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

IIC2343 - Arquitectura de Computadores (I/2023)

Tarea 2

Pauta de evaluación

Consideraciones

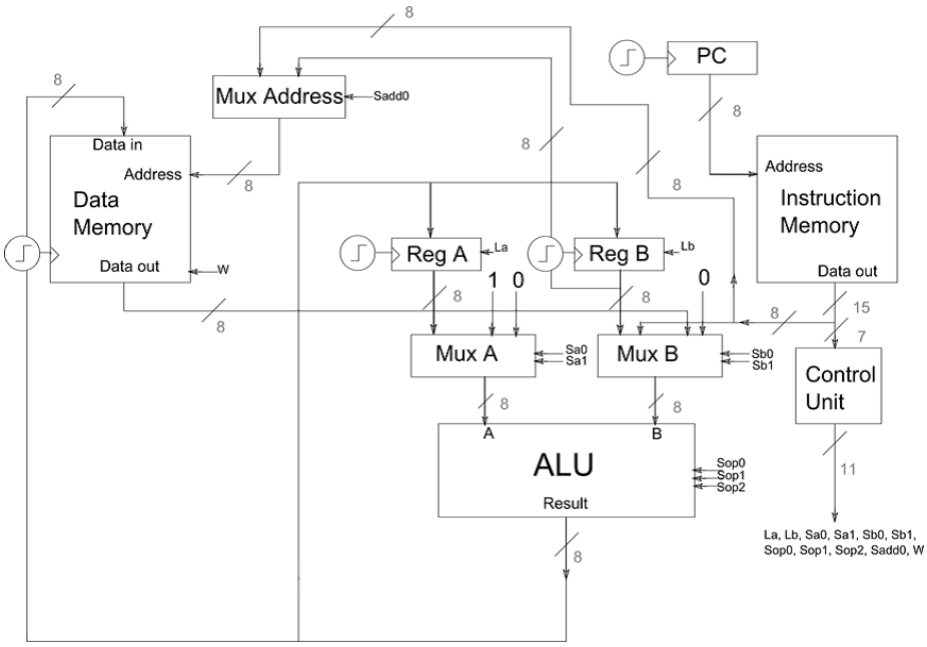
- Respuestas sin desarrollo o justificación no tendrán puntaje.
- Cada pregunta podría tener más de un desarrollo válido. La pauta evalúa eso y este documento solo muestra una alternativa de solución.
- Cualquier detección de infracción al código de honor será sancionada.

Código de Honor de la UC

"Como miembro de la comunidad de la Pontificia Universidad Católica de Chile me comprometo a respetar los principios y normativas que la rigen. Asimismo, prometo actuar con rectitud y honestidad en las relaciones con los demás integrantes de la comunidad y en la realización de todo trabajo, particularmente en aquellas actividades vinculadas a la docencia, el aprendizaje y la creación, difusión y transferencia del conocimiento. Además, velaré por la integridad de las personas y cuidaré los bienes de la Universidad."

Pregunta 1: Programabilidad (16 ptos.)

Desarrolle las siguientes preguntas a partir del diagrama del computador básico **sin soporte de saltos** incluido a continuación.



(a) (8 ptos.) Modifique el diagrama del computador básico sin soporte de saltos para habilitar las siguientes instrucciones **en una sola iteración**:

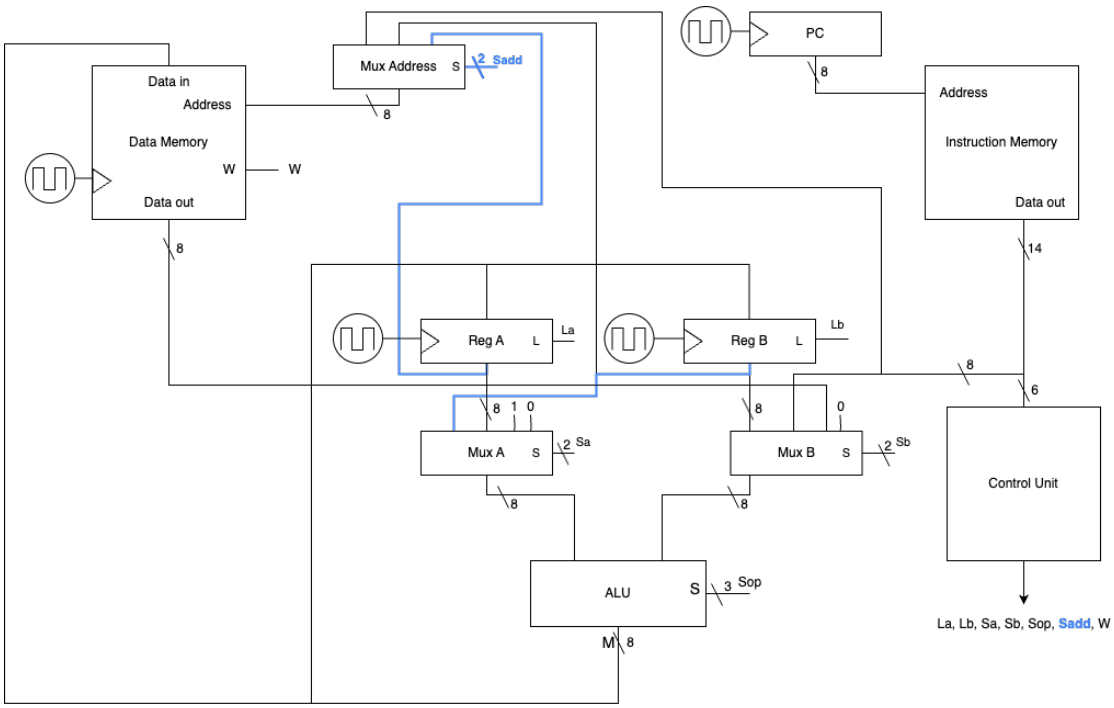
- `Op1 B,(A)`; `Op1` $\in$ {`MOV, ADD, SUB, AND, OR, XOR`}
- `Op1 A,(A)`; `Op1` $\in$ {`MOV, ADD, SUB, AND, OR, XOR`}
- `INC A`
- `INC (A)`

En resumen, se le pide dar soporte para **direccionamiento indirecto a través del registro *A***. No es necesario asociar un *opcode* a las instrucciones anteriores, pero sí debe incluir la combinación de señales que ejecuta cada una de ellas. Además, debe indicar si alguna de las combinaciones de señales de instrucciones existentes se ve modificada por su diseño.

Solución: Las modificaciones mínimas que requiere el diagrama son las siguientes:

- Agregar una conexión entre el registro A y el *input* de direccionamiento de la memoria de datos. La forma más directa es conectando dicho registro con el componente `Mux Address`, lo que a su vez implica aumentar a 2 bits el bus de selección de este.
- Agregar una conexión entre el registro B y el componente `Mux A`. Esto tiene como fin poder hacer instrucciones del tipo `Op1 B,(A)` en el orden esperado (donde B es el primer operando).

La descripción anterior se traduce en el siguiente diagrama y tabla de instrucciones:



Instrucción	Operandos	Literal	La	Lb	Sa	Sb	Sop	Sadd	W
ADD	B,(A)	-	0	1	B	DOUT	ADD	A	0
	A,(A)	-	1	0	A	DOUT	ADD	A	0
SUB	B,(A)	-	0	1	B	DOUT	SUB	A	0
	A,(A)	-	1	0	A	DOUT	SUB	A	0
AND	B,(A)	-	0	1	B	DOUT	AND	A	0
	A,(A)	-	1	0	A	DOUT	AND	A	0
OR	B,(A)	-	0	1	B	DOUT	OR	A	0
	A,(A)	-	1	0	A	DOUT	OR	A	0
XOR	B,(A)	-	0	1	B	DOUT	XOR	A	0
	A,(A)	-	1	0	A	DOUT	XOR	A	0
INC	A	1	1	0	A	LIT	ADD	-	0
	(A)	-	0	0	ONE	DOUT	ADD	A	1

`DOUT` representa la salida de la memoria de datos; `LIT` representa el literal obtenido de la memoria de instrucciones; y `ONE` representa la selección de la señal constante 1 del componente `Mux A`.

En este caso, `INC A` se implementa con la operación `ADD A,Lit`, con `Lit = 1`. Esta transformación queda a cargo del *Assembler*. Se podría implementar agregando una señal constante 1 en el componente `Mux B`, pero se opta por implementar por lo ya existente.

Estos cambios podrían habilitar más instrucciones, pero solo se listan las solicitadas en el enunciado.

Distribución de puntaje

- **2 puntos** por realizar una conexión correcta entre la memoria de datos y el registro A.
- **2 puntos** por realizar una conexión correcta entre el registro B y el componente `Mux A` para una correcta ejecución de las instrucciones `Op1 B,(A)`. Se otorgará la mitad del puntaje si se hace la implementación con el orden **invertido**.
- **1 punto** por implementar la instrucción `INC A` (con o sin modificación de *hardware*).
- **1 punto** por implementar la instrucción `INC (A)` (con o sin modificación de *hardware*).
- **2 puntos** por la inclusión de una tabla de instrucciones correcta con operandos y señales de control correspondientes.

(b) (8 ptos.) Realice una modificación adicional sobre la arquitectura del computador básico para habilitar, al menos, **una nueva instrucción**. **No es válido**:

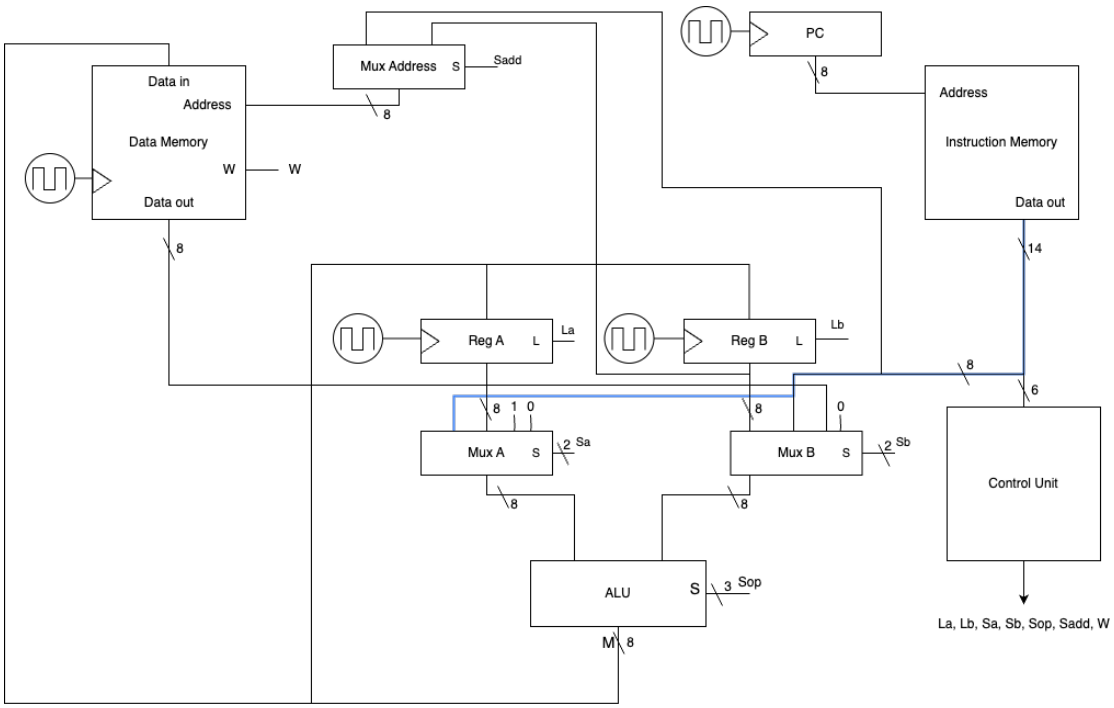
- Realizar cambios ya conocidos, tales como el soporte de saltos y subrutinas o diagramas explicitados en las diapositivas de clases.
- Hacer uso de la modificación de la pregunta anterior.
- Usar combinaciones de señales no utilizadas. Por ejemplo, cargar simultáneamente una operación en los registros A y B.

Debe agregar al menos un nuevo componente o conexión al diagrama y realizar las modificaciones pertinentes para que la instrucción nueva se pueda ejecutar.

Solución: En este caso, se puede optar por realizar la conexión faltante más simple y directa de todas: transmitir el literal al componente `Mux A`. Esto habilita operaciones del siguiente tipo:

- `Op Lit,B`; `Op` $\in$ {`MOV, ADD, SUB, AND, OR, XOR`}
- `Op Lit,(B)`; `Op` $\in$ {`MOV, ADD, SUB, AND, OR, XOR`}

No solo agrega la capacidad de poder realizar más operaciones con el literal en un ciclo, sino que además no requiere de modificaciones en la señal de control S_a. A continuación, se muestra el diagrama resultante y su tabla de instrucciones.



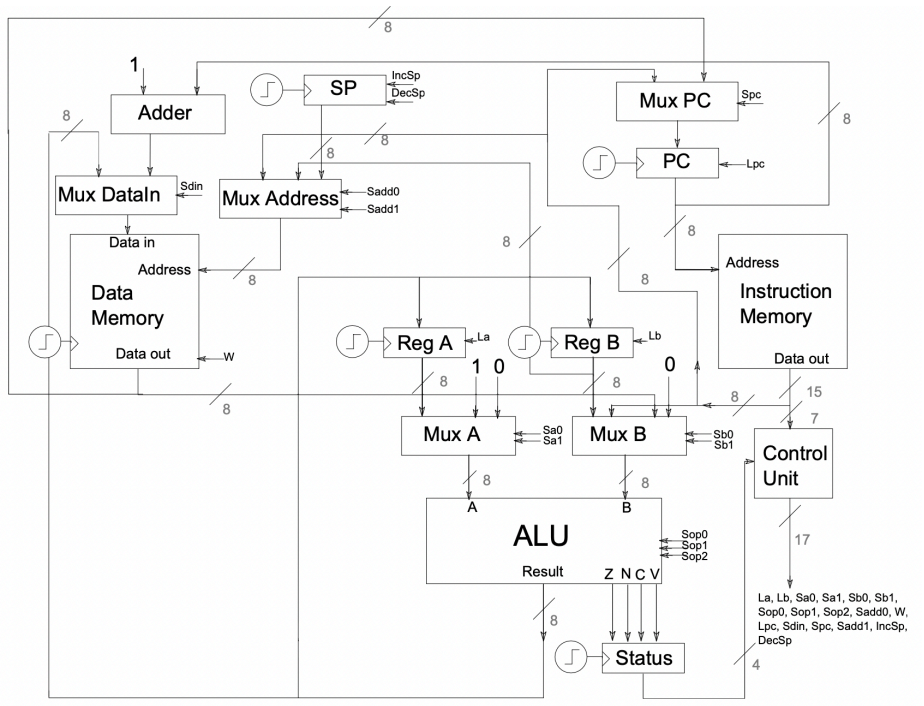
Instrucción	Operandos	Literal	La	Lb	Sa	Sb	Sop	Sadd	W
ADD	Lit,B	LIT	0	1	LIT	B	ADD	-	0
	Lit,(B)	LIT	0	0	LIT	DOUT	ADD	B	1
SUB	Lit,B	LIT	0	1	LIT	B	SUB	-	0
	Lit,(B)	LIT	1	0	LIT	DOUT	SUB	B	1
AND	Lit,B	LIT	0	1	LIT	B	AND	-	0
	Lit,(B)	LIT	1	0	LIT	DOUT	AND	B	1
OR	Lit,B	LIT	0	1	LIT	B	OR	-	0
	Lit,(B)	LIT	1	0	LIT	DOUT	OR	B	1
XOR	Lit,B	LIT	0	1	LIT	B	XOR	-	0
	Lit,(B)	LIT	1	0	LIT	DOUT	XOR	B	1

Distribución de puntaje

- **6 puntos** por realizar una modificación que cumpla con las reglas establecidas. Se otorgará la mitad del puntaje si dicha implementación presenta al menos un error en la ejecución de las instrucciones añadidas. No se otorgará puntaje si rompe las reglas del enunciado.
- **2 puntos** por la inclusión de una tabla de instrucciones correcta con operandos y señales de control correspondientes.

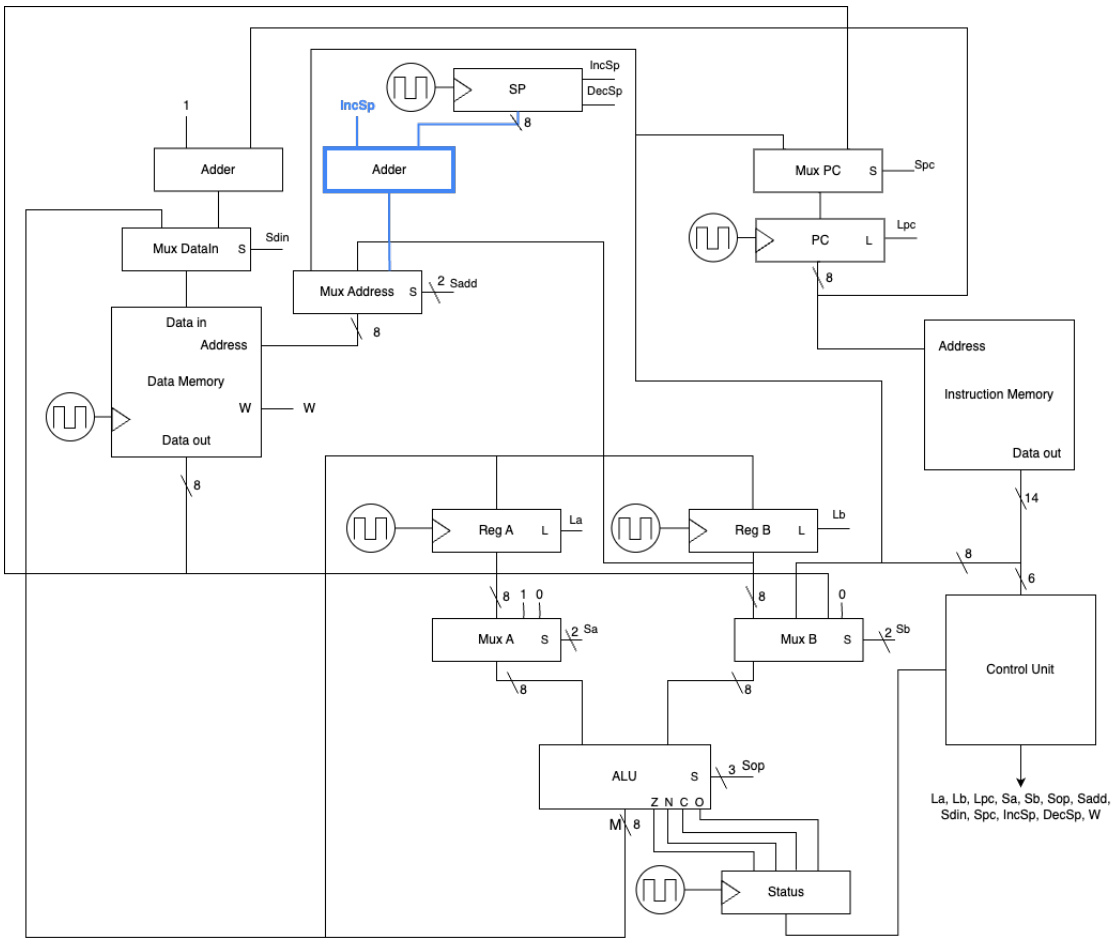
Pregunta 2: Saltos y subrutinas (30 ptos.)

Desarrolle las siguientes preguntas a partir del diagrama del computador básico incluido a continuación.



(a) (6 ptos.) Modifique el diagrama del computador básico para que las instrucciones `RET` y `POP` tomen un solo ciclo. Debe cuidar que sus modificaciones **no alteren el funcionamiento del resto de las instrucciones**.

Solución: El problema de las instrucciones `RET` y `POP` es que requieren invertir un ciclo de *clock* para incrementar en una unidad el contador `SP` y, de esta forma acceder al último elemento del *stack*. Entonces, la solución concreta es transmitir la dirección `SP + 1` a la memoria de datos para poder acceder de inmediato al dato deseado. Esto se puede hacer de varias maneras, pero a continuación se muestra la forma que requiere menos cambios de *hardware* y que no implica la modificación de otras instrucciones para su funcionamiento:



En este caso, se utilizará la dirección `SP + 1` si, y solo si la señal Inc_{SP} está activa (que es lo deseado para las instrucciones `RET` y `POP`). Entonces, la tabla de estas instrucciones quedan de la siguiente manera:

Instrucción	Operandos	Literal	Lpc	La	Lb	Sa	Sb	Sop	Sadd	Sdin	Spc	W	IncSp	DecSp
RET	-	-	1	0	0	-	-	-	SP	-	DOUT	0	1	0
POP	A	-	0	1	0	ZERO	DOUT	ADD	SP	-	-	0	1	0
	B	-	0	0	1	ZERO	DOUT	ADD	SP	-	-	0	1	0

Distribución de puntaje

- **4 puntos** por realizar una modificación que permita que las instrucciones `RET` y `POP` puedan realizarse en un solo ciclo. Se otorgará la mitad del puntaje si existe un caso que no funcione o si se ven afectadas otras instrucciones por esta modificación.
- **2 puntos** por la inclusión de una tabla de instrucciones correcta con operandos y señales de control correspondientes; o bien por la explicación de cómo se ejecutan las instrucciones con la modificación realizada.

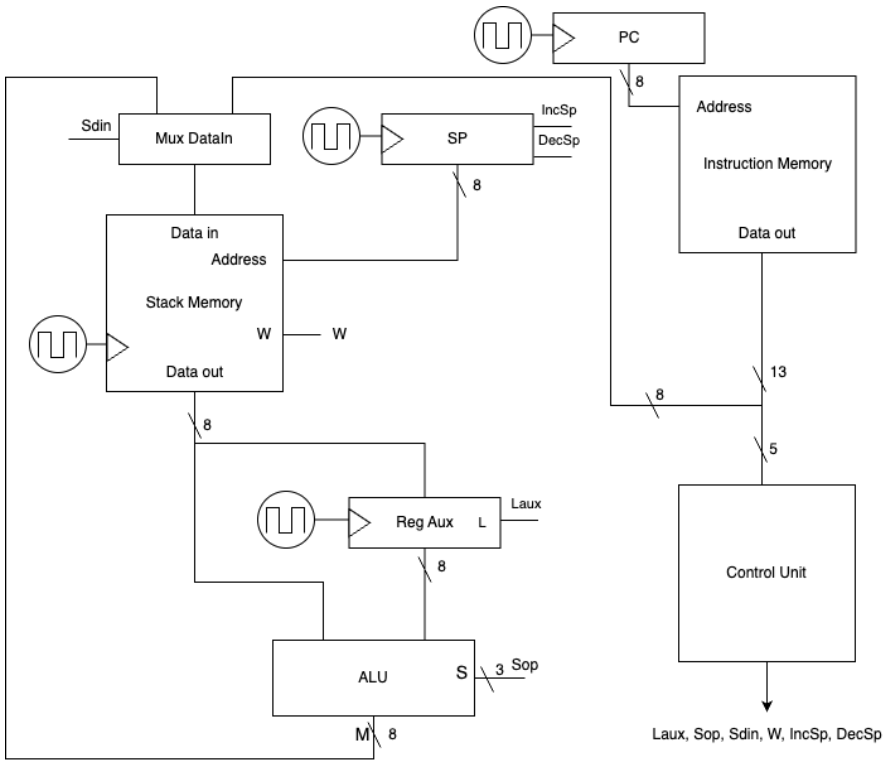
(b) (12 ptos.) Una máquina de *stack* es un computador que utiliza una memoria de *stack* en vez de registros para almacenar los resultados de las operaciones. Esto significa que cada instrucción aritmética o lógica de dos parámetros, toma los dos valores en el tope del *stack* y luego los elimina, sustituyéndolos por el valor de la operación recién realizada. Para el caso de las operaciones de un parámetro, por ejemplo `NOT`, el computador solo sustituye un valor en el tope del *stack* por el nuevo valor. Además, una máquina de *stack* es capaz de cargar valores literales en el tope del *stack* y también descartarlos. Al igual que en el computador básico, la memoria de *stack* se accede a través del contador `SP`.

A partir de la descripción anterior, elabore el diagrama de la microarquitectura de una máquina de *stack* de 8 bits. Esta máquina debe ser capaz de realizar las mismas operaciones aritméticas y lógicas que el computador básico. Puede incluir en su diagrama un **registro de memoria auxiliar** donde puede almacenar datos temporalmente para ejecutar operaciones de más de un parámetro, así como también el contador `PC` para la lectura de la memoria de instrucciones. Junto a su diagrama, debe incluir el *set* de instrucciones *Assembly* para su máquina, indicando *opcodes*, señales de control y cantidad de ciclos de ejecución. Recuerde que un *opcode* se asocia a un ciclo de ejecución, por lo que puede tener instrucciones asociadas a más de uno.

Solución: Para elaborar el diagrama de la máquina de *stack*, se toman las siguientes consideraciones:

- Al ejecutar operaciones de dos operandos (*i.e.* `ADD`, `SUB`, `AND`, `OR`, `XOR`), se ocupa el tope del stack como el *input B* de la ALU y el valor debajo de este como el *input A*. En el caso de instrucciones de un operando (*i.e.* `NOT`, `SHL`, `SHR`), simplemente se reemplaza el tope del *stack* con el valor operado en la ALU.
- El utilizará el literal directamente solo para ser incorporado en el tope del *stack* con una instrucción `PUSH`.
- El funcionamiento del *stack pointer* (contador `SP`) será igual al del computador básico, esto es: siempre apuntará **una dirección arriba del tope del *stack***.
- Si bien se podría incluir la modificación desarrollada en la pregunta anterior para reducir la cantidad de ciclos por instrucción, se optará por no utilizarla para no complejizar el *hardware*.

Con los puntos anteriores en consideración, se muestra un posible diagrama para la máquina de *stack*:



Por otra parte, la ISA que cubre todo lo descrito en el enunciado y que puede ejecutarse en la máquina elaborada es la siguiente:

Instrucción	Operandos	*Opcode*	Laux	Sop	Sdin	W	IncSp	DecSp
PUSH	LIT	00000	0	-	LIT	1	0	1
POP	-	00001	0	-	-	0	1	0
ADD	-	00001	0	-	-	0	1	0
	-	00010	1	-	-	0	1	0
	-	00011	0	ADD	ALU	1	0	1
SUB	-	00001	0	-	-	0	1	0
	-	00010	1	-	-	0	1	0
	-	00100	0	SUB	ALU	1	0	1
AND	-	00001	0	-	-	0	1	0
	-	00010	1	-	-	0	1	0
	-	00101	0	AND	ALU	1	0	1
OR	-	00001	0	-	-	0	1	0
	-	00010	1	-	-	0	1	0
	-	00110	0	OR	ALU	1	0	1
XOR	-	00001	0	-	-	0	1	0
	-	00010	1	-	-	0	1	0
	-	00111	0	XOR	ALU	1	0	1
NOT	-	00001	0	-	-	0	1	0
	-	01000	0	NOT	ALU	1	0	1
SHL	-	00001	0	-	-	0	1	0
	-	01001	0	SHL	ALU	1	0	1
SHR	-	00001	0	-	-	0	1	0
	-	01010	0	SHR	ALU	1	0	1

De la ISA provista se destaca lo siguiente:

- El *opcode* de la instrucción `POP` se reutiliza en todas las instrucciones que requieren de la ALU ya que se necesita para posicionar al *stack pointer* en la dirección del tope del *stack*.
- El *opcode* `00010` carga en el registro `Aux` el valor al que apunta el *stack pointer* en dicha iteración. En este caso, siempre se ocupa después de posicionarlo en el último elemento del *stack*. Aquí almacenamos temporalmente el tope para utilizarlo como *input B* de la ALU.
- La última operación de cada instrucción se encarga de posicionar el *stack pointer* una dirección sobre el tope del *stack*.

Distribución de puntaje

- **2 puntos** si el diagrama permite implementar correctamente el ingreso de literal y descarte del tope del *stack*. Se otorga la mitad del puntaje si falla en algún caso.
- **4 puntos** si el diagrama permite implementar correctamente instrucciones que ejecutan operaciones de la ALU de un operando. Se otorga la mitad del puntaje si falla en algún caso.
- **4 puntos** si el diagrama permite implementar correctamente instrucciones que ejecutan operaciones de la ALU de dos operandos. Se otorga la mitad del puntaje si falla en algún caso.
- **2 puntos** por la inclusión de una tabla de instrucciones correcta con operandos, *opcodes* y señales de control correspondientes.

(c) (8 ptos.) Extienda la implementación de la máquina de *stack* para que incluya saltos condicionales e incondicionales, con la restricción que las *flags* provenientes de la `ALU` solo pueden ser almacenadas dentro del *stack*, no en un registro *Status*. Debe actualizar tanto el diagrama de la microarquitectura como el *set* de instrucciones.

Solución: Para esta implementación, necesitamos lo siguiente:

- Una conexión entre el valor literal de las instrucciones y el *Program Counter*. Esto permitirá poder cargar direcciones de instrucciones para ejecutar los saltos (en complemento con una señal de carga L_{pc}).
- Una conexión entre las *flags* de la ALU y la memoria de *stack*. En este caso, agregaremos un nuevo componente `Extend` cuya única función será agregar un *padding* de 4 bits para que, en conjunto con las *flags*, se genere una palabra de 8 bits que pueda ser almacenada.
- Una conexión entre la memoria de *stack* y la unidad de control. Esto será importante solo para las instrucciones de salto condicional, donde el dato transmitido corresponderá al conjunto de *flags* para evaluar la condición.

Lo anterior es lo mínimo requerido para implementar saltos incondicionales y condicionales. Adicionalmente, agregamos lo siguiente:

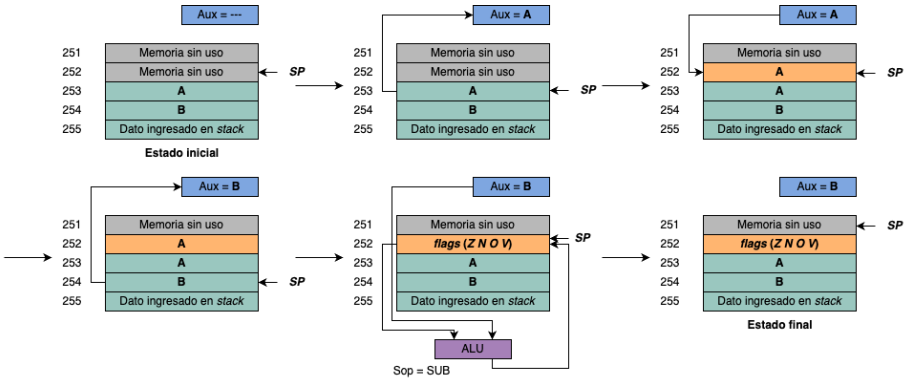
- Componente `Mux Aux` para escoger como segundo *input* de la ALU entre el valor del registro auxiliar y el valor literal asociado a la instrucción. Esto se utilizará para hacer instrucciones `CMP` con valor literal. Si bien esto también añade la capacidad de operar el tope del *stack* con literales para operaciones de dos operandos (por ejemplo, `ADD Lit`), en este caso nos enfocaremos netamente en las instrucciones de salto.
- Componente `Mux Stack` para escoger como primer *input* de la ALU entre la salida de la memoria de *stack* y el valor constante cero. Esto se utilizará para copiar valores del registro auxiliar en la memoria de *stack*.

Respecto a la instrucción `CMP`, existen dos formas de hacerlo: con una comparación entre los dos valores al tope del *stack*; y con una comparación entre el valor del tope y un literal. En este caso se incluirán ambas opciones y en ambos casos lo que harán será almacenar las *flags* de estado de la ALU en una posición sobre el tope del *stack*. A continuación, se describe la ejecución de cada una en conjunto con un diagrama:

`CMP`

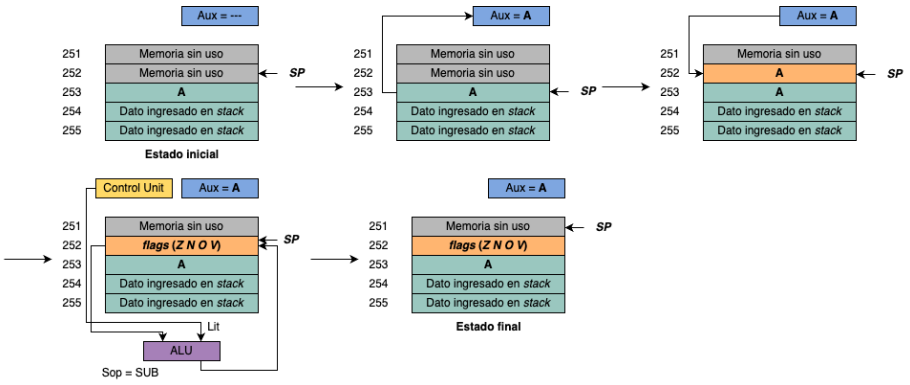
1. Se incrementa en una unidad el contador `SP` para que apunte al valor del tope del *stack* (que llamaremos A).
2. Se copia el valor A al registro `Aux` y se decrementa en una unidad el contador `SP`.
3. Se copia el valor A en la dirección `SP` actual (siendo esta copia el nuevo tope del *stack*) y vuelve a incrementarse en una unidad `SP`.

4. Se incrementa en una unidad el contador `SP` para que apunte al segundo valor del tope del *stack* (que llamaremos B).
5. Se copia el valor B al registro `Aux` y se decrementa en una unidad el contador `SP`.
6. Se decrementa en una unidad el contador `SP` para que apunte al nuevo valor del tope del *stack* (que de momento será A).
7. Se reemplaza la copia A por las *flags* de la ALU a partir de la operación $A - B$ (A proveninente de la memoria; B proveniente del registro auxiliar) y se decrementa en una unidad el contador `SP` para que esté una dirección sobre el nuevo tope del *stack*.



`CMP Lit`

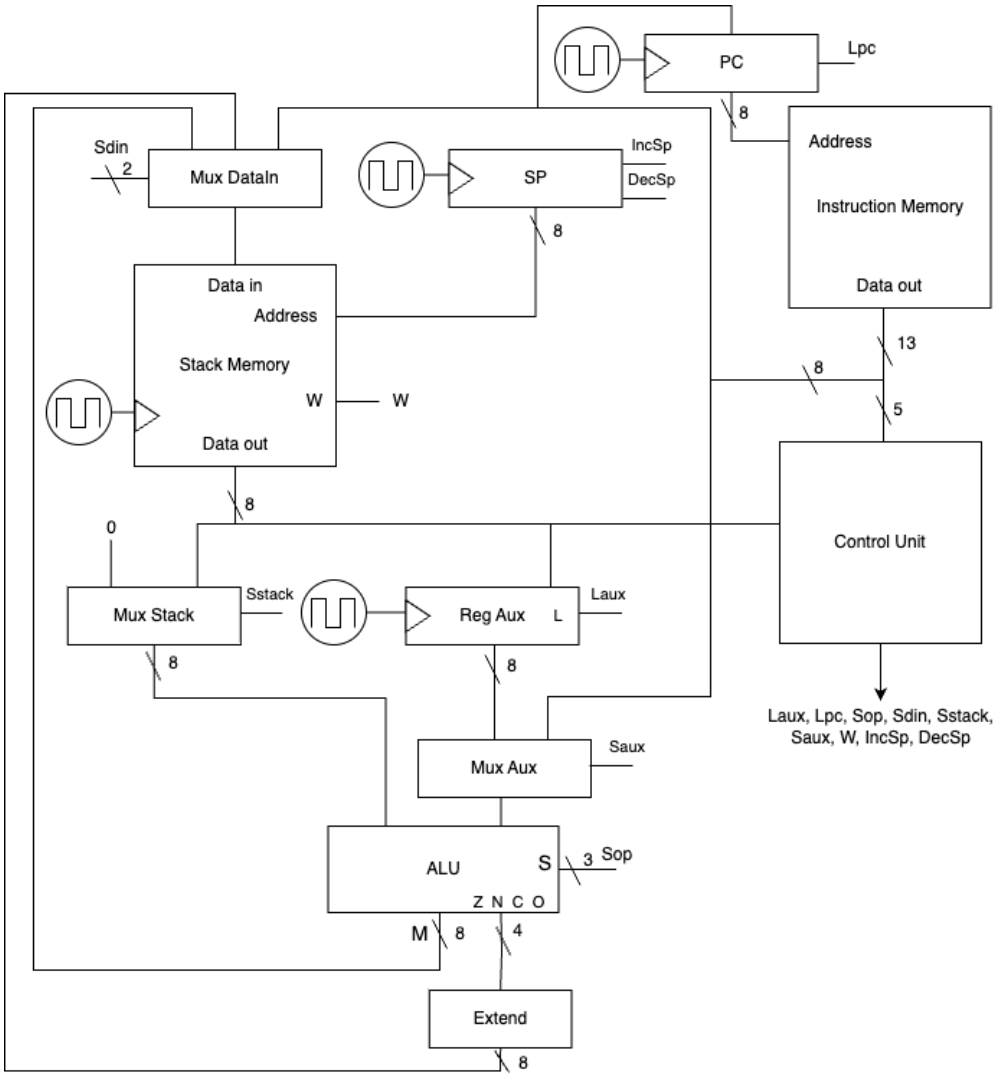
1. Se incrementa en una unidad el contador `SP` para que apunte al valor del tope del *stack* (que llamaremos A).
2. Se copia el valor A al registro `Aux` y se decrementa en una unidad el contador `SP`.
3. Se copia el valor A en la dirección `SP` actual (siendo esta copia el nuevo tope del *stack*) y se mantiene el contador `SP` sin modificaciones.
4. Se reemplaza la copia A por las *flags* de la ALU a partir de la operación $A - Lit$ (A proveniente de la memoria; Lit proveniente de la memoria de instrucciones) y se decrementa en una unidad el contador `SP` para que esté una dirección sobre el nuevo tope del *stack*.



Finalmente, las instrucciones de salto condicional asumirán que existe en el tope del *stack* el valor que posee las *flags* de estado y, finalizada su ejecución (independiente de si se lleva a cabo el salto o no), se descartará el valor almacenado mediante el incremento del contador `SP`.

Cabe destacar que si bien la arquitectura se podría complejizar más para reducir la cantidad de ciclos de estas instrucciones (por ejemplo, usando *adders* o *subtractors* para posicionar el contador `SP` de forma directa), en esta respuesta se incluye lo mínimo para cumplir lo solicitado. Por otra parte, si bien es posible almacenar las *flags* de estado en el registro `Aux`, esto atenta contra lo solicitado en el enunciado (almacenamiento válido solo dentro del *stack*).

A continuación, el diagrama de la microarquitectura e ISA resultantes:



Instrucción	Operandos	*Opcode*	Condición	Laux	Lpc	Sstack	Saux	Sop	Sdin	W	IncSp	DecSp
PUSH	LIT	00000	-	0	0	-	-	-	LIT	1	0	1
POP	-	00001	-	0	0	-	-	-	-	0	1	0
ADD	-	00001	-	0	0	-	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	-	0	1	0
	-	00011	-	0	0	DOUT	AUX	ADD	ALU	1	0	1
SUB	-	00001	-	0	0	-	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	-	0	1	0
	-	00100	-	0	0	DOUT	AUX	SUB	ALU	1	0	1
AND	-	00001	-	0	0	-	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	-	0	1	0
	-	00101	-	0	0	DOUT	AUX	AND	ALU	1	0	1
OR	-	00001	-	0	0	-	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	-	0	1	0
	-	00110	-	0	0	DOUT	AUX	OR	ALU	1	0	1
XOR	-	00001	-	0	0	-	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	-	0	1	0
	-	00111	-	0	0	DOUT	AUX	XOR	ALU	1	0	1
NOT	-	00001	-	0	0	-	-	-	-	0	1	0
	-	01000	-	0	0	DOUT	AUX	NOT	ALU	1	0	1
SHL	-	00001	-	0	0	-	-	-	-	0	1	0
	-	01001	-	0	0	DOUT	AUX	SHL	ALU	1	0	1
SHR	-	00001	-	0	0	-	-	-	-	0	1	0
	-	01010	-	0	0	DOUT	AUX	SHR	ALU	1	0	1
CMP	-	00001	-	0	0	-	-	-	-	0	1	0
	-	01011	-	1	0	-	-	-	-	0	0	1
	-	01100	-	0	0	ZERO	AUX	ADD	ALU	1	1	0
	-	00001	-	0	0	-	-	-	-	0	1	0
	-	01011	-	1	0	-	-	-	-	0	0	1
	-	01101	-	0	0	-	-	-	-	0	0	1
	-	01110	-	0	0	DOUT	AUX	SUB	EXT	1	0	1
	LIT	00001	-	0	0	-	-	-	-	0	1	0
		01011	-	1	0	-	-	-	-	0	0	1
		01111	-	0	0	ZERO	AUX	ADD	ALU	1	0	0
		10000	-	0	0	DOUT	LIT	SUB	EXT	1	0	1
JMP	LIT	10001	-	0	1	-	-	-	-	0	0	0
JEQ	LIT	00001	-	0	0	-	-	-	-	0	1	0
		10010	Z = 1	0	1	-	-	-	-	0	0	0
JNE	LIT	00001	-	0	0	-	-	-	-	0	1	0
		10011	Z = 0	0	1	-	-	-	-	0	0	0
JGT	LIT	00001	-	0	0	-	-	-	-	0	1	0
		10100	N = 0 y Z = 0	0	1	-	-	-	-	0	0	0
JLT	LIT	00001	-	0	0	-	-	-	-	0	1	0
		10101	N = 1	0	1	-	-	-	-	0	0	0
JGE	LIT	00001	-	0	0	-	-	-	-	0	1	0
		10110	N = 0	0	1	-	-	-	-	0	0	0
JLE	LIT	00001	-	0	0	-	-	-	-	0	1	0
		10111	N = 0 o Z = 1	0	1	-	-	-	-	0	0	0
JCR	LIT	00001	-	0	0	-	-	-	-	0	1	0
		11000	C = 1	0	1	-	-	-	-	0	0	0
JOV	LIT	00001	-	0	0	-	-	-	-	0	1	0
		11001	V = 1	0	1	-	-	-	-	0	0	0

Distribución de puntaje

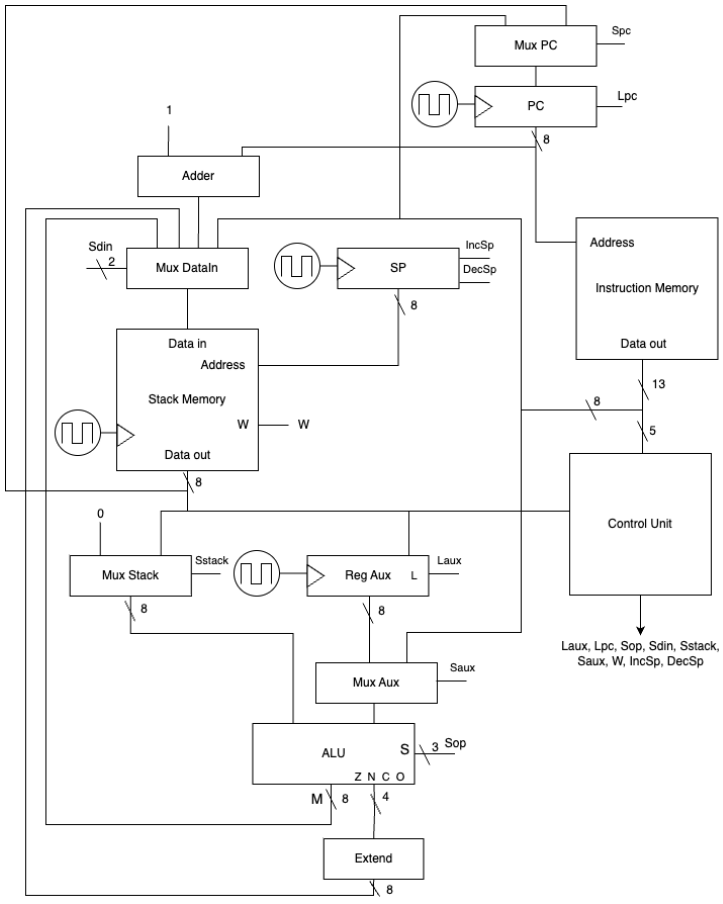
- **2 puntos** si el diagrama permite implementar correctamente la carga dentro del registro `PC` para saltos condicionales e incondicionales. Se otorga la mitad del puntaje si falla en algún caso.
- **4 puntos** si el diagrama permite implementar correctamente el ingreso de las *flags* de estado de la ALU a la memoria de *stack* (a través de la comparación de los dos valores al tope del *stack* o el tope con un literal). **Debe explicarse cómo se lleva a cabo**. Se otorga la mitad del puntaje si falla en algún caso.
- **2 puntos** por la inclusión de una tabla de instrucciones correcta con operandos, *opcodes* y señales de control correspondientes.

(d) (4 ptos.) Extienda la implementación de la máquina de *stack* para dar soporte a la ejecución de subrutinas. Debe actualizar tanto el diagrama de la microarquitectura como el *set* de instrucciones.

Solución: Para esta implementación, necesitamos lo siguiente:

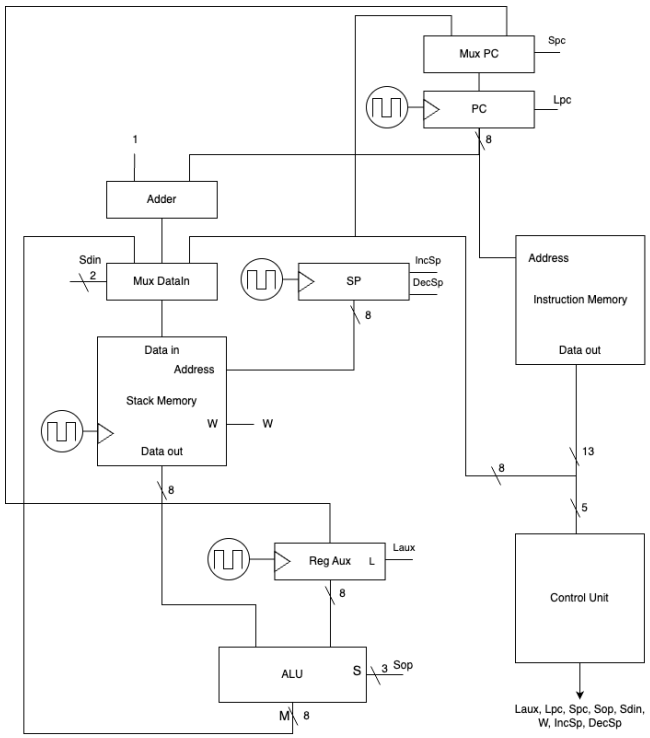
- Una conexión entre el *Program Counter* y la memoria de *stack*. Esto permitirá poder almacenar la dirección de retorno de una subrutina y, asimismo, cargarla en el contador `PC` para su ejecución.
- Una conexión entre el valor literal de las instrucciones y el *Program Counter*. Esto permitirá cargar direcciones de instrucciones para ejecutar los llamados a subrutinas. Si se respondió la pregunta anterior, esta conexión ya debiese existir.

Como ahora tendremos dos valores posibles a cargar en el contador `PC` (literal o dirección de retorno desde la memoria), debemos añadir un componente `Mux PC` para seleccionar el valor a cargar (si corresponde). De esta manera, el diagrama e ISA finales son los siguientes:



Instrucción	Operandos	*Opcode*	Condición	Laux	Lpc	Spc	Sstack	Saux	Sop	Sdin	W	IncSp	DecSp
PUSH	LIT	00000	-	0	0	-	-	-	-	LIT	1	0	1
POP	-	00001	-	0	0	-	-	-	-	-	0	1	0
ADD	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	-	-	0	1	0
	-	00011	-	0	0	-	DOUT	AUX	ADD	ALU	1	0	1
SUB	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	-	-	0	1	0
	-	00100	-	0	0	-	DOUT	AUX	SUB	ALU	1	0	1
AND	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	-	-	0	1	0
	-	00101	-	0	0	-	DOUT	AUX	AND	ALU	1	0	1
OR	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	-	-	0	1	0
	-	00110	-	0	0	-	DOUT	AUX	OR	ALU	1	0	1
XOR	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	-	-	0	1	0
	-	00111	-	0	0	-	DOUT	AUX	XOR	ALU	1	0	1
NOT	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	01000	-	0	0	-	DOUT	AUX	NOT	ALU	1	0	1
SHL	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	01001	-	0	0	-	DOUT	AUX	SHL	ALU	1	0	1
SHR	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	01010	-	0	0	-	DOUT	AUX	SHR	ALU	1	0	1
CMP	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	01011	-	1	0	-	-	-	-	-	0	0	1
	-	01100	-	0	0	-	ZERO	AUX	ADD	ALU	1	1	0
	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	01011	-	1	0	-	-	-	-	-	0	0	1
	-	01101	-	0	0	-	-	-	-	-	0	0	1
	-	01110	-	0	0	-	DOUT	AUX	SUB	EXT	1	0	1
	LIT	00001	-	0	0	-	-	-	-	-	0	1	0
		01011	-	1	0	-	-	-	-	-	0	0	1
		01111	-	0	0	-	ZERO	AUX	ADD	ALU	1	0	0
		10000	-	0	0	-	DOUT	LIT	SUB	EXT	1	0	1
JMP	LIT	10001	-	0	1	LIT	-	-	-	-	0	0	0
JEQ	LIT	00001	-	0	0	-	-	-	-	-	0	1	0
		10010	Z = 1	0	1	LIT	-	-	-	-	0	0	0
JNE	LIT	00001	-	0	0	-	-	-	-	-	0	1	0
		10011	Z = 0	0	1	LIT	-	-	-	-	0	0	0
JGT	LIT	00001	-	0	0	-	-	-	-	-	0	1	0
		10100	N = 0 y Z = 0	0	1	LIT	-	-	-	-	0	0	0
JLT	LIT	00001	-	0	0	-	-	-	-	-	0	1	0
		10101	N = 1	0	1	LIT	-	-	-	-	0	0	0
JGE	LIT	00001	-	0	0	-	-	-	-	-	0	1	0
		10110	N = 0	0	1	LIT	-	-	-	-	0	0	0
JLE	LIT	00001	-	0	0	-	-	-	-	-	0	1	0
		10111	N = 0 o Z = 1	0	1	LIT	-	-	-	-	0	0	0
JCR	LIT	00001	-	0	0	-	-	-	-	-	0	1	0
		11000	C = 1	0	1	LIT	-	-	-	-	0	0	0
JOV	LIT	00001	-	0	0	-	-	-	-	-	0	1	0
		11001	V = 1	0	1	LIT	-	-	-	-	0	0	0
CALL	LIT	11010	-	0	1	LIT	-	-	-	PC	1	0	1
RET	-	00001	-	0	0	-	-	-	-	-	0	1	0
	-	11011	-	0	1	DOUT	-	-	-	-	0	0	0

Alternativamente, en caso de no responder la pregunta 2.c., se puede realizar esta implementación sin el soporte de saltos condicionales (pero sí los incondicionales). A continuación, se muestra el diagrama e ISA de dicho caso:



Instrucción	Operandos	*Opcode*	Condición	Laux	Lpc	Spc	Sop	Sdin	W	IncSp	DecSp
PUSH	LIT	00000	-	0	0	-	-	LIT	1	0	1
POP	-	00001	-	0	0	-	-	-	0	1	0
ADD	-	00001	-	0	0	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	0	1	0
	-	00011	-	0	0	-	ADD	ALU	1	0	1
SUB	-	00001	-	0	0	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	0	1	0
	-	00100	-	0	0	-	SUB	ALU	1	0	1
AND	-	00001	-	0	0	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	0	1	0
	-	00101	-	0	0	-	AND	ALU	1	0	1
OR	-	00001	-	0	0	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	0	1	0
	-	00110	-	0	0	-	OR	ALU	1	0	1
XOR	-	00001	-	0	0	-	-	-	0	1	0
	-	00010	-	1	0	-	-	-	0	1	0
	-	00111	-	0	0	-	XOR	ALU	1	0	1
NOT	-	00001	-	0	0	-	-	-	0	1	0
	-	01000	-	0	0	-	NOT	ALU	1	0	1
SHL	-	00001	-	0	0	-	-	-	0	1	0
	-	01001	-	0	0	-	SHL	ALU	1	0	1
SHR	-	00001	-	0	0	-	-	-	0	1	0
	-	01010	-	0	0	-	SHR	ALU	1	0	1
JMP	LIT	01011	-	0	1	LIT	-	-	0	0	0
CALL	LIT	01100	-	0	1	LIT	-	PC	1	0	1
RET	-	00001	-	0	0	-	-	-	0	1	0
	-	01101	-	0	1	DOUT	-	-	0	0	0

Distribución de puntaje

- **2 puntos** si el diagrama permite implementar correctamente subrutinas, independiente de los saltos condicionales. Se otorga la mitad del puntaje si falla en algún caso.
- **2 puntos** por la inclusión de una tabla de instrucciones correcta con operandos, *opcodes* y señales de control correspondientes.

Pregunta 3: Assembly (14 ptos.)

(a) (14 ptos.) Implemente en *Assembly* del computador básico visto en clases, el algoritmo para ordenar arreglos de enteros *min-max sort*. Este algoritmo opera buscando el máximo y el mínimo de un arreglo, a continuación los coloca en los extremos correspondientes, para luego repetir el proceso con el sub-arreglo que no contiene a los extremos. Cualquier cosa que asuma con respecto al ejercicio debe quedar claramente especificada.

Solución: En el siguiente programa elaborado con el Assembly del computador básico se considera lo siguiente:

- Solo se aceptan números en el intervalo $[-64, 63]$. Los números que se encuentren fuera del intervalo $[-128, 127]$ no serán correctamente interpretados por los simuladores (al requerir más de 8 bits para su representación), mientras que los números del intervalo $[-128, 127] - [-64, 63]$ tienen la particularidad de que pueden generar *overflow* en la ejecución de la instrucción `CMP` (por ejemplo, `CMP 64, -64`).
- En línea con el punto anterior, se asume que el largo de los arreglos es menor a 128 para poder representar bien esta cantidad de forma numérica dentro del código.
- Se recorre el arreglo dos veces por iteración: una para encontrar el mínimo y una para encontrar el máximo. Esto se puede hacer de forma simultánea, pero para facilitar la lectura del código se hace la separación de las búsquedas.
- Se realiza un *swap* directo entre los elementos del arreglo en memoria en vez de almacenar un segundo arreglo con la ordenación final. Ambas formas están correctas, pero esta última es más limitada debido a que restringe la memoria de datos considerablemente según el largo de los arreglos.
- Se asume que la cantidad de variables definida no excede la cantidad de direcciones disponibles en la memoria de datos ($2^8 = 256$).
- Se controlan los casos de arreglos con una cantidad par e impar de elementos. Asimismo, se controla el caso donde se tiene un arreglo de un elemento. No se evaluará, pero se esperaba que se controlara este caso.

```
DATA:
  arr 3
      41
      -5
      0
      -27
      0
      11
  len 7
  sub_arr_min_index 0
  sub_arr_max_index 0
  iteration_min_index 0
  iteration_max_index 0
  min_val_temp 0
  max_val_temp 0
```

```
CODE:
  // Inicialización de variables auxiliares: sub_arr_max_index, iteration_max_index
  var_init:
    MOV A,(len)
    SUB A,1
    MOV (sub_arr_max_index),A
    MOV (iteration_max_index),A
    MOV B,(sub_arr_min_index)
  // Condición de término: Índice de la cota superior del sub-arreglo menor o igual al índice de la
  // cota inferior.
  end_check:                        // A = sub_arr_max_index; B = sub_arr_min_index
    CMP A,B                         // sub_arr_max_index <= sub_arr_min_index -> end.
    JLE end
    // Si no se cumple la condición, se establecen los índices iniciales nuevos y comienza la
    // ejecución.
    MOV (iteration_max_index),A
    MOV (iteration_min_index),B
  // Inicialización del mínimo temporal del sub-arreglo en la iteración.
  temp_min_set:
    MOV A,(iteration_min_index)
    MOV B,arr
    ADD B,A
    MOV A,(B)                       // A = sub_arr[0]
    MOV (min_val_temp),A            // min_val_temp = sub_arr[0]
    MOV A,(iteration_min_index)
    JMP min_iteration
  // Subrutina que intercambia el mínimo del sub-arreglo (sub_arr[0]) por un valor menor encontrado
  // dentro de este (sub_arr[i]).
  min_swap:
    MOV B,arr
    MOV A,(iteration_min_index)
    ADD B,A
    MOV A,B
    MOV B,(B)
    PUSH B                          // Mem[SP] = sub_arr[i] = new_min_value
    MOV B,A
    MOV A,(min_val_temp)
    MOV (B),A                       // sub_arr[i] = old_min_value
    MOV B,arr
    MOV A,(sub_arr_min_index)
    ADD B,A
    POP A                           // A = new_min_value
    MOV (B),A                       // sub_arr[0] = new_min_value
    MOV (min_val_temp),A            // min_val_temp = new_min_value
    RET
  // Se recorre el sub-arreglo buscando el valor mínimo y se intercambia si corresponde.
  min_iteration:
    ADD A,1
    MOV (iteration_min_index),A
    MOV B,arr
    ADD B,A
    MOV A,(min_val_temp)            // A = sub_arr[0]
    MOV B,(B)                       // B = sub_arr[i]
    CMP A,B
    JLE continue_min_without_swap   // if sub_arr[0] <= sub_arr[i] evitar swap.
    CALL min_swap
  // Independiente de si se intercambian valores o no, se revisa si terminó el recorrido del
  // sub-arreglo.
  // Si se termina, empieza la búsqueda del máximo. En otro caso continua la iteración con la
  // siguiente posición.
  continue_min_without_swap:
    MOV A,(iteration_min_index)
    MOV B,(sub_arr_max_index)
    CMP A,B
    JLT min_iteration               // if iteration_min_index < sub_arr_max_index -> min_iteration
  // else -> temp_max_set && max_iteration
```

```
temp_max_set:
  MOV A,(iteration_max_index)
  MOV B,arr
  ADD B,A
  MOV A,(B)                       // A = sub_arr[-1]
  MOV (max_val_temp),A            // max_val_temp = sub_arr[-1]
  MOV A,(iteration_max_index)
  JMP max_iteration
// Subrutina que intercambia el máximo del sub-arreglo (sub_arr[-1]) por un valor mayor
// encontrado dentro de este (sub_arr[i]).
max_swap:
  MOV B,arr
  MOV A,(iteration_max_index)
  ADD B,A
  MOV A,B
  MOV B,(B)
  PUSH B                          // Mem[SP] = sub_arr[i] = new_max_value
  MOV B,A
  MOV A,(max_val_temp)
  MOV (B),A                       // sub_arr[i] = old_max_value
  MOV B,arr
  MOV A,(sub_arr_max_index)
  ADD B,A
  POP A                           // A = new_max_value
  MOV (B),A                       // sub_arr[-1] = new_max_value
  MOV (max_val_temp),A            // max_val_temp = new_max_value
  RET
// Se recorre el sub-arreglo buscando el valor máximo y se intercambia si corresponde.
max_iteration:
  SUB A,1
  MOV (iteration_max_index),A
  MOV B,arr
  ADD B,A
  MOV A,(max_val_temp)            // A = sub_arr[-1]
  MOV B,(B)                       // B = sub_arr[i]
  CMP A,B
  JGE continue_max_without_swap   // if sub_arr[-1] >= sub_arr[i] evitar swap.
  CALL max_swap
// Independiente de si se intercambian valores o no, se revisa si terminó el recorrido del
// sub-arreglo.
// Si se termina, se revisa el siguiente sub-arreglo. En otro caso continua la iteración con
// la siguiente posición.
continue_max_without_swap:
  MOV A,(iteration_max_index)
  MOV B,(sub_arr_min_index)
  CMP A,B
  JGT max_iteration               // if iteration_max_index > sub_arr_min_index -> max_iteration
// Se obtienen los índices del siguiente sub-arreglo y se procede a hacer la revisión de término.
next_sub_arr:
  MOV A,(sub_arr_max_index)
  MOV B,(sub_arr_min_index)
  SUB A,1
  INC B
  MOV (sub_arr_max_index),A       // sub_arr_max_index -= 1
  MOV (sub_arr_min_index),B       // sub_arr_min_index += 1
  JMP end_check
// Fin.
end:
  NOP
```

Distribución de puntaje

- **2 puntos** si el programa no se cae (válido solo para entregas donde se haya tratado de resolver el problema).
- **4 puntos** si el algoritmo realiza el *set* del valor menor y mayor, al menos, en la primera iteración.
- **4 puntos** si el algoritmo funciona correctamente en arreglos de largo par.
- **4 puntos** si el algoritmo funciona correctamente en arreglos de largo impar.