1
        RET

    func_2:
        NOT B, A
        MOV (B), 14
        INC (res)
        JMP func_1

    _start:
        MOV A, 6
        CALL func_1
        MOV B, (res)
```

(b)

```
DATA:
    var   10

CODE:
    JMP _start

    func_1:
        MOV B, (var)
        NOT A, A
        ADD A, 1
        ADD B, A
        RET

    _start:
        MOV A, 3
        CALL func_1

    end:
        MOV A, (255)
```

(c)

```
DATA:
    res     0
    X       3
    Y       4
    iter    0

CODE:
    MOV A, (Y)
    PUSH A
    CALL mult
    MOV A, (res)
    JMP end

    mult:
        POP B
    sumar:
        MOV A, (res)
        ADD A, (X)
        MOV (res), A
        INC (iter)
        MOV A, (iter)
        CMP A, B
        JNE sumar
        RET

    end:
```

Consideraciones:

1. La primera instrucción del segmento de `CODE` se guardará en la dirección 0 de la memoria de instrucciones.
2. En las instrucciones de *shifting*, el bit descartado se almacena en el *condition code* de *carry*.

Solución:

(a) Los valores finales de los registros `A` y `B` son 0 y 255 respectivamente. La explicación es la siguiente:

- Al comenzar la ejecución de código, se inicializa el registro `A` con el literal 6 y luego se llama a la subrutina `func_1`.
- Lo que realiza esta subrutina es lo siguiente: (1) Aplica un *shift right* sobre el registro `A`, equivalente a actualizar su valor con el resultado de su división entera por 2; (2) se revisa si se activó la señal de *carry*, lo que ocurre si el bit descartado en el *shift right* fue igual a 1 (*i.e.* se dividió un número impar). Si es el caso, se salta al *label* `func_2`; en otro caso; (3) se revisa si el valor de `A` es igual a 0, en cuyo caso se retorna; si no, se vuelve a ejecutar desde un inicio la subrutina.
- En caso de que se gatille `func_2`, lo que realiza este fragmento es lo siguiente: (1) Almacena en el registro `B` la negación de `A`; (2) utiliza este valor como dirección de la memoria de datos y almacena en dicha ubicación el literal 14; (3) incrementa en una unidad la variable `res`; (4) vuelve a retomar la ejecución de la subrutina `func_1`.
- Para descifrar los valores finales, es necesario ejecutar todos los ciclos de la subrutina:
  1. `A` = `SHL(A)` = 3, $C = 0$. No se realiza el salto con `JCR` y se repite la ejecución de la subrutina.
  2. `A` = `SHL(A)` = 1, $C = 1$. Se se realiza el salto con `JCR` y se tiene que `B` = `NOT(00000001)` = 11111110. Por lo tanto, en dicha dirección se almacena el literal 14: `Mem[11111110]` = `Mem[254]` = 14. La variable `res` incrementa su valor a 1 y se procede a repetir la ejecución de la subrutina.
  3. `A` = `SHL(A)` = 0, $C = 1$. Se se realiza el salto con `JCR` y se tiene que `B` = `NOT(00000000)` = 11111111. Por lo tanto, en dicha dirección se almacena el literal 14: `Mem[11111111]` = `Mem[255]` = 14. La variable `res` incrementa su valor a 2 y se procede a repetir la ejecución de la subrutina.
  4. `A` = `SHL(A)` = 0, $C = 0$. No se realiza el salto con `JCR` y finaliza la ejecución, siendo entonces este el valor final del registro `A`. Al retornar, se carga en el contador `PC` el contenido del tope del `stack`, en este caso, `PC` = `Mem[255]` = 14.
- Considerando que `JMP _start` se almacena en la dirección 0 y que `RET` ocupa dos direcciones en la memoria de instrucciones, se tiene que la dirección de retorno original de la subrutina, correspondiente a la instrucción `MOV B, (res)`, es igual a 13. No obstante lo anterior, este valor se sobreescribe con la subrutina con el literal 14, lo que implica que al retornar el programa **no ejecute** esta instrucción y, por ende, `B` mantenga el valor 255 computado dentro de la subrutina.

Importante: Si bien la instrucción `MOV (B),Lit` no es parte de la ISA del computador básico, la microarquitectura permite su ejecución, por lo que durante la interrogación se informa que se puede asumir que existe y se puede utilizar en este fragmento de código.

Se otorgan **0.75 ptos.** por cada valor final señalado correctamente. En caso de que un valor esté erróneo pero exista desarrollo de la ejecución de código para su obtención, se otorgan **0.375 ptos.** como puntaje parcial.

(b) Los valores finales de los registros `A` y `B` son 9 y 7 respectivamente. La explicación es la siguiente:

- Al comenzar la ejecución del programa, se inicializa el registro `A` con el literal 3 y luego se llama a la subrutina `func_1`.
- Dentro de la subrutina, se almacena el valor de la variable `var` en `B`, que es igual a 10.
- Posteriormente, se actualiza el valor del registro `A` y se reemplaza por $\overline{\texttt{A}} + 1$, lo que es equivalente a computar su inverso aditivo, es decir, -3.
- Se le suma al registro `B` lo computado en `A`, por lo que $\texttt{B} = 10 + -3 = 7$. Este es el valor final del registro.
- Finalmente, al retornar, se almacena en `A` el valor de la dirección de memoria 255, correspondiente al primer valor almacenado en la memoria de *stack*: la dirección de retorno del llamado a `func_1`. Considerando que `JMP _start` se almacena en la dirección 0 y que `RET` ocupa dos direcciones en la memoria de instrucciones, se tiene entonces que la dirección de retorno del llamado, correspondiente a la ubicación de la instrucción `MOV A, (255)`, es igual a 9, siendo este el valor final de `A`. Es importante recordar que el incremento de `SP` en el retorno **no elimina el valor almacenado en el tope del *stack***.

Se otorgan **0.75 ptos.** por cada valor final señalado correctamente. En caso de que un valor esté erróneo pero exista desarrollo de la ejecución de código para su obtención, se otorgan **0.375 ptos.** como puntaje parcial.

(c) Los valores finales de los registros `A` y `B` son 3 y 3 respectivamente, mientras que el valor de la variable `res` es 9. La explicación es la siguiente:

- Al comenzar la ejecución del programa, se almacena en `A` el valor de la variable `Y` (correspondiente a 4) y se respalda en la memoria de *stack* a través de un `PUSH`.
- Se hace el llamado a la subrutina `mult`, almacenando en el tope del *stack* la dirección de retorno que apunta a la instrucción `MOV A,(res)`. No obstante, lo primero que se ejecuta en la subrutina es un `POP B`, lo que hace que se almacene la dirección de retorno en el registro `B` y **cambiando la posición del contador `SP`**, lo que tendrá consecuencias a ser descritas más adelante.
- Se empieza a computar la multiplicación a partir de la suma iterativa de la variable `X` sobre `res`, ejecutándose hasta que el total de iteraciones sea igual al valor de `B`. En este punto, se esperaría que la suma se repita `Y` veces para computar correctamente la multiplicación, pero el registro `B` ahora posee la dirección de memoria de la instrucción `MOV A, (res)`. Considerando que `MOV A, (Y)` se almacena en la dirección 0, se tiene que la dirección de retorno almacenada en `B` es 3. Por lo tanto, lo que se computará será la multiplicación entre `X` igual a 3 y 3, cuyo resultado es igual a 9 y no 12 como se esperaría. Este valor se almacena en la variable `res` y se termina la ejecución de la subrutina.
- Al retornar de la subrutina, se guarda en el contador `PC` el valor del tope del *stack*, correspondiente al valor del registro `A` respaldado en un comienzo. En este caso el

valor fue igual a 4, que corresponde a la dirección de la instrucción `JMP end`, por lo que no se ejecuta `MOV A, (res)` y el programa termina sin actualizar el valor de este registro. Por lo tanto, el valor del registro `A` es igual al asignado con la instrucción `MOV A,(iter)` en la última iteración, que fue igual a 3 (correspondiente a la cantidad de iteraciones ejecutadas). El valor de `B` no se vuelve a actualizar después de la ejecución de la instrucción `POP B`, por lo que su valor es igual a 3.

Para que la multiplicación se compute correctamente, basta con asegurar que: (1) a `B` se le asigne correctamente el valor de la variable `Y`; y (2) que en el tope del *stack* se encuentre almacenada la dirección de retorno de la subrutina. Esto se puede realizar de muchas maneras, pero a continuación se deja un ejemplo:

```
DATA:
    res     0
    X       3
    Y       4
    iter    0

CODE:
    MOV B, (Y)
    CALL mult
    MOV A, (res)
    JMP end

    mult:
        sumar:
        MOV A, (res)
        ADD A, (X)
        MOV (res), A
        INC (iter)
        MOV A, (iter)
        CMP A, B
        JNE sumar
        RET

    end:
```

Se otorgan **0.5 ptos.** por cada valor final señalado correctamente; **0.75 ptos.** por señalar correctamente el error en el programa; y **0.75 ptos.** por indicar una forma válida para su resolución. En caso de que un valor esté erróneo pero exista desarrollo de la ejecución de código para su obtención, se otorgan **0.25 ptos.** como puntaje parcial.

Pregunta 5: Assembly con subrutinas (P1-I1-2023-2)

Indique el valor final de los registros `A` y `B` luego de ejecutar el siguiente fragmento de código escrito en el Assembly del computador básico.

```
CODE:
    MOV A,7
    MOV B,1
    PUSH B
    PUSH A
    RET
    MOV A,0
    MOV B,0
    JMP end
    end:
```

Solución: Cuando se ejecuta el fragmento de código anterior, `RET` gatilla la carga del valor numérico del tope del *stack* en el registro `PC`. En este caso, el tope del `stack` posee el valor 7 y por ende el programa saltará a dicha dirección en la memoria de instrucciones. A continuación, se muestra el mismo fragmento mostrando las direcciones en las que se almacena cada instrucción.

```
CODE:
    MOV A,7     ;Dirección 0x00
    MOV B,1     ;Dirección 0x01
    PUSH B      ;Dirección 0x02
    PUSH A      ;Dirección 0x03
    RET         ;Dirección 0x04-0x05
    MOV A,0     ;Dirección 0x06
    MOV B,0     ;Dirección 0x07       <- RET gatilla un salto aquí.
    JMP end     ;Dirección 0x08
    end:
```

Por lo tanto, el programa terminará ejecutando las instrucciones `MOV B,0` y `JMP end`, saltándose `MOV A,0`. De esta forma, el valor final del registro `A` es igual a 7 y el del registro `B` es igual a 0. Se otorga **1 pto.** por cada registro donde se indique su valor final correcto