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

IIC2343 - Arquitectura de Computadores (I/2024)

Interrogación 1

Pauta de evaluación

Pregunta 1: Representación de Números (6 ptos.)

(a) (2 ptos.) De acuerdo con los autores Patterson y Hennessy, la representación de números enteros negativos usando **signo y magnitud** tiene tres problemas:

1. El bit de signo puede ser el de más a la izquierda o el de más a la derecha.
2. Al sumar dos números, se necesita un paso adicional para conocer sus bits de signo.
3. Existen dos representaciones del número cero: uno con bit de signo positivo y otro con bit de signo negativo.

¿Cuál o cuáles de estos problemas son resueltos por la representación **complemento de dos**, y cómo?

Solución: La representación complemento de dos resuelve los tres problemas:

1. No hay un bit de signo propiamente, pero el signo del número se conoce mirando solo el bit más significativo: $0 \rightarrow$ positivo, $1 \rightarrow$ negativo.
2. El número resultante de una suma siempre tiene el (bit de) signo correcto (si no hay *overflow*).
3. Solo hay un patrón de bits que representa el valor 0: 000...0.

Se otorgan **0.5 ptos.** por la justificación correcta de cada uno de los problemas resueltos y **0.5 ptos.** por señalar correctamente que la representación complemento de dos los resuelve todos.

(b) (2 ptos.) Suponga que en un videojuego posee un contador de vidas que almacena números **sin signo**. Durante el juego, va acumulando vidas constantemente hasta llegar a un total de 127. Posterior a ello, obtiene 3 vidas consecutivas y se da cuenta que su contador posee un valor igual a 2. Explique, a partir de la composición en bits del contador, por qué ocurre esto.

Solución: El contexto anterior se explica si el contador de vidas se compone de 7 bits. Dado que almacena números sin signo, el máximo valor que puede representar es $1111111b = 127$. Luego, si este valor se incrementa en 3 unidades, se observa el siguiente resultado:

$$\begin{array}{r} \text{bit de } \textit{carry} \rightarrow 11111110 \\ 1111111 \\ +\ 0000011 \\ \hline 1\ 0000010 \end{array}$$

Dado que se cuenta con 7 bits para contar las vidas, se pierde el bit más significativo de *carry*, generando *overflow* y llegando a un valor igual a $0000010b = 2$. Se otorga **1 pto.** por indicar correctamente que el resultado se da por *overflow* y **1 pto.** por señalar que el problema ocurre al tener un contador de 7 bits.

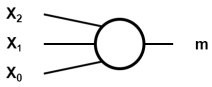
(c) (2 ptos.) Suponga que posee dos números positivos a, b tales que $a \geq b$. Si se realiza la operación $a - b$, entonces se tendrá un *carry* de salida igual a 1. Explique por qué se da este resultado tomando en consideración el bit más significativo de a, b, $a - b$ y la representación con complemento de 2.

Solución: Asumamos que a, b y $a - b$ son de N bits y, de esta forma, a_{N-1}, b_{N-1} y $(a-b)_{N-1}$ sus bits más significativos. Llamaremos al *carry* de salida C_{out}; al *carry* de entrada de la suma de los bits más significativos C_{in}^{N-1}; y $C_2(b)$ al cómputo de $-b$. Dado que $a > 0$, $b > 0$ y $a \geq b$, entonces, $a_{N-1} = 0$, $b_{N-1} = 0$ y $(a-b)_{N-1} = 0$. Ahora, sabemos que $(a-b)_{N-1} = (a_{N-1}$ `XOR` $C_2(b)_{N-1})$ `XOR` C_{in}^{N-1} y $C_{\text{out}} = (a_{N-1}$ `AND` $C_2(b)_{N-1})$ `OR` $((a_{N-1}$ `XOR` $C_2(b)_{N-1})$ `AND` $C_{\text{in}}^{N-1})$. Como $b > 0$, entonces $C_2(b) < 0$ y, así, $C_2(b)_{N-1} = 1$. Si reemplazamos esto en el cómputo de $(a-b)_{N-1}$, cuyo valor conocemos, entonces: $0 = (0$ `XOR` $1)$ `XOR` $C_{\text{in}}^{N-1} = 1$ `XOR` C_{in}^{N-1}. La única forma de cumplir esto es que $C_{\text{in}}^{N-1} = 1$. De esta forma se deduce que **el *carry* de entrada de la suma entre los bits más significativos de a y $-b$ debe ser igual a 1**. Con este resultado, se tiene $C_{\text{out}} = (0$ `AND` 1 `OR` $((0$ `XOR` 1 `AND` $1) = 0$ `OR` $(1$ `AND` $1) = 0$ `OR` $1 = 1$. En resumen: como el bit más significativo de $a - b$ es 0, se debe tener un *carry* de entrada igual a 1 al sumar los bits más significativos de a y $-b$, lo que a raíz de sus valores genera un *carry* de salida igual a 1. Se otorga **1 pto.** por señalar correctamente los bits más significativos de los operandos y el resultado (ya sea explicado o con ejemplos); y **1 pto.** por indicar que la suma de los bits más significativos **debe** recibir un acarreo igual a 1, lo que genera el *carry* de salida igual a 1.

Pregunta 2: Circuitos digitales y Almacenamiento de datos (6 ptos.)

En esta pregunta, **debe** contestar el inciso (a) y escoger un inciso a responder: (b) o (c).

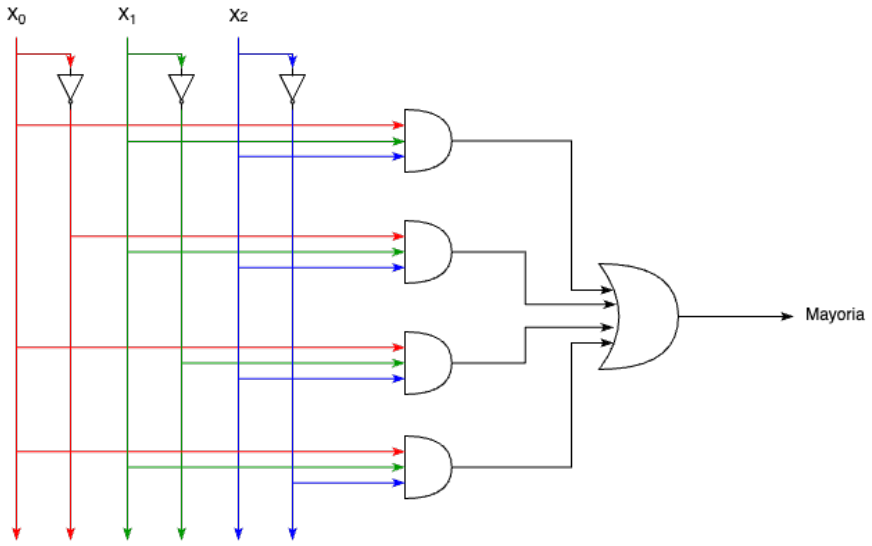
Definiremos una nueva compuerta lógica llamada `Mayoria`. Esta recibirá tres señales de entrada de 1 bit y su salida, de 1 bit, corresponderá al valor que más se repite. Por ejemplo, si tenemos $X_0 = 0$, $X_1 = 0$, $X_2 = 1$, la salida de la compuerta `Mayoria` es 0.



A partir de esta compuerta, responda los incisos (a) y (b). Si responde (b), ignore el inciso (c).

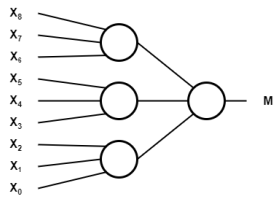
(a) (3 ptos.) Diseñe la compuerta lógica `Mayoria`, de 3 entradas, a partir de las siguientes compuertas: `AND`, `OR`, `NOT`, `XOR`, `NAND` o `NOR`.

Solución: Tres valores solo pueden darse como 3 a 0 o 2 a 1. Si los valores corresponden a los *inputs* X_0, X_1 y X_2, la fórmula lógica que calcula el valor mayoritario, computada mediante *minterms*, es $X_0X_1X_2 + \overline{X_0}X_1X_2 + X_0\overline{X_1}X_2 + X_0X_1\overline{X_2}$. El circuito equivalente es el siguiente:



Cualquier circuito que cumpla con la salida esperada para cada combinación de *inputs* será considerado correcto. Se otorgan **0.375 ptos.** por cada salida correctamente computada según cada combinación de valores de X_0, X_1 y X_2.

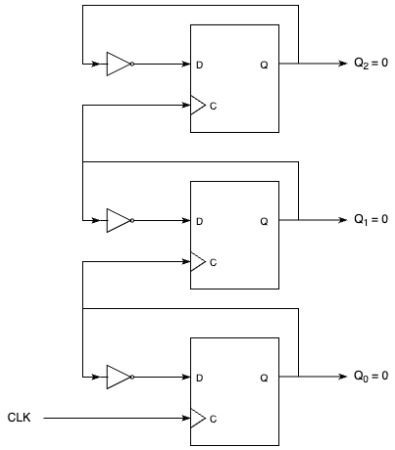
(b) (3 ptos.) Haciendo uso de la compuerta `Mayoria`, se propone el siguiente circuito para determinar el valor que más se repite entre 9 *inputs*:



Indique si el circuito propuesto resuelve correctamente el problema planteado. Si lo resuelve, justifique por qué; si no lo hace, indique en qué casos no funciona.

Solución: Hay solo dos casos que analizar: si la mayoría fue 6 a 3 o si fue 5 a 4. Si fue 6 a 3, el circuito propuesto determina correctamente el valor que más se repite. Para comprobarlo, hay que ver cómo pueden dividirse los 6 valores de mayoría entre las tres compuertas de la primera "fase" del circuito; solo hay tres posibilidades: 3, 3, 0; o 2, 2, 2; o 3, 2, 1. En los tres casos, la cuarta compuerta (la segunda "fase") siempre produce el resultado correcto. Si fue 5 a 4, el circuito propuesto puede equivocarse. Aquí basta un contraejemplo: si los *inputs* se dividen 3-0, 1-2 y 1-2, el circuito dice que la opción minoritaria, de 4 valores, es la mayoritaria. Se otorgan **1.5 ptos.** por señalar correctamente que el circuito no resuelve el problema planteado; y **1.5 ptos.** por justificar con un contraejemplo.

(c) (3 ptos.) Suponga que posee un componente con una salida de 3 bits igual a $Q_2Q_1Q_0$ conectado a una señal de control `CLK`. Su circuito interno se compone de flip-flops de tipo D conectados de la siguiente forma:



Asumiendo que se tiene un valor inicial $Q_2Q_1Q_0 = 000$, indique, justificadamente, cómo se va modificando el estado del componente con cada flanco de subida de la señal `CLK`.

Solución: En este componente, es importante notar que la entrada de la señal C del flip-flop de estado Q_1 recibe el valor del estado Q_0; mientras que el flip-flop de estado Q_2 recibe como señal de control el valor del estado Q_1. Entonces, para poder evaluar bien cómo se alternan los valores de estos, es importante determinar la frecuencia de estas señales. Si tomamos como F la frecuencia de la señal `CLK` y $\frac{1}{F}$ como su periodo, entonces la señal Q_0 alterna su valor de 0 a 1 en $\frac{1}{F}$ unidades y de 1 a 0 en $\frac{1}{F}$ unidades; es decir, tiene un periodo igual a $\frac{2}{F}$ y, por ende, una frecuencia igual a $\frac{F}{2}$. Si extendemos este análisis para Q_1, determinaremos que su frecuencia es igual a $\frac{F}{4}$. Con este resultado en consideración, Q_0 alterna su valor por cada flanco de subida; Q_1 por cada dos flancos de subida; y Q_2 por cada cuatro flancos de subida. De esta forma, el componente actualiza sus salidas $Q_2Q_1Q_0$ de la siguiente manera: $000 \rightarrow^{\texttt{CLK}^{\uparrow}} 111 \rightarrow^{\texttt{CLK}\ \uparrow} 110 \rightarrow^{\texttt{CLK}^{\uparrow}} 101 \rightarrow^{\texttt{CLK}^{\uparrow}} 100 \rightarrow^{\texttt{CLK}^{\uparrow}} 011 \rightarrow^{\texttt{CLK}^{\uparrow}} 010 \rightarrow^{\texttt{CLK}^{\uparrow}} 001 \rightarrow^{\texttt{CLK}^{\uparrow}} 000$. En resumen: es un contador de 3 bits que decrementa su valor por flanco de subida. Es importante notar que este comportamiento se explica por la activación simultánea de los tres flip-flops con el primer flanco de subida, para luego cambiar sus valores según las frecuencias antes descritas. Se otorgan **1.5 ptos.** por indicar correctamente que la frecuencia de cada flip-flop se reduce a la mitad; y **1.5 ptos.** por señalar correctamente cómo se van modificando los valores de salida del componente por flanco de subida.

Pregunta 3: Saltos y Subrutinas (6 ptos.)

(a) (1.5 ptos.) Indique los valores de los registros `A` y `B` al finalizar la ejecución del código **(a)**.

(b) (1.5 ptos.) Indique los valores de los registros `A` y `B` al finalizar la ejecución del código **(b)**.

(c) (3 ptos.) 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 4: Computador básico (6 ptos.)

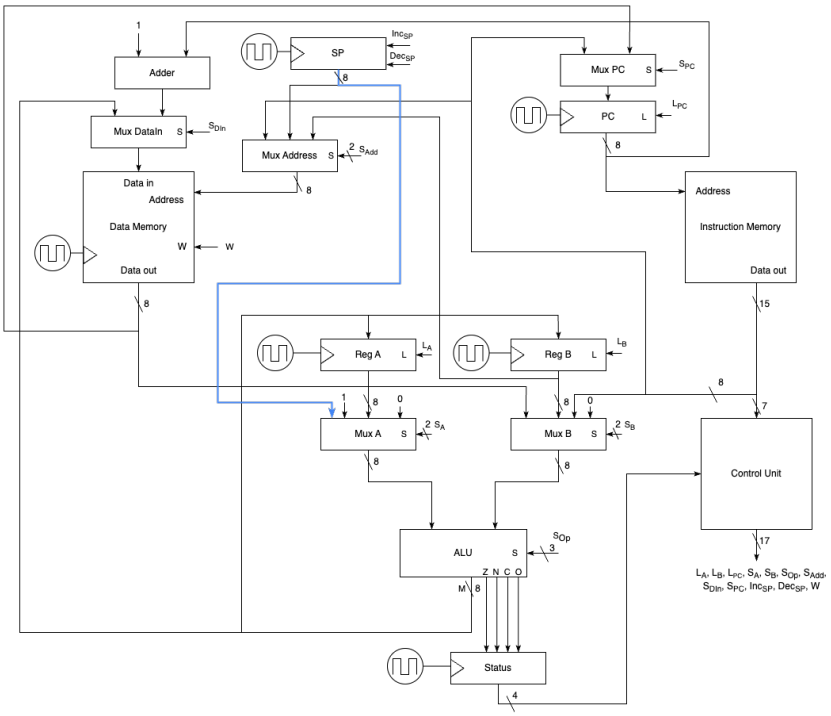
Asuma que se encuentra trabajando como programador(a) en la oficina del famoso juego desarrollado completamente en Assembly: *Rollercoaster Tycoon*. Luego de unas horas, se da cuenta que la letra `T` del teclado de su computador ha dejado de funcionar, la que es de suma importancia al momento de escribir 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.

(a) (1.5 ptos.) `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:



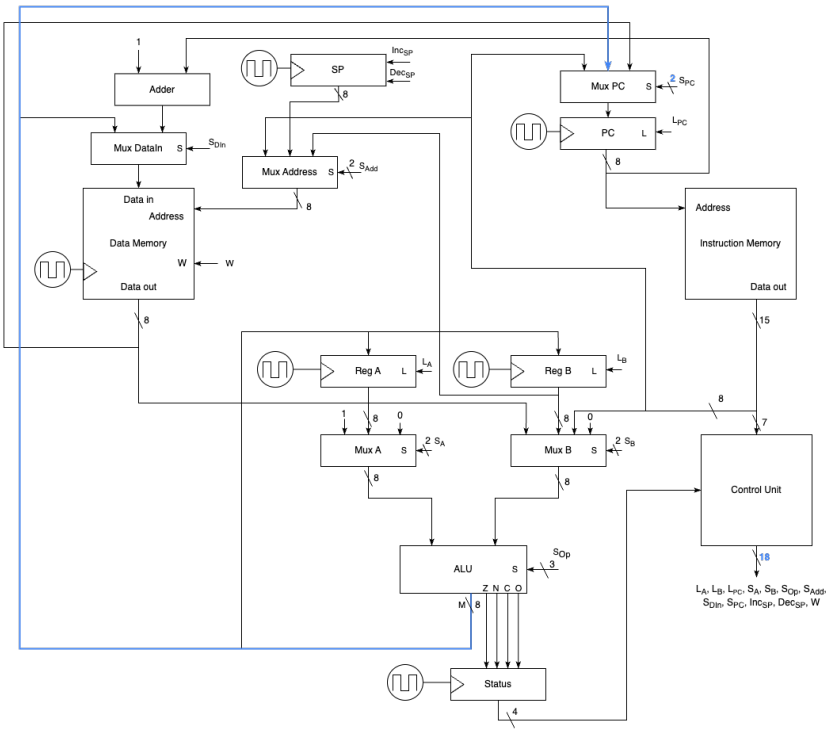
Para ejecutar la instrucción, se utiliza la siguiente combinación de señales:

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.

(b) (1.5 ptos.) `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.

(c) (1.5 ptos.) `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

- 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.

(d) (1.5 ptos.) 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:

1. Reemplazar `RET` por:
   a) `POP B`
   b) `JMP B`

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

2. Reemplazar `RET` por:

   a) `MOV B, SP`

   b) `INC B`

   c) `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.