DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

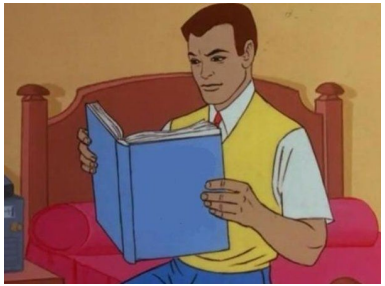
Clase 5 - Saltos y Subrutinas

Profesor: Germán Leandro Contreras Sagredo

Bibliografía

- **Apuntes históricos. Hans Löbel, Alejandro Echeverría**
  - 5 - Saltos y Subrutinas

- **D. Patterson, Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Morgan Kaufmann, 2020.**
  - Capítulos 2.7-2.8. Página 92, 131 en PDF.
  - Capítulos 4.1-4.2-4.3-4.4. Página 236, 320 en PDF.*



* Estos se basan en la implementación de una arquitectura que implementa RISC-V. Tomar solo como referencia.

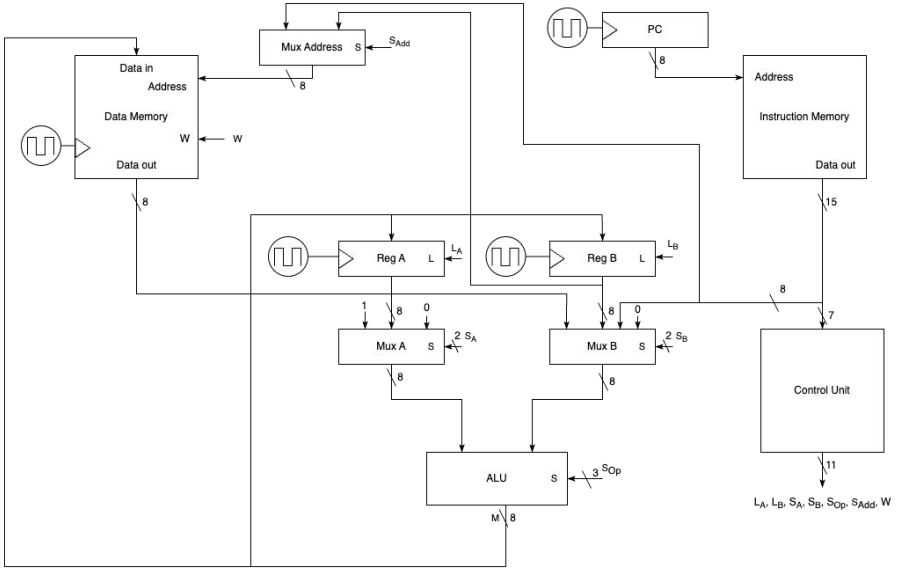
Objetivos de la clase

- Entender el impacto a nivel de código de habilitar saltos y subrutinas en el computador básico.
- Conocer los componentes necesarios para habilitar saltos y subrutinas en el computador básico.
- Realizar ejercicios que consoliden los conocimientos anteriores.

Hasta ahora...

En la figura se ve que tenemos lo necesario para:

- Operar con constantes numéricas (literales).
- Almacenar variables.
- Operar con direccionamiento directo e indirecto.

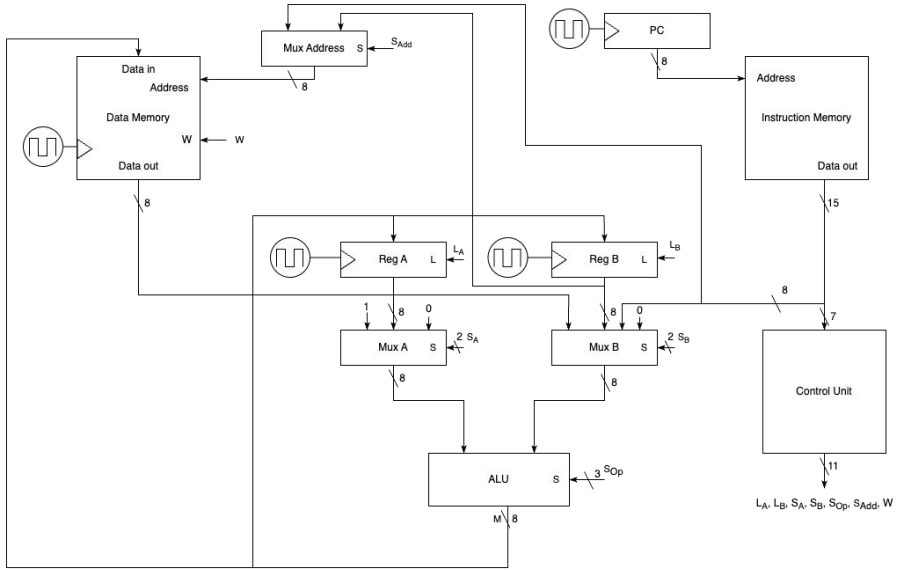


Hasta ahora…

No obstante lo anterior:

- ¿Podemos ejecutar iteraciones? (*for*, *while*)
- ¿Podemos definir funciones y ejecutarlas?

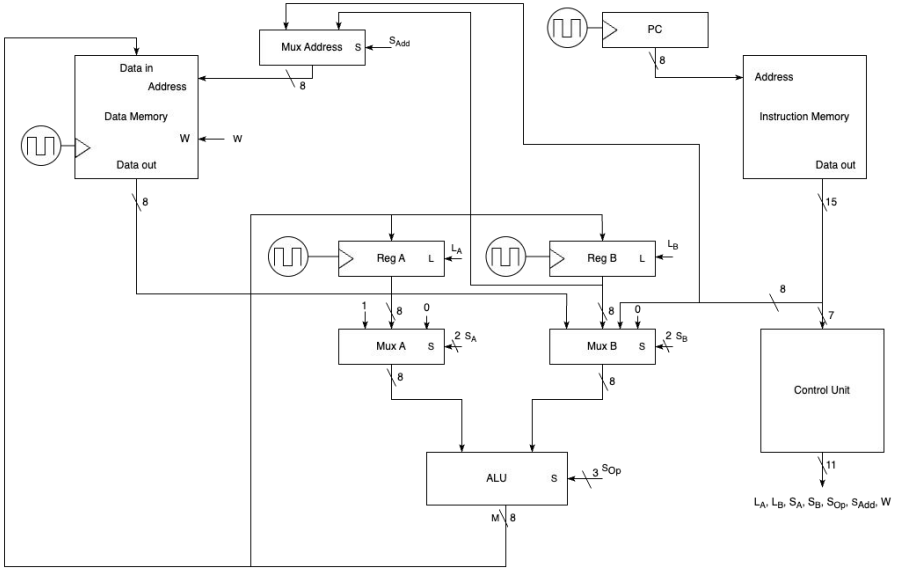
Al otorgar esta capacidad a nuestra máquina, esta será **realmente** programable.



Computador básico - Saltos incondicionales

Entenderemos por “salto” la capacidad de ejecutar, a partir de una instrucción, una línea de código de nuestro programa que **no necesariamente** sea la siguiente de la secuencia.

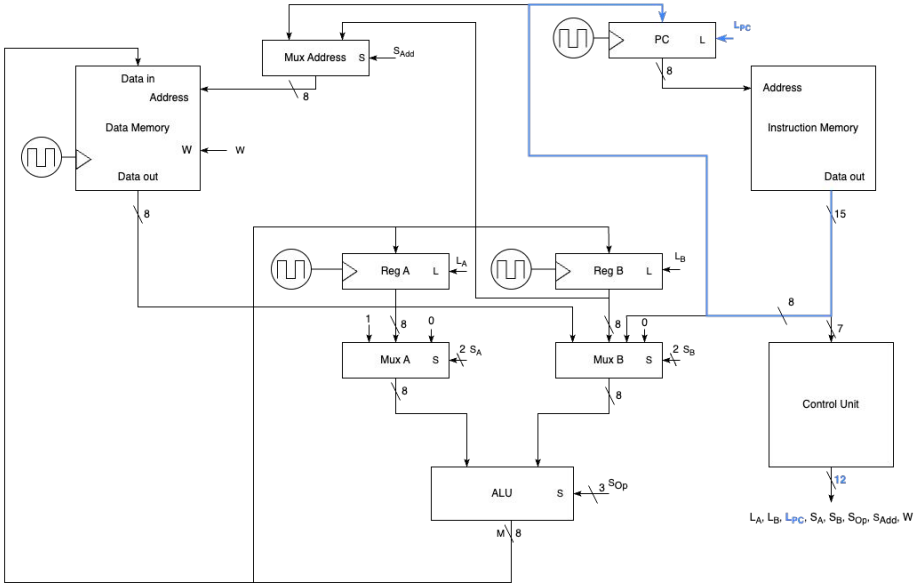
El componente que debemos modificar para este fin es el *Program Counter*.



Computador básico - Saltos incondicionales

En primer lugar, se añade la señal L_{PC} para que el contador PC pueda actualizar su valor según el dato de entrada. En este caso, corresponderá al **literal** de la instrucción.

El literal será la **dirección de memoria de la instrucción que deseamos ejecutar en el siguiente ciclo**.



* En este caso, la señal L_{PC} tiene prioridad por sobre la señal Up del contador PC, lo que permite que, aunque ambas se activen, el resultado sea la carga del bus de entrada y no el incremento del valor actual.

Computador básico - Saltos incondicionales en Assembly

El *hardware* ahora permite definir la instrucción `JMP label`, que realiza un **salto incondicional** a la **primera** instrucción de código asociada a `label`. En el ejemplo, se realiza un salto incondicional hacia `test`, lo que deriva en que el registro *A* almacene el valor 8 y no 12.

```
MOV A,2
JMP test
MOV A,4
test:
  ADD A,2
  SHL A,A
```

El **Assembler** se encarga de hacer la traducción correcta del *label* (en este caso, `test`) al literal correspondiente a la dirección de la primera instrucción a la que se asocia (en este caso, `ADD A,2`).

Computador básico - Saltos condicionales

Consideremos ahora el siguiente código (pseudocódigo en Python). En este caso, se lleva a cabo la iteración dentro del `while` si, y solo si la variable `c` cumple con ser mayor a cero. Es decir, si se cumple la condición, **realizamos un salto a las instrucciones que se encuentran dentro del `while`.**

```
a = 2
b = 3
c = 2
while (c > 0):
    a += b
    c -= 1
```

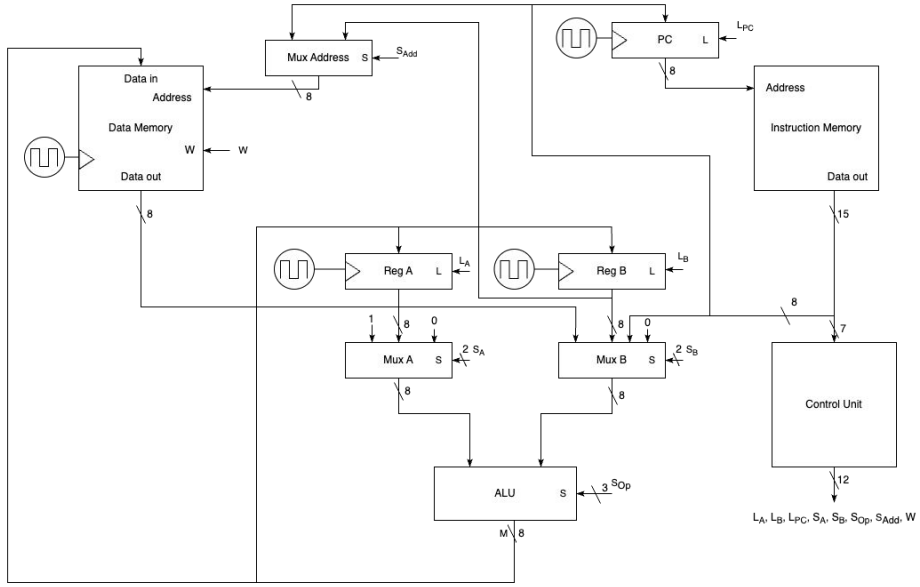
Estos son **saltos condicionales** y veremos cómo incluirlos en nuestros programas a través de modificaciones en el *hardware*.

Computador básico - Saltos condicionales

Para realizar saltos condicionales, necesitamos contar con la capacidad de revisar condiciones.

Estas condiciones pueden provenir de una operación. Por ejemplo: `a > b == a - b > 0`

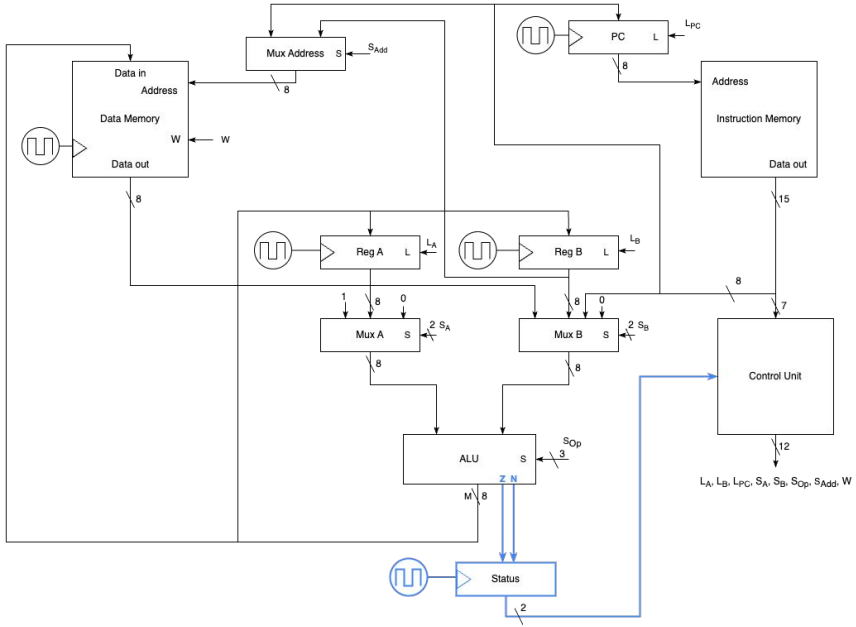
La **ALU** nos puede ayudar a evaluar condiciones **a partir del resultado de su operación**.



Computador básico - Saltos condicionales

De la ALU ahora obtenemos *flags* (señales de 1 bit) que nos indican el cumplimiento o no de una condición. Particularmente:

- **Z:** ***Z****ero*. Indica si el resultado de la operación fue cero o no.
- **N:** ***N****egative*. Indica si el resultado de la operación fue negativo o no.

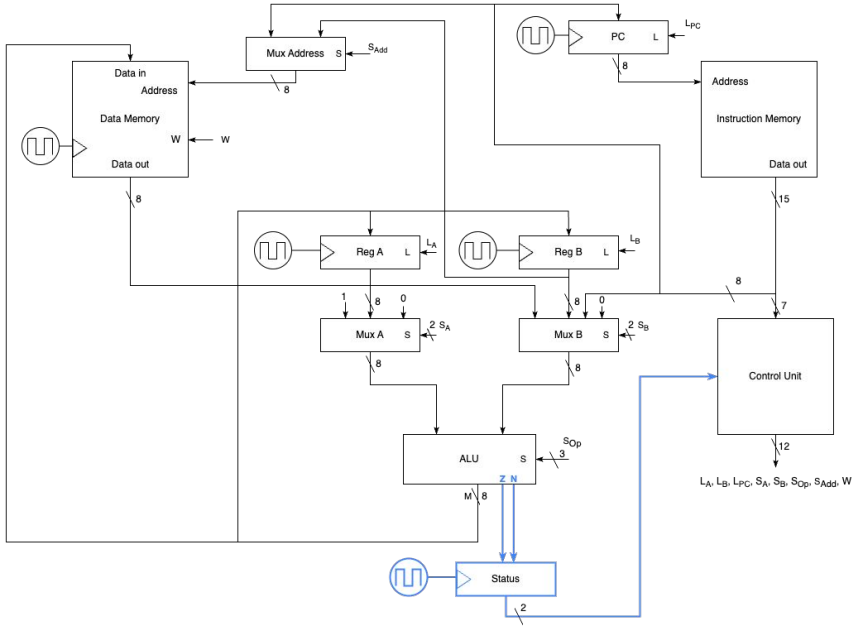


Asumimos que la ALU, en su circuito interno, se encarga de la obtención de estas señales. Por ejemplo, la *flag N* se puede deducir del bit más significativo del número resultante, mientras que la *flag* `Z` se puede obtener a partir de un circuito que verifique todos los bits sean iguales a cero (compuertas `OR` en cadena o pirámide, por ejemplo).

Computador básico - Saltos condicionales

Las *flags* se almacenan en un nuevo registro ***Status*** conectado al *clock* del sistema para poder evaluar la condición **sin que cambie durante un ciclo de ejecución. Siempre** se actualiza, no depende de una señal *Load*.

La *Control Unit* evalúa la condición y realiza el salto o no a través del valor de la señal L_{PC}.

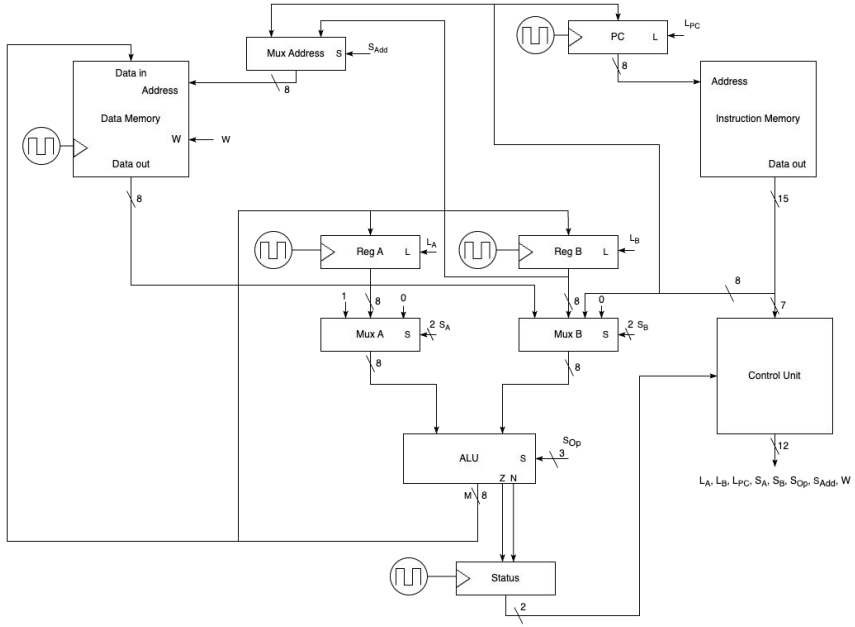


Si no se cumple la condición, la unidad de control propaga la señal $L_{PC} = 0$ para asegurar que se ejecute la siguiente instrucción y no la indicada por el literal. Si se cumple, se propaga la señal $L_{PC} = 1$ y se habilita el salto al sobreescribir la dirección de la instrucción a ejecutar.

Computador básico - Saltos condicionales

Por ahora, estamos habilitando saltos en caso de que el resultado de la operación de la ALU sea negativo o cero.

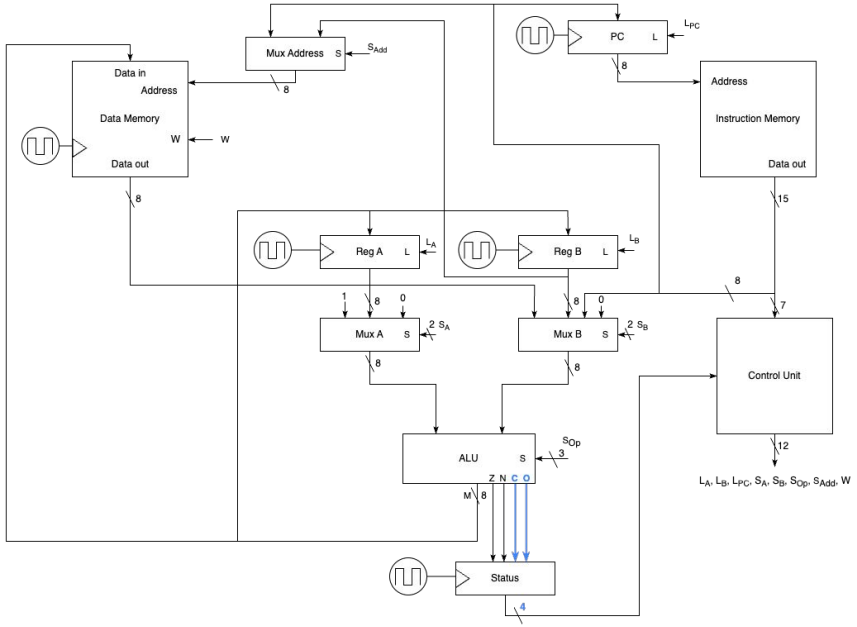
¿Qué otras condiciones podríamos incluir?



Computador básico - Saltos condicionales

- **C: *C*arry**. Indica si hubo un bit de *carry* al finalizar la operación.
- **O: *O*verflow**. Indica si hubo o no *overflow* al finalizar la operación.

Ahondaremos un poco más en estas nuevas *flags*.



Computador básico - *Flags* C y O

***Flag* C:** Es 1 si hubo bit de *carry* en una operación **ADD**/**SUB**, pero también se utiliza para representar el bit "descartado" en operaciones tipo *shift*.
Ejemplo: **SHR**(00101011) → C = 1

***Flag* O:** Es 1 si hubo *overflow* en operaciones **ADD**/**SUB**. Algunas implementaciones también lo consideran en operaciones tipo *shift*, pero no todas. Existen cuatro casos a considerar:

- $A \geq 0; B \geq 0 \rightarrow O = 1$ si $A + B < 0$
- $A < 0; B < 0 \rightarrow O = 1$ si $A + B \geq 0$
- $A \geq 0; B < 0 \rightarrow O = 1$ si $A - B < 0$
- $A < 0; B \geq 0 \rightarrow O = 1$ si $A - B \geq 0$

Computador básico - Saltos condicionales en Assembly

Para computar las *flags* antes de un salto condicional, definimos las instrucciones `CMP` que tienen las siguientes señales de control:

- $L_A = L_B = 0$ (no cargamos datos en los registros).
- $S_A = A$, $S_B = B$, $S_{Op} = SUB$ (operación resta).

De esta forma, se actualiza el registro *Status* a partir de la operación *A - B* **sin actualizar el valor de estos**. La instrucción siguiente a `CMP` **debe** ser de salto condicional para poder evaluar las *flags* de la operación y realizar o no la carga sobre el registro `PC`.

Computador básico - Saltos condicionales en Assembly

A continuación, las instrucciones de salto disponibles.

- `JEQ:` *Jump equal*. Se realiza el salto si $Z = 1$ ($A = B$).
- `JNE:` *Jump not equal*. Se realiza el salto si $Z = 0$ ($A \neq B$).
- `JGT:` *Jump greater than*. Se realiza el salto si $N = 0$ y $Z = 0$ ($A > B$).
- `JLT:` *Jump less than*. Se realiza el salto si $N = 1$ ($A < B$).

Computador básico - Saltos condicionales en Assembly

A continuación, las instrucciones de salto disponibles.

- `JGE:` *Jump greater or equal than*. Se realiza el salto si $N = 0$ ($A \geq B$).
- `JLE:` *Jump less or equal than*. Se realiza el salto si $N = 1$ o $Z = 1$ ($A \leq B$).
- `JCR:` *Jump carry*. Se realiza el salto si $C = 1$ (hubo *carry*).
- `JOV:` *Jump overflow*. Se realiza el salto si $O = 1$ (hubo *overflow*).

Computador básico - Assembly de saltos

Instrucción	Operandos	Operación	Condición	Ejemplo
CPM	A,B	A - B	-	
	A,(B)	A - Mem[B]	-	
	A,Lit	A - Lit	-	CMP A,0
	A,(Dir)	A - Mem[Dir]	-	CMP A,(label)
JMP	Dir	PC = Dir	-	JMP label
JEQ	Dir	PC = Dir	Z = 1	JEQ label
JNE	Dir	PC = Dir	Z = 0	JNE label
JGT	Dir	PC = Dir	Z = 0 y N = 0	JGT label
JLT	Dir	PC = Dir	N = 1	JLT label
JGE	Dir	PC = Dir	N = 0	JGE label
JLE	Dir	PC = Dir	Z = 1 o N = 1	JLE label
JCR	Dir	PC = Dir	C = 1	JCR label
JOV	Dir	PC = Dir	O = 0	JOV label

Ahora que tenemos a nuestra disposición las instrucciones de comparación y saltos, revisemos cómo queda el código de la iteración `while` en Assembly.

Computador básico - Saltos condicionales en Assembly

```
a = 2
b = 3
c = 2
while (c > 0):
    a += b
    c -= 1
```

Programa en pseudocódigo (Python).

```
DATA:
  a 2
  b 3
  c 2
CODE:
  MOV A,(c)
  MOV B,(b)
  while:
    CMP A,0
    JLE end
    SUB A,1
    MOV (c),A
    MOV A,(a)
    ADD A,B
    MOV (a),A
    MOV A,(c)
    JMP while
  end:
    MOV A,(a)
```

Programa en Assembly.

Computador básico - Saltos condicionales en Assembly

```
DATA:
  fibOne 1            ;Fibonacci N
  fibTwo 0            ;Fibonacci N-1
CODE:
  loop:
    MOV A,(fibOne)    ;A = Mem[fibOne]
    MOV B,(fibTwo)    ;B = Mem[fibTwo]
    MOV (fibTwo),A    ;Mem[fibTwo] = A = Fibonacci N
    ADD A,B           ;A += B = Fibonacci N+1
    JOV display       ;O = 1 -> PC = display
    MOV (fibOne),A    ;Mem[fibOne] = A = Fibonacci N+1
    JMP loop
  display:
    MOV A,(fibTwo)    ;A = Último elemento válido
    MOV B,0
  end:
    JMP end
```

Ahora, podemos crear un programa más sofisticado para la obtención de los números de Fibonacci. Se registran los últimos números de la secuencia en las variables `fibOne` y `fibTwo` **según el poder de representación del computador, lo que se determina por *overflow***.

Computador básico - Saltos condicionales en Assembly

```
DATA:
  arr   1           ;arr = [1, 3, 5]
        3
        5
  len   3           ;len = len(arr)
  index 0           ;index = arr[index]
CODE:
  MOV B,arr         ;B = arr = dirección memoria arr
  loop:
    MOV A,(B)       ;A = Mem[B] = Mem[arr]
    SHL A,A         ;A = SHL A = A * 2
    MOV (B),A       ;Mem[B] = A = Mem[arr] * 2
    MOV A,(index)   ;A = Mem[index]
    ADD A,1         ;A += 1
    CMP A,(len)     ;A - Mem[len]
    JGE end         ;N = 0 -> PC = end
    MOV (index),A   ;Mem[index] = A = index + 1
    INC B           ;B += 1 = dirección elemento arr + 1
    JMP loop
  end:
    JMP end
```

También podemos recorrer un arreglo hasta pasar por todos sus elementos. En este caso, duplicamos todos los elementos y terminamos la ejecución cuando el índice es mayor o igual al largo del arreglo (*i.e.* `while index < len`).

Computador básico - Saltos condicionales en Assembly

```
DATA:
  arr   1           ;arr = [1, 3, 5]
        3
        5
  len   3           ;len = len(arr)
  index 0           ;index = arr[index]
CODE:
  MOV A,(index)     ;A = Mem[index] = valor índice
  loop:
    CMP A,(len)     ;A - Mem[len]
    JEQ end         ;Termina si index == largo
    MOV B,arr       ;B = arr = dirección memoria arr
    ADD B,A         ;B = A + B = dirección arr + index
    MOV A,(B)       ;A = Mem[B] = Mem[arr+index]
    SHL A,A         ;A = SHL A = A * 2
    MOV (B),A       ;Mem[arr+index] = Mem[arr+index] * 2
    MOV A,(index)
    ADD A,1
    MOV (index),A
    JMP loop
  end:
    JMP end
```

Otra alternativa para el código anterior. En vez de usar `INC B`, usamos la suma entre la dirección de memoria `arr` y el índice `index` para acceder al elemento de memoria de interés.

Computador básico - Saltos condicionales en Assembly

```
DATA:
  n      5
  result 0
CODE:
  MOV A,(n)          ;A = Mem[n] = valor n
  SHR A,A            ;A = SHR(A)
  JCR is_odd         ;if c == 1 goto is_odd
  is_even:           ;else goto is_even
    MOV B,0          ;B = 0
    MOV (result),B   ;result = 0
    JMP end
  is_odd:
    MOV B,1          ;B = 1
    MOV (result),B   ;result = 1
  end:
    JMP end
```

Otro ejemplo consiste en el uso del bit de *carry*. En este caso, podemos fácilmente detectar si un número es par o impar con un **SHR** y revisando si hubo *carry* o no. Notar que **no es necesario el uso de CMP porque queremos el resultado de la *flag* C posterior al *shift***.

Actividad en clase - Ejercicio de Assembly

Elabore, en Assembly, un programa que encuentre la mediana de un arreglo ordenado de números.

Para realizarlo, puede acceder a la siguiente página, donde podrá compilar y ejecutar código Assembly de forma *online*.

Computador básico - Subrutinas

Consideremos ahora el siguiente código. Se define una función que no se ejecutará de inmediato, sino que una vez que sea llamada. Esto es lo que llamaremos, dentro de nuestro computador básico, una **subrutina**. Con ellas, podemos modularizar nuestro código y optimizarlo aún más.

```
def sum(a, b):
    r = 2*a + b
    return r
a = 3
b = 4
c = sum(a, b)
```

Antes de ver su implementación en *hardware*, veamos los elementos que requieren.

Subrutinas - Requisitos

¿Qué elementos necesita una subrutina?

- Parámetros de entrada.
- Valor de retorno.
- Llamada a la subrutina (salto de ida y de vuelta).

Subrutinas - Parámetros de entrada

Al igual que con las variables, necesitamos almacenar los parámetros de una subrutina. Tenemos dos opciones:

- Almacenamiento en los registros: Rápido procesamiento, pero limitado por su cantidad (en nuestra arquitectura solo se podrían definir subrutinas de dos parámetros).

- Almacenamiento en la memoria de datos: Procesamiento más lento por la lectura, pero sin límite de parámetros. **Utilizaremos esta implementación**.

Subrutinas - Valor de retorno

El valor o valores de retorno de una subrutina presentan la misma logística y opciones que los parámetros de entrada respecto a su almacenamiento:

- Almacenamiento en los registros: Limitado por su cantidad (dos valores de retorno como máximo en este computador).
- Almacenamiento en la memoria de datos: Sin límite para la cantidad de valores de retorno. **Utilizaremos esta implementación**.

Subrutinas - Llamada y retorno

La llamada y el retorno de una subrutina suponen desafíos distintos.

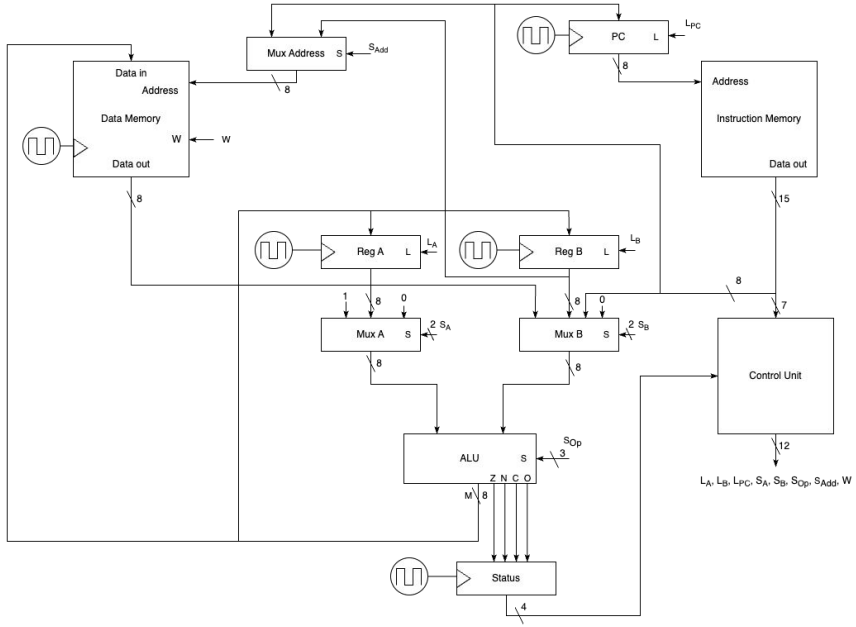
- La llamada se puede implementar como una instrucción `JMP label`, siendo `label` la dirección de memoria de la primera instrucción de la subrutina.

- Para el retorno podemos hacer lo mismo, pero saltando a la instrucción **siguiente** al llamado. ¿Podemos fijar este valor? ¿Qué pasa si realizamos más de un llamado? **Necesitamos almacenar la dirección de la instrucción de retorno**.

Computador básico - Subrutinas

Necesitamos:

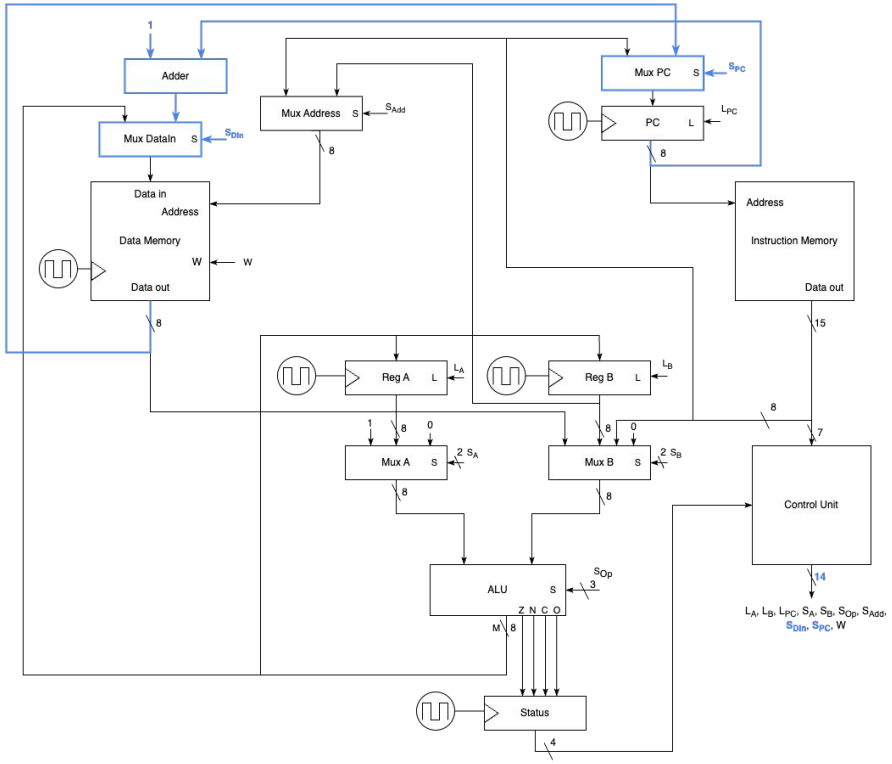
- Conexión entre la memoria de datos y *Program Counter* para cargar direcciones de instrucciones asociadas a *labels*.
- Capacidad para almacenar dirección PC+1 en memoria (dirección de retorno).



Computador básico - Subrutinas

- Con el Mux `DataIn` y la señal S_{DIn} escogemos el dato a almacenar en memoria (valor `PC+1` o resultado de la ALU).

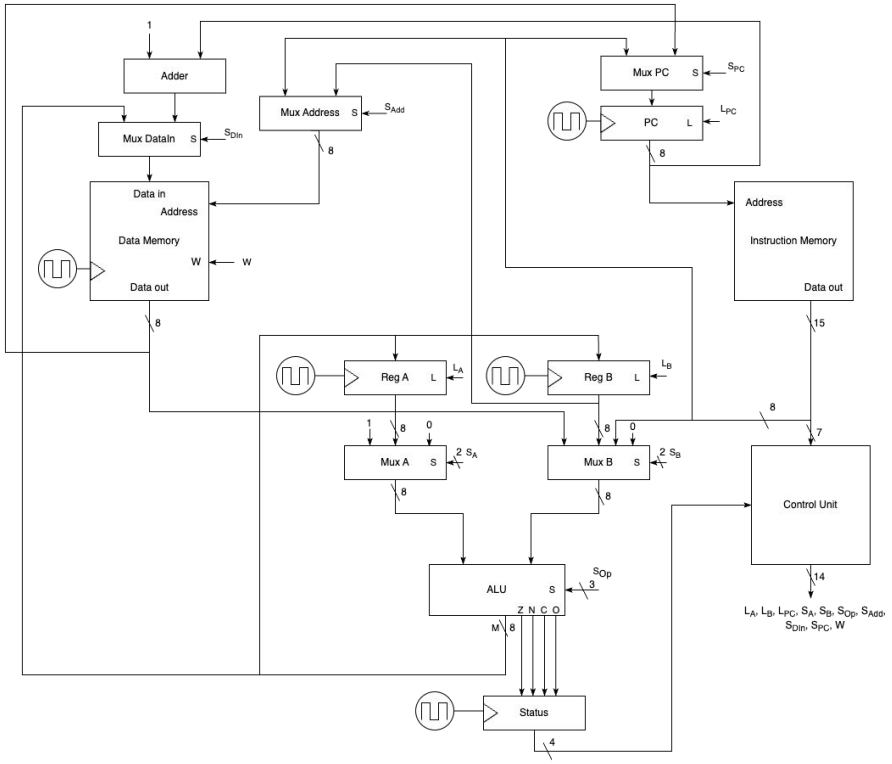
- Con el Mux `PC` y la señal S_{PC} ahora escogemos el dato a cargar en el *Program Counter* (literal o la dirección de la instrucción de retorno de la memoria de datos).



Computador básico - Subrutinas

Ahora, ¿de dónde proviene la dirección de la instrucción de retorno? ¿Cómo nos aseguramos que su almacenamiento no choque con las variables del programa?

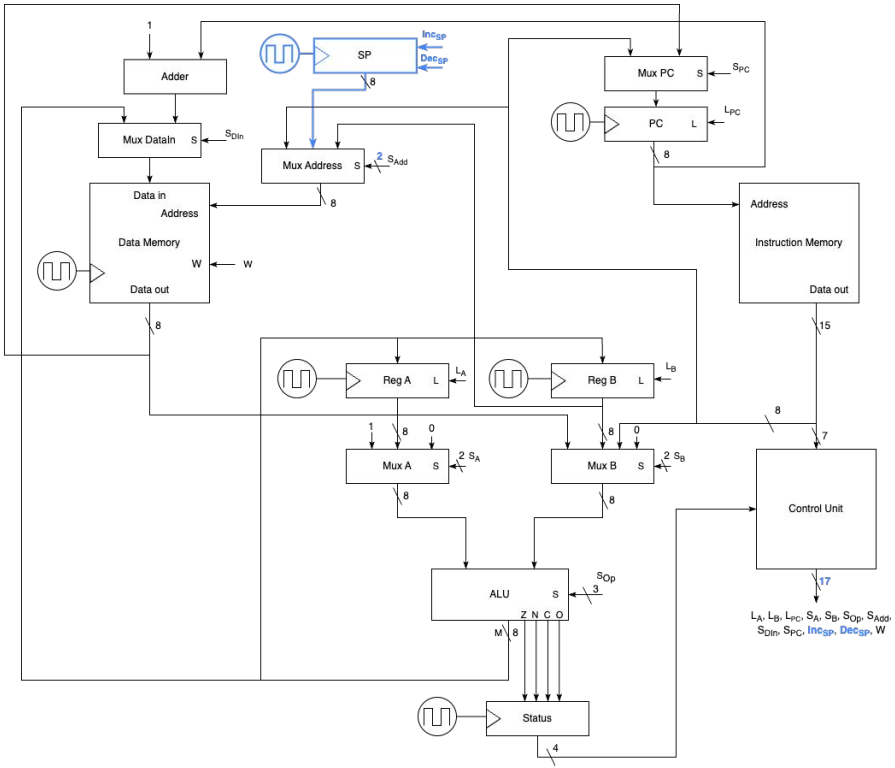
Resolvemos esto a partir de la **memoria de *stack***.



Computador básico - Subrutinas

Se hace uso del segmento inferior de la memoria de datos, siendo la **última dirección** la primera de la memoria de *stack*.

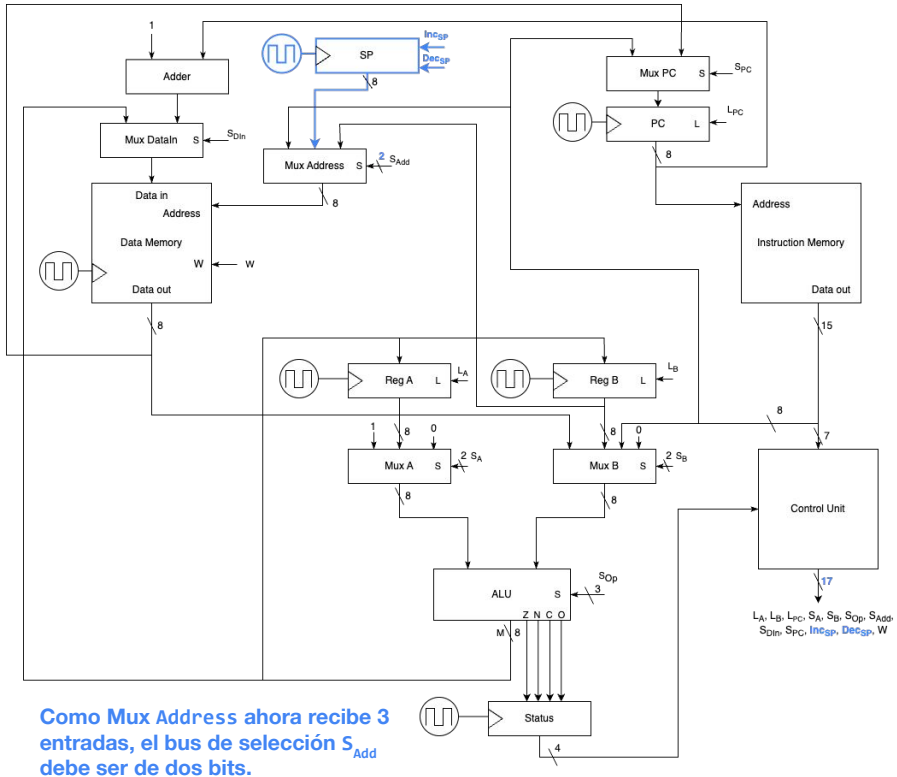
Se accede a estas direcciones desde el ***Stack Pointer*** (SP), contador cuyo valor parte con la última dirección (valor 255 para memorias con bus de 8 bits de direccionamiento).



Computador básico - Subrutinas

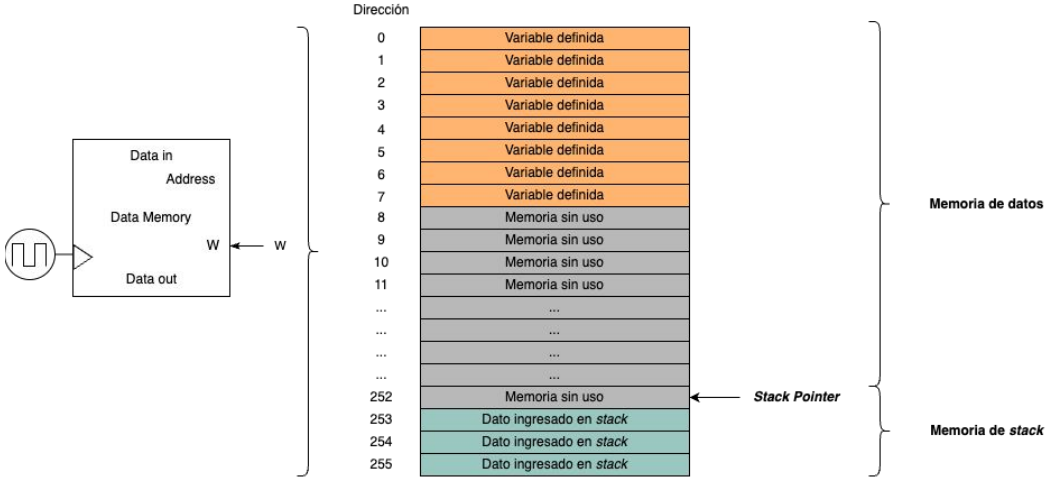
- La señal $\texttt{Dec}_{SP}$ gatilla el decremento en una unidad el contador SP, lo que **aumenta en una unidad el tamaño del *stack***.

- La señal $\texttt{Inc}_{SP}$ gatilla el incremento en una unidad el contador SP, lo que **disminuye en una unidad el tamaño del *stack***.



Computador básico - Memoria de *stack*

La figura muestra cómo se divide la memoria de datos con el uso de *stack*. Cabe destacar que el tamaño del *stack* es **dinámico** y su crecimiento dependerá del programa en ejecución.

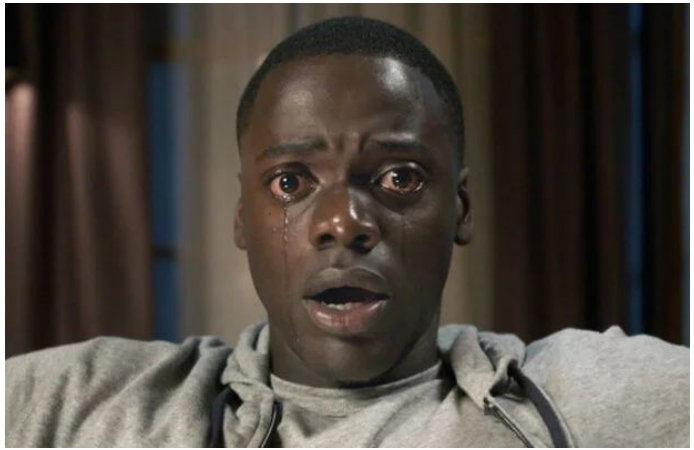


Por construcción, el *stack pointer* apuntará siempre a **una dirección mayor en una unidad** respecto al tamaño del *stack*. Esto **puede** ser distinto en otras arquitecturas.

Computador básico - Memoria de *stack*

¿Qué pasa si la memoria de *stack* crece lo suficiente para **chocar** con el segmento donde se definen las variables?

Se genera un ***stack overflow***.



En la práctica, el tamaño de la memoria de *stack* es fijo y se tiene acceso a la última dirección de memoria disponible, lo que permite detectar el *stack overflow* antes de que ocurra y matar el programa anticipadamente.

Computador básico - Subrutinas en Assembly

Con las modificaciones en *hardware* definimos nuevas instrucciones:

- `CALL label:` Almacena el valor `PC+1` en la dirección de memoria SP; reduce en una unidad el contador `SP` (*i.e.* aumenta el *stack* en una unidad) y carga en `PC` la dirección de memoria asociada a `label` (es decir, salta a la primera instrucción de la subrutina).

- Se realiza en **un ciclo de *clock*** con la siguiente combinación de señales:
  S_{DIn} = PC+1; S_{Add} = SP; W = 1; Dec_{SP} = 1; S_{PC} = LIT; L_{PC} = 1

Computador básico - Subrutinas en Assembly

Con las modificaciones en *hardware* definimos nuevas instrucciones:

- **RET:** Aumenta en una unidad el contador SP (*i.e.* disminuye el *stack* en una unidad); extrae de memoria el valor almacenado en la dirección SP y, finalmente, carga dicho valor en PC (es decir, retorna a la dirección PC+1).

- Se realiza en **dos ciclos de *clock*** de la siguiente forma:
  Ciclo 1: $\texttt{Inc}_{\texttt{SP}} = 1$
  Ciclo 2: $\texttt{S}_{\texttt{Add}} = \texttt{SP}$; $\texttt{S}_{\texttt{PC}} = \texttt{DOUT}$; $\texttt{L}_{\texttt{PC}} = 1$

  Se requieren dos por la necesidad de **actualizar el *stack pointer*.**

Computador básico - Subrutinas en Assembly

Con las modificaciones en *hardware* definimos nuevas instrucciones:

- **`PUSH Reg:`** Almacena en la dirección `SP` el valor del registro `Reg`. Su composición es similar a **`CALL`**, con la salvedad de que en el Mux DataIn se selecciona el resultado de la ALU (que será igual al valor almacenado en `Reg`).

- **`POP Reg:`** Aumenta en una unidad el contador `SP` y almacena en el registro `Reg` el dato almacenado en la dirección `SP`. Su composición es similar a `RET`, con la salvedad de que el *Program Counter* no actualiza su valor, sino que el registro `Reg`.

Computador básico - Assembly de subrutinas

Instrucción	Operandos	Operación	Ejemplo
`CALL`	`Dir`	`Mem[SP] = PC+1; SP--; PC = Dir`	`CALL func`
`RET`	`-`	`SP++`	`-`
		`PC = Mem[SP]`	
`PUSH`	`A`	`Mem[SP] = A; SP--`	`-`
	`B`	`Mem[SP] = B; SP--`	`-`
`POP`	`A`	`SP++`	`-`
		`A = Mem[SP]`	
	`B`	`SP++`	`-`
		`B = Mem[SP]`	

En la práctica, `PUSH` y `POP` se utilizarán para no perder los valores originales de los registros *A* y *B* al momento de ejecutar una subrutina.

Ahora que tenemos a nuestra disposición las instrucciones de comparación y saltos, revisemos cómo queda el código de la función `sum` en Assembly.

Computador básico - Subrutinas en Assembly

```python
def sum(a, b):
    r = 2*a + b
    return r
a = 3
b = 4
c = sum(a, b)
```

Programa en pseudocódigo (Python).

```
DATA:
  a 3
  b 4
  c 0
CODE:
  main: ;código principal
    MOV A,(a)
    MOV B,(b)
    CALL sum
    JPM end
  sum:  ;subrutina sum
    SHL A,A
    ADD A,B
    RET
  end: ;fin de ejecución
    MOV (c),A
```

Programa en Assembly.

Computador básico - Subrutinas en Assembly

```
DATA:
  fibOne 1
  fibTwo 0
CODE:
  JMP start
  next_fibonacci: ;Fibonacci. A = Elemento N y B = Elemento N-1
    MOV (fibTwo),A          ;Mem[fibTwo] = Elemento N
    ADD A,B                 ;A = A + B = Elemento N + 1
    MOV (fibOne),A          ;Mem[fibOne] = Elemento N + 1
    RET
  start:
    MOV A,(fibOne)          ;A = Mem[fibOne] = Elemento N
    MOV B,(fibTwo)          ;B = Mem[fibTwo] = Elemento N - 1
    CALL next_fibonacci     ;Siguiente elemento de la secuencia
    CALL next_fibonacci     ;Siguiente elemento de la secuencia
    CALL next_fibonacci     ;Siguiente elemento de la secuencia
  end:
    JMP end
```

Usemos ahora subrutinas para implementar una nueva versión de Fibonacci.

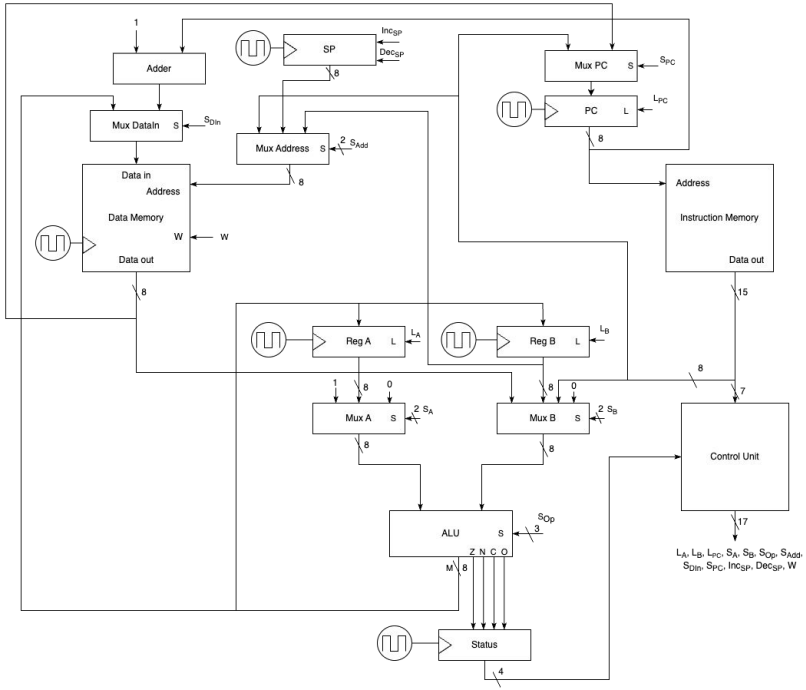
Computador básico - Subrutinas en Assembly

```
DATA:
  fibOne 1
  fibTwo 0
  iterations 3
  currentIteration 0
CODE:
  MOV B,(fibTwo)          ;B = Mem[fibTwo] = Elemento N - 1
  JMP loop
  next_fibonacci:
    MOV (fibTwo),A        ;Mem[fibTwo] = Elemento N
    ADD A,B               ;A = A + B = Elemento N + 1
    MOV (fibOne),A        ;Mem[fibOne] = Elemento N + 1
    MOV B,(fibTwo)        ;B = Mem[fibTwo] = Elemento N
    RET
  loop:
    MOV A,(currentIteration)
    CMP A,(iterations)
    JEQ end               ;iterations = currentIteration -> PC = end
    ADD A,1
    MOV (iterations),A    ;iterations += 1
    MOV A,(fibOne)        ;A = Mem[fibOne] = Elemento N
    CALL next_fibonacci ;Siguiente elemento de la secuencia
    JMP loop
  end:
    JMP end
```

También podemos incorporar ciclos para controlar mejor la cantidad de ejecuciones.

Computador básico - Versión final

Hemos llegado a la versión final del computador básico del curso.



Computador básico - Listado oficial de instrucciones

Instrucción	Operandos	*Opcode*	Condición	L_{PC}	L_A	L_B	$S_A0,1$	$S_B0,1$	$S_{SP}0,1,2$	$S_{Add}0,1$	S_{DIn}	S_{PC}	W	Inc_{SP}	Dec_{SP}
MOV	A,B	0000000		0	1	0	ZERO	B	ADD	-	-	-	0	0	0
	B,A	0000001		0	0	1	A	ZERO	ADD	-	-	-	0	0	0
	A,Lit	0000010		0	1	0	ZERO	LIT	ADD	-	-	-	0	0	0
	B,Lit	0000011		0	0	1	ZERO	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0000100		0	1	0	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	B,(Dir)	0000101		0	0	1	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	(Dir),A	0000110		0	0	0	A	ZERO	ADD	LIT	ALU	-	1	0	0
	(Dir),B	0000111		0	0	0	ZERO	B	ADD	LIT	ALU	-	1	0	0
	A,(B)	0001000		0	1	0	ZERO	DOUT	ADD	B	-	-	0	0	0
	B,(B)	0001001		0	0	1	ZERO	DOUT	ADD	B	-	-	0	0	0
	(B),A	0001010		0	1	0	A	ZERO	ADD	B	ALU	-	1	0	0
ADD	A,B	0001011		0	1	0	A	B	ADD	-	-	-	0	0	0
	B,A	0001100		0	0	1	A	B	ADD	-	-	-	0	0	0
	A,Lit	0001101		0	1	0	A	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0001110		0	1	0	A	DOUT	ADD	LIT	-	-	0	0	0
	A,(B)	0001111		0	1	0	A	DOUT	ADD	B	-	-	0	0	0
	(Dir)	0010000		0	0	0	A	B	ADD	LIT	ALU	-	1	0	0
SUB	A,B	0010001		0	1	0	A	B	SUB	-	-	-	0	0	0
	B,A	0010010		0	0	1	A	B	SUB	-	-	-	0	0	0
	A,Lit	0010011		0	1	0	A	LIT	SUB	-	-	-	0	0	0
	A,(Dir)	0010100		0	1	0	A	DOUT	SUB	LIT	-	-	0	0	0
	A,(B)	0010101		0	1	0	A	DOUT	SUB	B	-	-	0	0	0
	(Dir)	0010110		0	0	0	A	B	SUB	LIT	ALU	-	1	0	0
AND	A,B	0010111		0	1	0	A	B	AND	-	-	-	0	0	0
	B,A	0011000		0	0	1	A	B	AND	-	-	-	0	0	0
	A,Lit	0011001		0	1	0	A	LIT	AND	-	-	-	0	0	0
	A,(Dir)	0011010		0	1	0	A	DOUT	AND	LIT	-	-	0	0	0
	A,(B)	0011011		0	1	0	A	DOUT	AND	B	-	-	0	0	0
	(Dir)	0011100		0	0	0	A	B	AND	LIT	ALU	-	1	0	0
OR	A,B	0011101		0	1	0	A	B	OR	-	-	-	0	0	0
	B,A	0011110		0	0	1	A	B	OR	-	-	-	0	0	0
	A,Lit	0011111		0	1	0	A	LIT	OR	-	-	-	0	0	0
	A,(Dir)	0100000		0	1	0	A	DOUT	OR	LIT	-	-	0	0	0
	A,(B)	0100001		0	1	0	A	DOUT	OR	B	-	-	0	0	0
	(Dir)	0100010		0	0	0	A	B	OR	LIT	ALU	-	1	0	0
NOT	A,A	0100011		0	1	0	A	-	NOT	-	-	-	0	0	0
	B,A	0100100		0	0	1	A	-	NOT	-	-	-	0	0	0
	(Dir)	0100101		0	0	0	A	B	NOT	LIT	ALU	-	1	0	0
XOR	A,B	0100110		0	1	0	A	B	XOR	-	-	-	0	0	0
	B,A	0100111		0	0	1	A	B	XOR	-	-	-	0	0	0
	A,Lit	0101000		0	1	0	A	LIT	XOR	-	-	-	0	0	0
	A,(Dir)	0101001		0	1	0	A	DOUT	XOR	LIT	-	-	0	0	0
	A,(B)	0101010		0	1	0	A	DOUT	XOR	B	-	-	0	0	0
	(Dir)	0101011		0	0	0	A	B	XOR	LIT	ALU	-	1	0	0
SHL	A,A	0101100		0	1	0	A	-	SHL	-	-	-	0	0	0
	B,A	0101101		0	0	1	A	-	SHL	-	-	-	0	0	0
	(Dir)	0101110		0	0	0	A	B	SHL	LIT	ALU	-	1	0	0
SHR	A,A	0101111		0	1	0	A	-	SHR	-	-	-	0	0	0
	B,A	0110000		0	0	1	A	-	SHR	-	-	-	0	0	0
	(Dir)	0110001		0	0	0	A	B	SHR	LIT	ALU	-	1	0	0
INC	B	0110010		0	0	1	ONE	B	ADD	-	-	-	0	0	0
	(B)	0110011		0	0	1	ONE	DOUT	ADD	B	ALU	-	1	0	0
	(Dir)	0110100		0	0	1	ONE	DOUT	ADD	LIT	ALU	-	1	0	0

CMP	A,B	0110101		0	0	0	A	B	SUB	-	-	-	0	0	0
	A,Lit	0110110		0	0	0	A	LIT	SUB	-	-	-	0	0	0
	A,(Dir)	0110111		0	0	0	A	DOUT	SUB	LIT	-	-	0	0	0
	A,(B)	0111000		0	0	0	A	DOUT	SUB	B	-	-	0	0	0
JMP	Dir	0111001		1	0	0	-	-	-	-	-	LIT	0	0	0
JEQ	Dir	0111010	Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JNE	Dir	0111011	Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JGT	Dir	0111100	N=0 y Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLT	Dir	0111101	N=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JGE	Dir	0111110	N=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLE	Dir	0111111	N=1 o Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JCR	Dir	1000000	C=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JOV	Dir	1000001	V=1	1	0	0	-	-	-	-	-	LIT	0	0	0
CALL	Dir	1000010		1	0	0	-	-	-	SP	PC	LIT	1	0	1
RET		1000011		0	0	0	-	-	-	-	-	-	0	1	0
		1000100		1	0	0	-	-	-	SP	-	DOUT	0	0	0
PUSH	A	1000101		0	0	0	A	ZERO	ADD	SP	ALU	-	1	0	1
	B	1000110		0	0	0	ZERO	B	ADD	SP	ALU	-	1	0	1
POP	A	1000011		0	1	0	-	-	-	-	-	-	0	1	0
		1000111		0	1	0	ZERO	DOUT	ADD	SP	-	-	0	0	0
POP	B	1000011		0	0	0	-	-	-	-	-	-	0	1	0
		1001000		0	0	1	ZERO	DOUT	ADD	SP	-	-	0	0	0
NOP		1001001		0	0	0	-	-	-	-	-	-	0	0	0

Computador básico - Otros ejemplos de Assembly

```
DATA:
  temp 0
CODE:
  MOV A,3
  MOV B,5
  CALL swap_3
  JMP end
  swap_1:          ;Intercambio con uso de memoria.
    MOV (temp),A   ;Mem[temp] = A
    MOV A,B        ;A = B
    MOV B,(temp)   ;B = Mem[temp] = A
    RET
  swap_2:          ;Intercambio con uso de stack
    PUSH A         ;Mem[SP] = A
    MOV A,B        ;A = B
    POP B          ;B = Mem[SP] = A (2 ciclos)
    RET
  swap_3:          ;Intercambio con XOR
    XOR A,B        ;A = A XOR B
    XOR B,A        ;B = (A XOR B) XOR B = A XOR (B XOR B) = A XOR 0 = A
    XOR A,B        ;A = (A XOR B) XOR A = B XOR (A XOR A) = B XOR 0 = B
    RET
  swap_4:          ;Intercambio con ADD y SUB
    ADD A,B        ;A = A + B
    SUB B,A        ;B = (A + B) - B = A
    SUB A,B        ;A = (A + B) - A = B
    RET
  end:
    JMP end
```

Un ejemplo útil es el **intercambio de registros**. Este se puede realizar de varias maneras, como se ilustra a continuación. Basta con que cambiemos el *label* de la instrucción **CALL** para cambiar el método, pero todos hacen lo mismo.

Computador básico - Otros ejemplos de Assembly

```
DATA:
  a 7
  b 9
  r 0
CODE:
  loop:
    MOV A,(a)
    CMP A,0
    JEQ end                ;A == 0 -> PC = end
    AND A,1
    CMP A,0                ;A % 2 == 0 -> PC = avoid_addition
    JEQ avoid_addition     ;No se suma si el número es par
    MOV A,(r)
    MOV B,(b)
    ADD (r)                ;r += b
    avoid_addition:
      MOV A,(a)
      SHR (a)              ;a = a // 2 (división entera por 2)
      MOV A,(b)
      SHL (b)              ;b = b * 2
      JMP loop
    end:
      JMP end
```

Otro ejemplo interesante es la **multiplicación rusa**. Esta se basa en realizarla solo a partir de *shifts left/right* sobre los operandos (*i.e.* dividir y multiplicar por dos). Pueden ver en los siguientes enlaces cómo funciona y por qué funciona.

Computador básico - Otros ejemplos de Assembly

```
CODE:
  MOV A,3
  MOV B,7
  PUSH A        ;Mem[255] = A = 3
  PUSH B        ;Mem[254] = B = 7
  MOV A,0
  NOT A,A       ;A = 11111111
  SUB A,1       ;A = 11111110
  MOV B,A       ;B = 11111110
  MOV A,(B)     ;A = Mem[11111110] = Mem[254] = 7
  INC B         ;B = 11111111
  MOV B,(B)     ;B = Mem[11111111] = Mem[255] = 3
```

Podemos acceder directamente a la memoria de *stack* a través de direccionamiento indirecto. Esto es útil al momento de querer corroborar que nuestras instrucciones de *stack* y subrutinas escriben en memoria correctamente.

Computador básico - Otros ejemplos de Assembly

```
DATA:
  sum 0
CODE:
  MOV A,3
  MOV B,7
  PUSH A        ;Mem[255] = A = 3
  PUSH B        ;Mem[254] = B = 7
  CALL func     ;Mem[253] = Ret. Dir = PC+1 = 5
  POP A         ;No se ejecuta por cambio de retorno.
  POP B         ;No se ejecuta por cambio de retorno.
  JMP end       ;Se ejecuta después de RET.
  func:
    ADD (sum)
    POP A       ;A = Mem[253] = 5
    RET         ;PC = Mem[254] = 7
  end:
```

Por último, podemos modificar la dirección de retorno de una subrutina. En el ejemplo, el **POP** de `func` hace que se pierda el retorno y PC carga el valor ingresado de *B* que, en este caso, hace que se ejecute **JMP** end y finalice el programa antes.

Errores comunes

A continuación, veremos los errores más comunes que cometen en las interrogaciones de forma que puedan evitarlos.

Errores comunes

Considerar un *label* de código como una instrucción

- Los *labels* solo son etiquetas que se asocian a la dirección de memoria de la primera instrucción bajo ellas. **No ocupan una dirección de memoria**.

```
MOV A,1         ; Dirección de memoria 0
MOV B,1         ; Dirección de memoria 1
loop:           ; Dirección de memoria 2
  ADD A,B       ; Dirección de memoria 3
  JMP loop      ; Dirección de memoria 4
```

INCORRECTO

El *label* no ocupa una dirección de memoria.

```
MOV A,1         ; Dirección de memoria 0
MOV B,1         ; Dirección de memoria 1
loop:           ; loop = literal 2
  ADD A,B       ; Dirección de memoria 2
  JMP loop      ; Dirección de memoria 3
```

CORRECTO

Errores comunes

Saltos condicionales consecutivos sin `CMP`

- Las *flags* de *Status* se actualizan en cada ciclo **independiente de la instrucción que se ejecute**. Las instrucciones de salto **también las modifican**, por lo que una condición evaluada **no se retiene**.

- Por este motivo, si se desean realizar saltos condicionales consecutivos, **siempre se debe realizar el `CMP` antes**, aunque sea uno ya realizado.

Errores comunes

Saltos condicionales consecutivos sin CMP

```
MOV A,1
MOV B,3
CMP A,B
JEQ case1      ; JMP case1 if A == B
JGT case2      ; JMP case2 if A > B
JMP case3      ; else JMP case3
```

INCORRECTO

La ejecución de JGT no usará las *flags* computadas por el CMP, sino las que se computan con JEQ. Aquí es importante recordar que, aunque no se ocupe su valor de salida, la ALU igual computará *algo*.

```
MOV A,1
MOV B,3
CMP A,B
JEQ case1      ; JMP case1 if A == B
CMP A,B
JGT case2      ; JMP case2 if A > B
JMP case3      ; else JMP case3
```

CORRECTO

Tanto JEQ como JGT utilizan las mismas *flags*, ya que se alimentan del resultado del mismo CMP.

Errores comunes

Cantidad de ciclos de POP y RET

- Es necesario conocer las direcciones de memoria que utiliza cada instrucción. Un error en estos casos es **no tomar en cuenta que POP y RET ocupan dos direcciones de memoria**.

```
CALL func       ; Dirección de memoria 0
JMP end         ; Dirección de memoria 1
func:
  ADD A,B       ; Dirección de memoria 2
  RET           ; Dirección de memoria 3
end:
  MOV B,A       ; Dirección de memoria 4
```

INCORRECTO

RET ocupa dos direcciones de memoria.

```
CALL func       ; Dirección de memoria 0
JMP end         ; Dirección de memoria 1
func:
  ADD A,B       ; Dirección de memoria 2
  RET           ; Dirección de memoria 3-4
end:
  MOV B,A       ; Dirección de memoria 5
```

CORRECTO

Ejercicios

Ahora, veremos algunos ejercicios.

Estos se basan en preguntas de tareas y pruebas de semestres anteriores, por lo que nos servirán de preparación para las evaluaciones.

Ejercicios

Si se elimina la instrucción `CMP` del computador básico, ¿cómo deben modificarse las instrucciones de salto, sin alterar el *hardware*, para que estas no dependan del resultado de la última instrucción ejecutada? Escriba detalladamente todas las modificaciones necesarias y sus implicancias. Asuma que solo es necesario resolver el caso de la comparación de los registros *A* y *B* y que no es posible sobreescribir los registros para realizar la comparación.

***Interrogación* 1, 2017-1**

Ejercicios

Modifique la arquitectura del computador básico para que el registro `Status` se actualice solo después de la ejecución de una instrucción `CMP`.

***Examen*, 2017-1**

Ejercicios

Modifique el *hardware* del computador básico para que las instrucciones `RET` y `POP` tomen un solo ciclo.

***Interrogación* 1, 2018-1**

Ejercicios

Modifique la arquitectura del computador básico para implementar las instrucciones `NOT B,B`; `SHL B,B` y `SHR B,B`; es decir, que realice las operaciones de un operando sobre el registro *B* y almacene el resultado en él.

***Examen*, 2023-1**

Ejercicios

Modifique la arquitectura del computador básico para implementar las instrucciones `MOV A,(B+offset)` y `MOV (B+offset),A`, *i.e.* instrucciones de direccionamiento indirecto con registro *B* y *offset*, siendo este último un literal que puede ser positivo o negativo.

***Examen*, 2023-1**

Ejercicios

Asuma que se agregan las instrucciones anteriores a la ISA del computador básico y se ejecuta el programa adjunto. Señale solo los valores finales de los registros *A*, *B* y SP. Asuma que todos los registros son de 8 bits y que SP se interpreta como un número positivo.

```
MOV A,2           ; Dirección Mem. Instr.: 0x00
MOV B,5           ; Dirección Mem. Instr.: 0x01
PUSH A            ; Dirección Mem. Instr.: 0x02
PUSH B            ; Dirección Mem. Instr.: 0x03
CALL mystery      ; Dirección Mem. Instr.: 0x04
POP B             ; Dirección Mem. Instr.: 0x05-0x06
POP A             ; Dirección Mem. Instr.: 0x07-0x08
JMP end           ; Dirección Mem. Instr.: 0x09
mystery:
  MOV B,0         ; Dirección Mem. Instr.: 0x0A
  NOT B,B         ; Dirección Mem. Instr.: 0x0B
  MOV A,(B + -2)  ; Dirección Mem. Instr.: 0x0C
  ADD A,4         ; Dirección Mem. Instr.: 0x0D
  MOV (B + -2),A  ; Dirección Mem. Instr.: 0x0E
  RET             ; Dirección Mem. Instr.: 0x0F-0x10
end:
```

Antes de terminar

¿Dudas?

¿Consultas?

¿Inquietudes?

¿Comentarios?

DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 5 - Saltos y Subrutinas

Profesor: Germán Leandro Contreras Sagredo

Anexo - Resolución de ejercicios

¡Importante!

Estos ejercicios pueden tener más de un desarrollo correcto. Las respuestas a continuación no son más que soluciones que **no excluyen** otras alternativas igual de correctas.

Actividad en clase - Respuesta

Elabore, en Assembly, un programa que encuentre la mediana de un arreglo ordenado de números.

- Si existe una cantidad impar de elementos, obtenemos el índice de la mitad del arreglo.
- Si existe una cantidad par de elementos, sumamos los números centrales y realizamos la división entera por 2.

```
DATA:
  len  6
  median 0
  arr -5
       0
       6
       7
       11
       20
CODE:
  MOV A,(len)
  AND A,1              ; Si A es par, AND A,1 == 0. Si es impar, AND A,1 == 1
  CMP A,0
  JEQ even_arr
  odd_arr:
    MOV A,(len)
    SHR A,A            ; Índice mitad del arreglo = len // 2
    MOV B,arr          ; B = dirección arr
    ADD B,A            ; B = dirección arr + len // 2
    MOV A,(B)          ; A = mediana del arreglo
    JMP end
  even_arr:
    MOV A,(len)
    SHR A,A            ; Primer elemento de la mediana = len // 2 - 1
    SUB A,1
    MOV B,arr          ; B = dirección arr
    ADD B,A            ; B = dirección arr + len // 2 - 1
    MOV A,(B)          ; A = primer elemento de la mediana
    INC B              ; Segundo elemento de la mediana = len // 2
    ADD A,(B)          ; A = (primer elemento + segundo elemento) // 2
    SHR A,A
  end:
    MOV (median),A     ; Escribimos en la variable "median" el resultado
```

Ejercicios - Respuesta

Si se elimina la instrucción `CMP` del computador básico, ¿cómo deben modificarse las instrucciones de salto, sin alterar el *hardware*, para que estas no dependan del resultado de la última instrucción ejecutada? Escriba detalladamente todas las modificaciones necesarias y sus implicancias. Asuma que solo es necesario resolver el caso de la comparación de los registros *A* y *B* y que no es posible sobreescribir los registros para realizar la comparación.

Respuesta en la siguiente diapositiva.

Ejercicios - Respuesta

Como no queremos que los saltos dependan de la instrucción anterior (esto es, que no se basen en cómo haya quedado el registro `STATUS` después de ejecutarlas), lo que se necesita hacer es añadir más operaciones a las instrucciones de salto. En este caso, lo que se debe hacer es replicar la instrucción `CMP` dentro de la ejecución que realizan los saltos. De esta forma, cada instrucción de salto contendrá dos *opcodes*:

1) El primero corresponderá al mismo que solía tener `CMP`: Se realiza la resta entre los registros A y B sin guardar el resultado para poder actualizar las flags del registro `STATUS`.
2) El segundo será el que tenía cada salto originalmente.

De esta forma, el cambio sustancial que se genera es que las instrucciones de salto toman **dos ciclos** en vez de uno.

Ejercicios - Respuesta

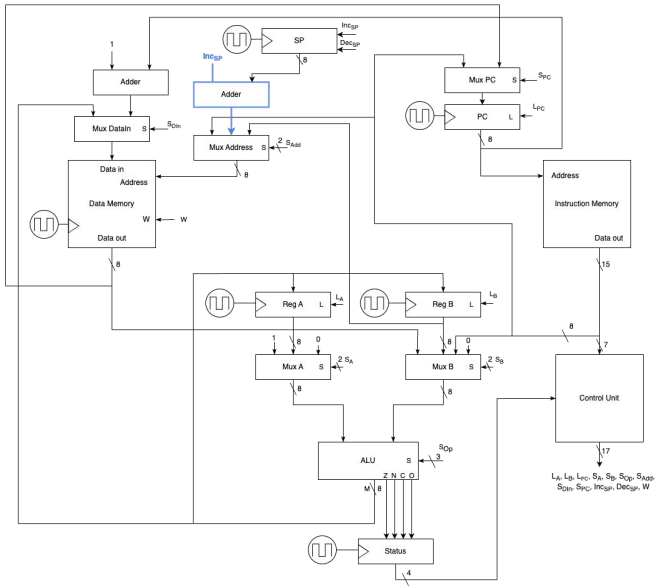
Modifique la arquitectura del computador básico para que el registro `Status` se actualice solo después de la ejecución de una instrucción `CMP`.

Basta con crear una nueva señal L_{Stat} (señal de carga del registro *Status*) que se active si, y solo si el *opcode* de la instrucción corresponde a `CMP`. En otro caso, la unidad de control debe encargarse de transmitir $L_{\text{Stat}} = 0$.

Ejercicios

Modifique el *hardware* del computador básico para que las instrucciones `RET` y `POP` tomen un solo ciclo.

Basta con conectar `SP` a un sumador que recibe, además, la señal Inc_{SP}. Solo cuando activamos esta señal es cuando necesitamos acceder a la siguiente dirección de la memoria *stack*, por lo que la dirección se modifica directamente y no requerimos esperar un ciclo para el ajuste del puntero.



Ejercicios - Respuesta

Modifique la arquitectura del computador básico para implementar las instrucciones `NOT B,B`; `SHL B,B` y `SHR B,B`; es decir, que realice las operaciones de un operando sobre el registro *B* y almacene el resultado en él.

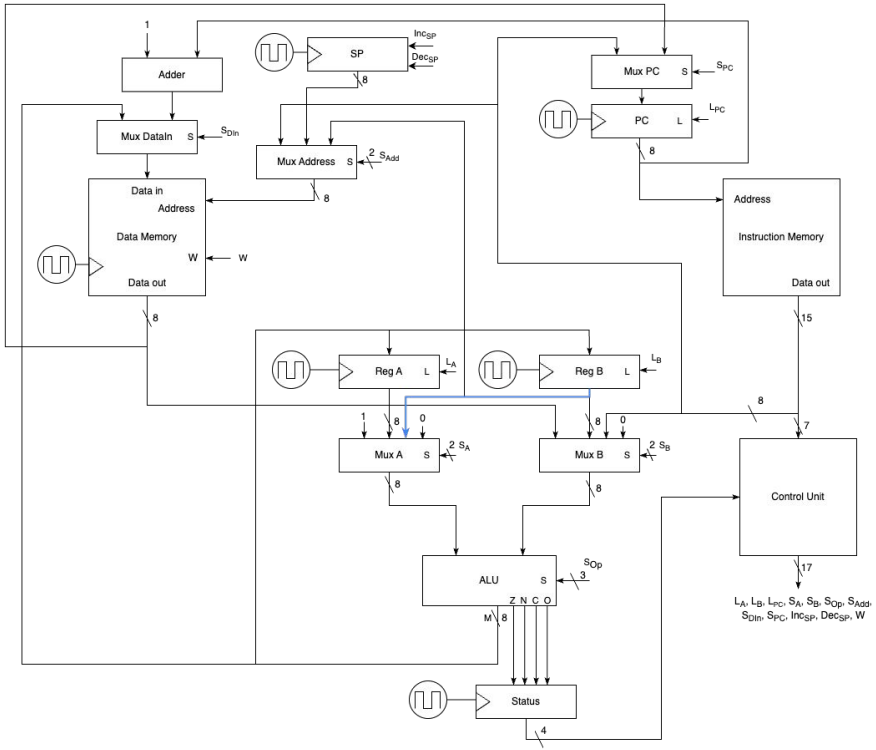
Respuesta en la siguiente diapositiva.

Ejercicios - Respuesta

Solución: Para esta implementación, se conecta el registro B con el componente `Mux A` para utilizar el valor de B como *input* de las operaciones `NOT`, `SHL` y `SHR` de la ALU:

De esta forma, sin modificar ni agregar las señales de la microarquitectura, se tiene la siguiente tabla para la ejecución de las instrucciones implementadas:

Instrucción	L_A	L_B	L_{PC}	W	Inc_{SP}	Dec_{SP}	S_A	S_B	S_{OP}	S_{Add}	S_{DIn}	S_{PC}
NOT B,B	0	1	0	0	0	0	B	-	NOT	-	-	-
SHL B,B	0	1	0	0	0	0	B	-	SHL	-	-	-
SHR B,B	0	1	0	0	0	0	B	-	SHR	-	-	-



Ejercicios - Respuesta

Modifique la arquitectura del computador básico para implementar las instrucciones `MOV A,(B+offset)` y `MOV (B+offset),A`, *i.e.* instrucciones de direccionamiento indirecto con registro *B* y *offset*, siendo este último un literal que puede ser positivo o negativo.

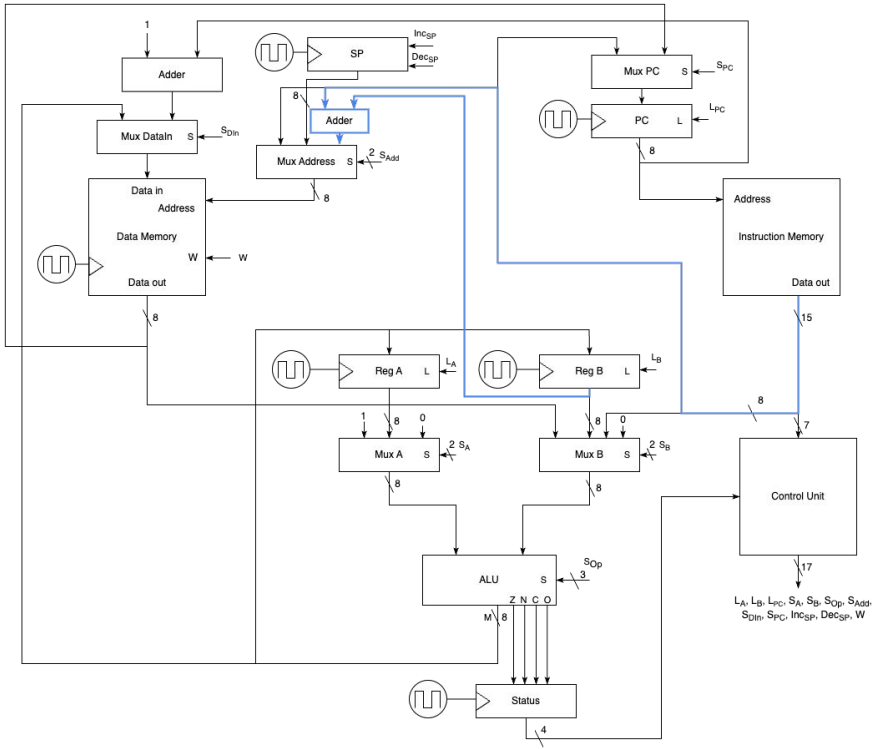
Respuesta en la siguiente diapositiva.

Ejercicios - Respuesta

Solución: Para la implementación de estas instrucciones, se puede modificar la conexión entre el registro B y el componente `Mux Address` para que ahora reciba el resultado de un sumador que sume el valor de B y el literal de la instrucción:

De esta forma, no es necesario agregar ni modificar señales, pero sí asegurar que el compilador haga uso del literal 0 para las instrucciones `MOV A,(B)` y `MOV (B),A`. La tabla de señales es como sigue para las instrucciones implementadas:

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,(B+offset)`	1	0	0	0	0	0	0	DOUT	ADD	B+offset	-	-
`MOV (B+offset),A`	0	0	0	0	0	0	A	0	ADD	B+offset	ALU	-



Ejercicios - Respuesta

Asuma que se agregan las instrucciones anteriores a la ISA del computador básico y se ejecuta el programa adjunto. Señale solo los valores finales de los registros *A*, *B* y SP. Asuma que todos los registros son de 8 bits y que SP se interpreta como un número positivo.

```
MOV A,2           ; Dirección Mem. Instr.: 0x00
MOV B,5           ; Dirección Mem. Instr.: 0x01
PUSH A            ; Dirección Mem. Instr.: 0x02
PUSH B            ; Dirección Mem. Instr.: 0x03
CALL mystery      ; Dirección Mem. Instr.: 0x04
POP B             ; Dirección Mem. Instr.: 0x05-0x06
POP A             ; Dirección Mem. Instr.: 0x07-0x08
JMP end           ; Dirección Mem. Instr.: 0x09
mystery:
  MOV B,0         ; Dirección Mem. Instr.: 0x0A
  NOT B,B         ; Dirección Mem. Instr.: 0x0B
  MOV A,(B + -2)  ; Dirección Mem. Instr.: 0x0C
  ADD A,4         ; Dirección Mem. Instr.: 0x0D
  MOV (B + -2),A  ; Dirección Mem. Instr.: 0x0E
  RET             ; Dirección Mem. Instr.: 0x0F-0x10
end:
```

Respuesta en la siguiente diapositiva.

Ejercicios - Respuesta

Solución: Se señalan los puntos clave en donde se modifican los registros y por qué:

1. Antes de `CALL mystery`, se tiene `A = 2`, `B = 5` por las instrucciones `MOV` y `SP = 253` por las instrucciones `PUSH`.

2. Posterior a `CALL mystery`, se tiene `A = 2`, `B = 5` y `SP = 252` por el almacenamiento de la dirección de retorno 0x05 (`PC+1`).

3. Antes de `RET`, se tiene `A = 9`, `B = 255` y `SP = 252`. En este punto, lo que se hizo fue modificar la dirección de retorno de la subrutina `mystery`, apuntando ahora a la instrucción `JMP end`.

4. Posterior a `RET`, el programa termina sin ejecutar las instrucciones `POP` y se tiene `A = 9`, `B = 255` y `SP = 253`.