DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 5 - Saltos y Subrutinas

Profesor: Germán Leandro Contreras Sagredo

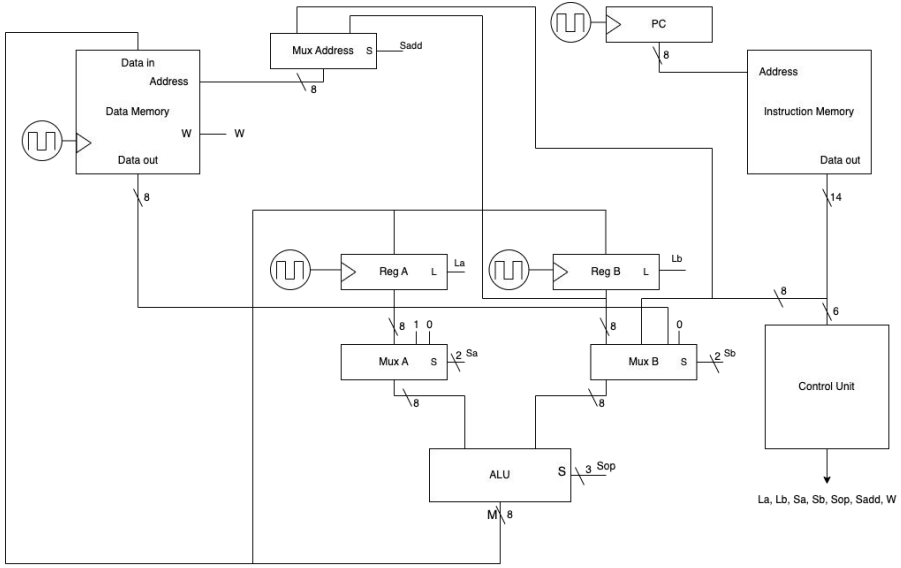
Objetivos de la clase

- 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.

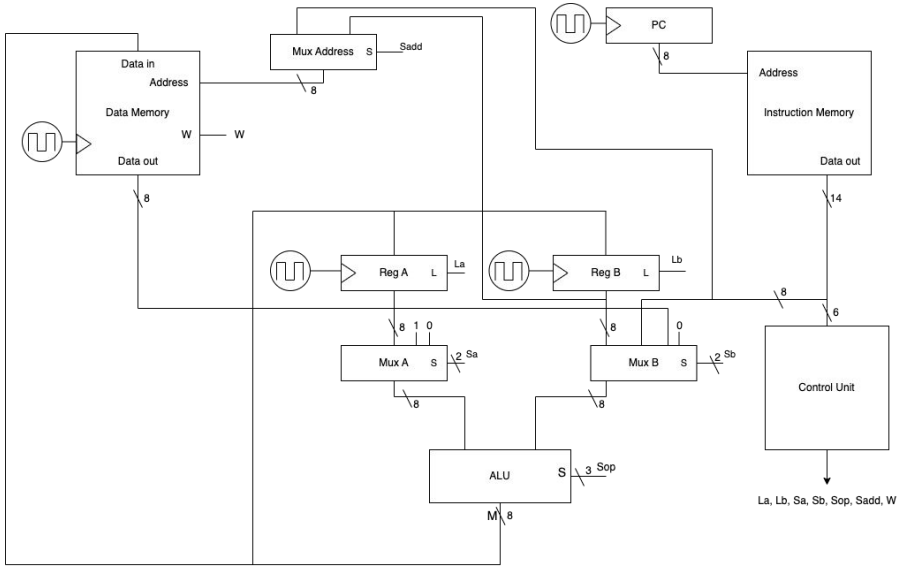


Hasta ahora...

No obstante lo anterior:

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

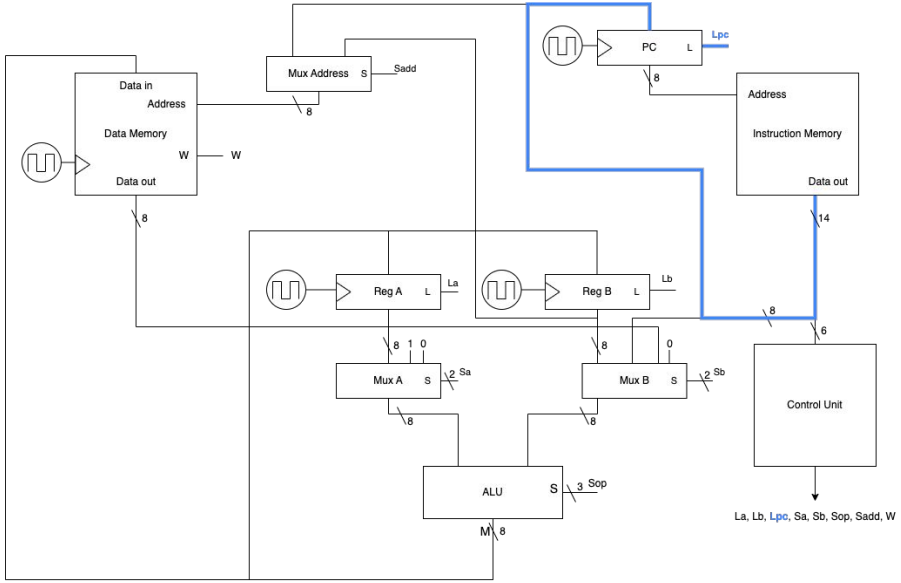
Al otorgar esta capacidad a nuestra máquina, esta será **realmente** programable.



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.

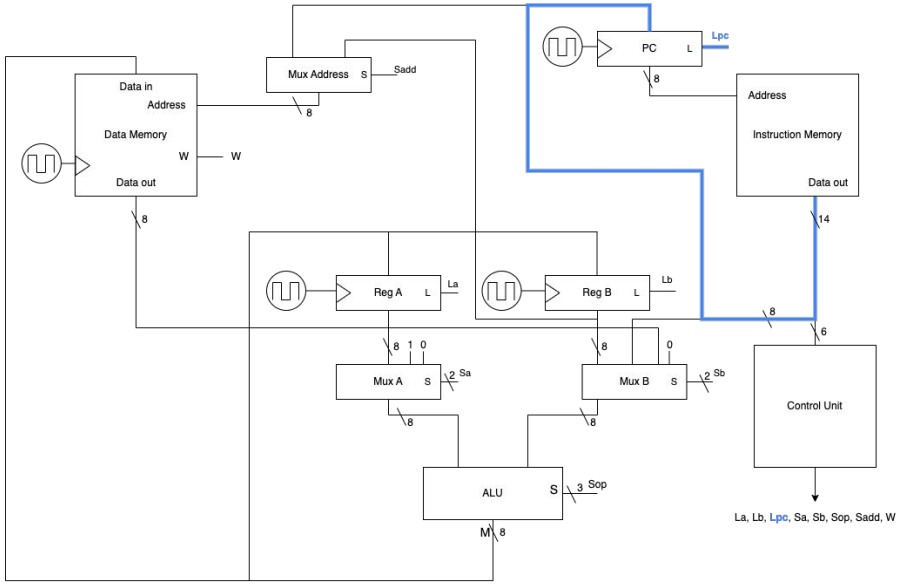
El componente que debemos modificar para este fin es el *Program Counter*.



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**.



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
```

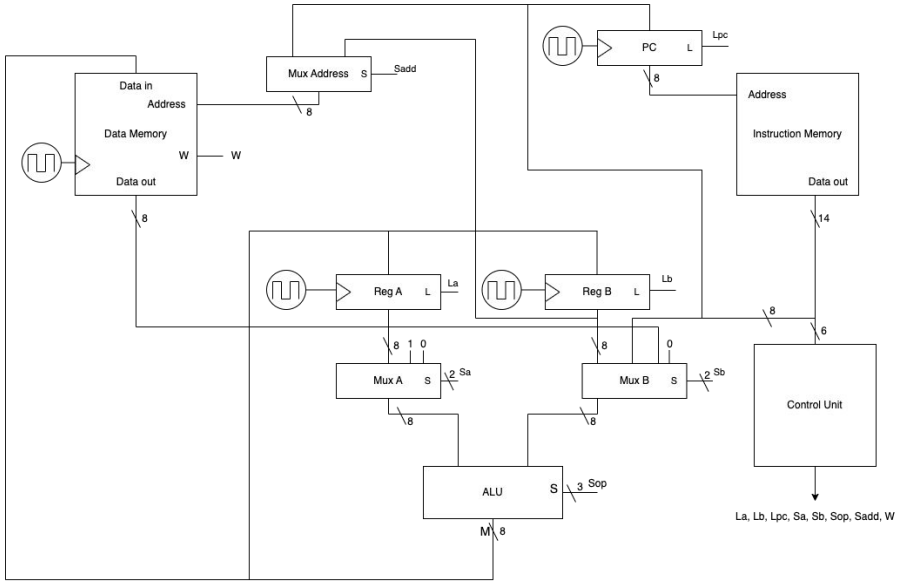
Estos son **saltos condicionales** y veremos cómo incluirlos en nuestros programas a través de modificaciones en el *hardware*.

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`

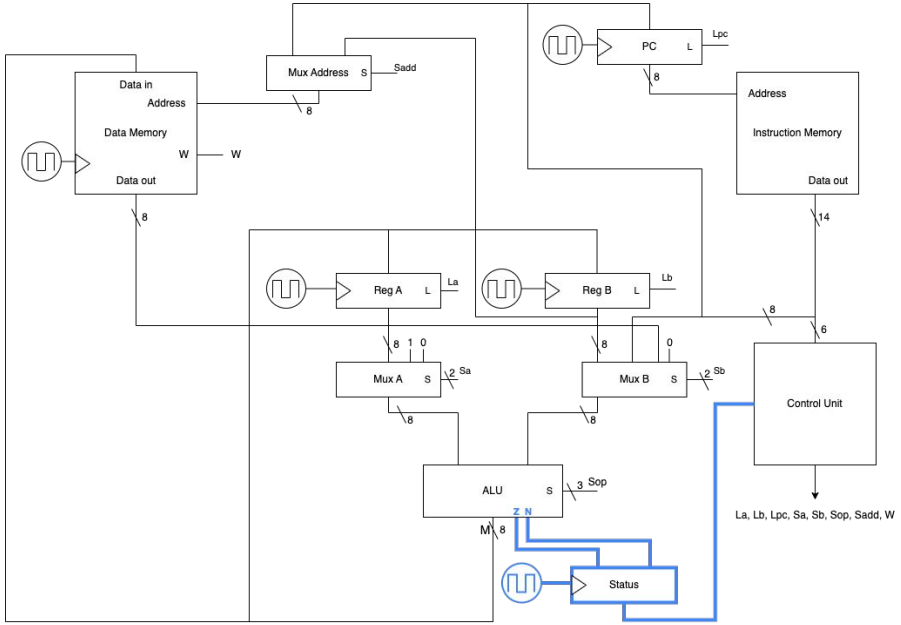
La **ALU** nos puede ayudar a evaluar condiciones **a partir del resultado de su operación**.



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*: *Zero***. Indica si el resultado de la operación fue cero o no.
- ***N*: *Negative***. Indica si el resultado de la operación fue negativo o no.

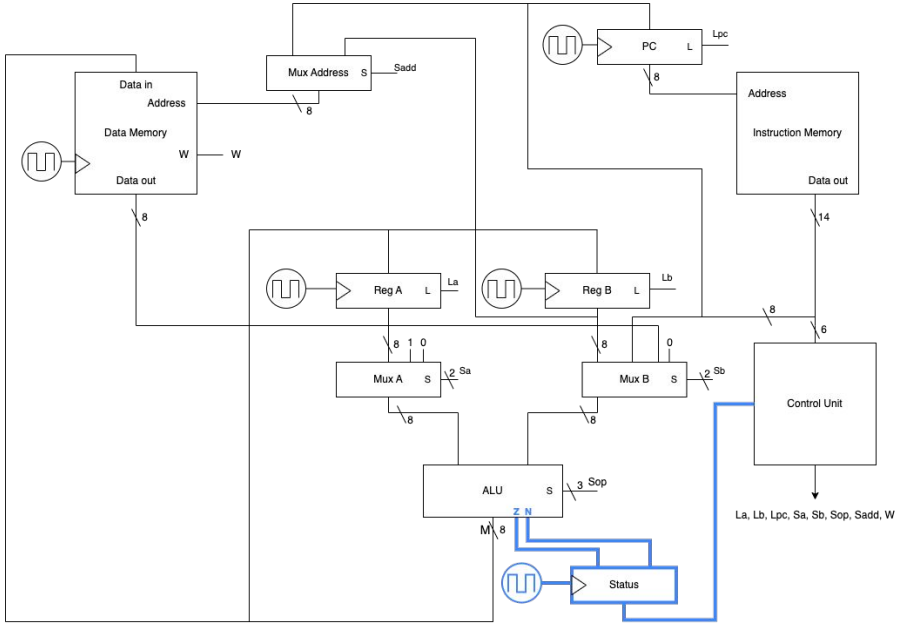


Asumimos que la ALU, en su circuito interno, se encarga de la obtención de estas señales. Por ejemplo, la *flag N* se puede deducir del bit más significativo del número resultante, mientras que la *flag Z* se puede obtener a partir de un circuito que verifique todos los bits sean iguales a cero (compuertas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

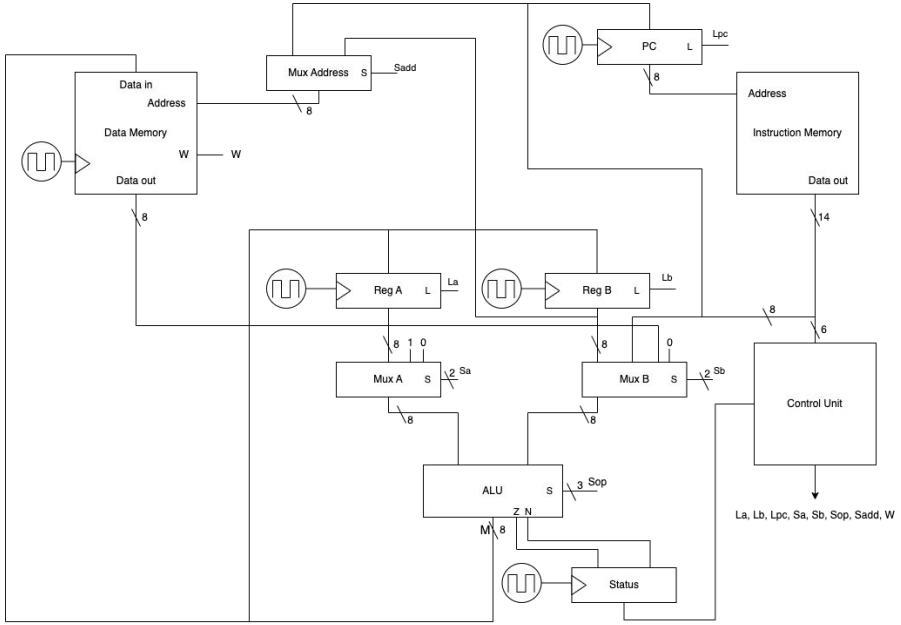


Si no se cumple la condición, la unidad de control propaga la señal $L_{PC} = 0$ para asegurar que se ejecute la siguiente instrucción y no la indicada por el literal. Si se cumple, se propaga la señal $L_{PC} = 1$ y se habilita el salto al sobreescribir la dirección de la instrucción a ejecutar.

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.

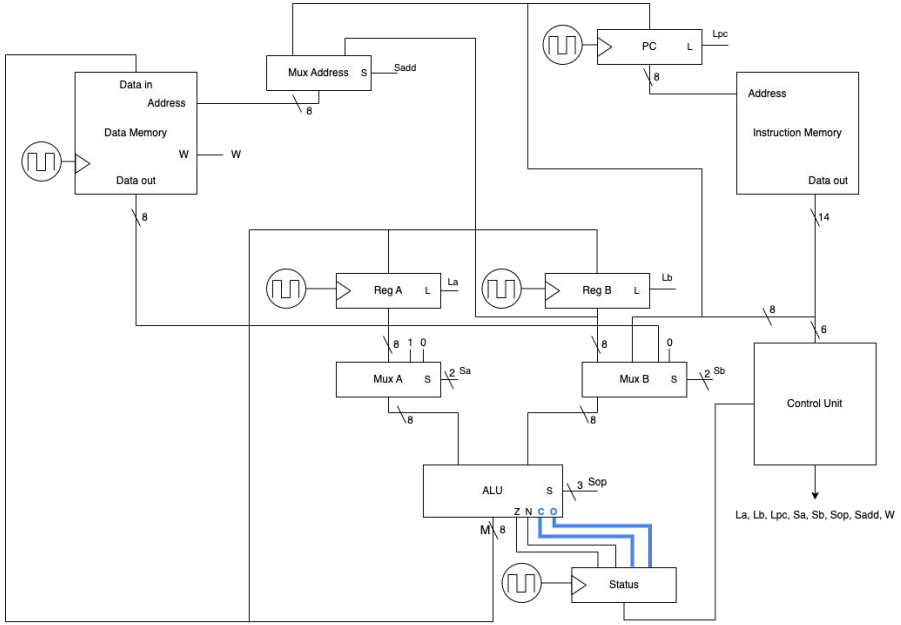
¿Qué otras condiciones podríamos incluir?



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.

Ahora, ¿qué operación utilizamos y cómo?



Computador básico - Saltos condicionales en Assembly

Definimos las instrucciones `CMP`, que tienen las siguientes señales de control:

- $L_A = L_B = 0$ (no cargamos datos en los registros).
- $S_{OP} = 001$ (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 $Z = 0$ ($A \leq B$).
- **`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* = 0 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 la capacidad de representación del computador (determinado por el *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]
    CMP A,(len)     ;A - Mem[len]
    JGE end         ;Z = 0 || N = 0 -> PC = end
    ADD A,1         ;A += 1
    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 - 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.

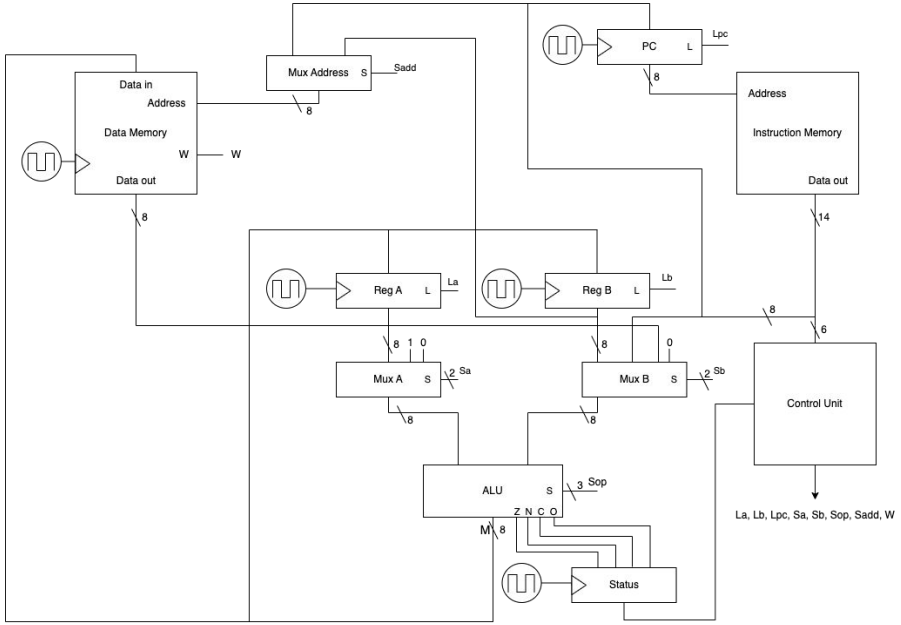
- La llamada se puede implementar como una instrucción `JMP label`, siendo `label` la dirección de memoria de la primera instrucción de la subrutina.

- 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:

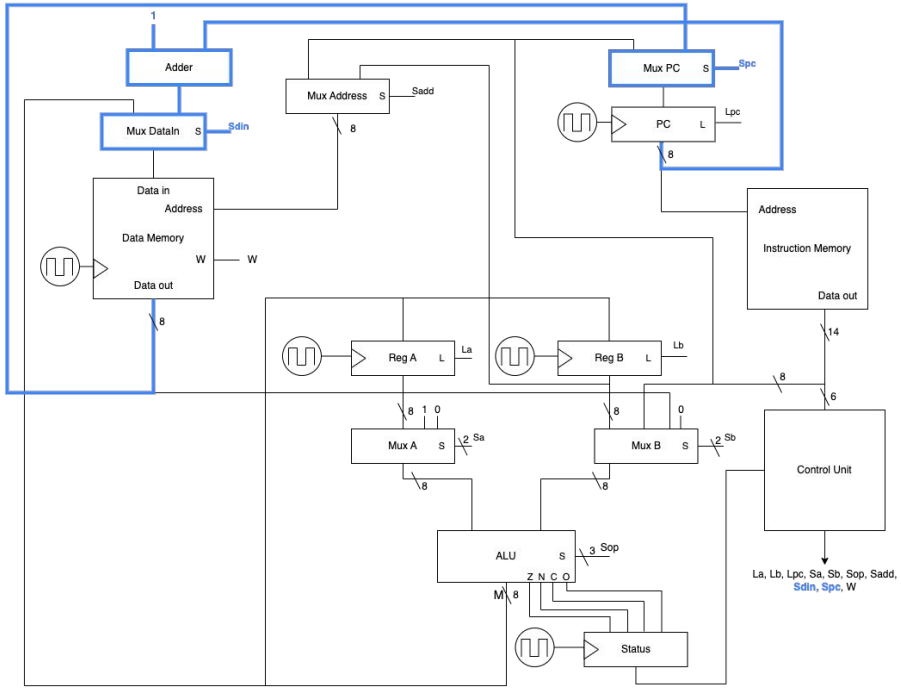
- Conexión entre la memoria de datos y *Program Counter* para cargar direcciones de instrucciones asociadas a *labels*.
- Capacidad para almacenar dirección PC+1 en memoria (dirección de retorno).



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).

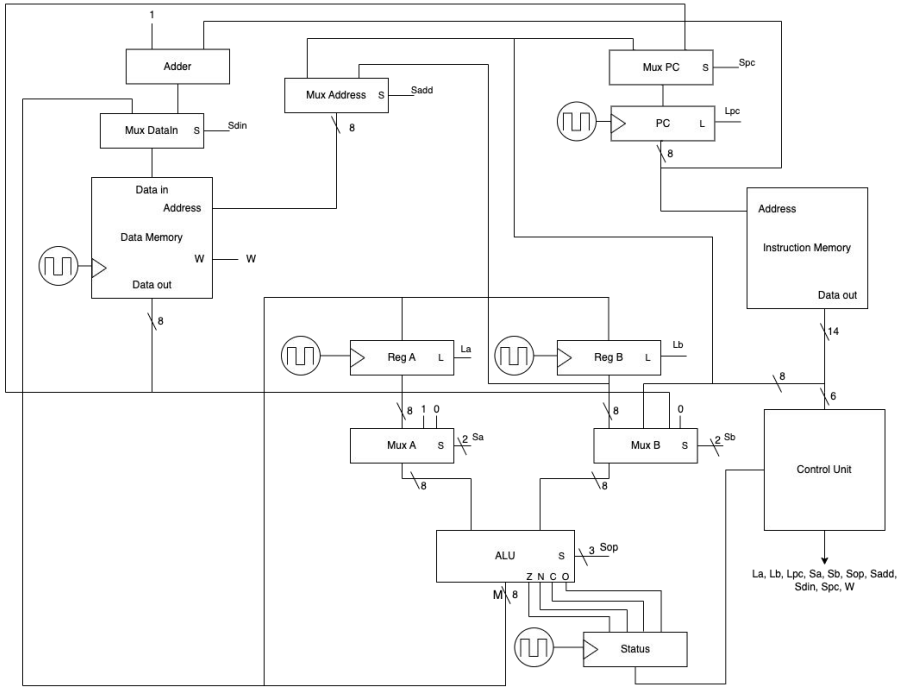
- Con el Mux PC y la señal S_{PC} ahora escogemos el dato a cargar en el *Program Counter* (literal o la dirección de la instrucción de retorno de la memoria de datos).



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?

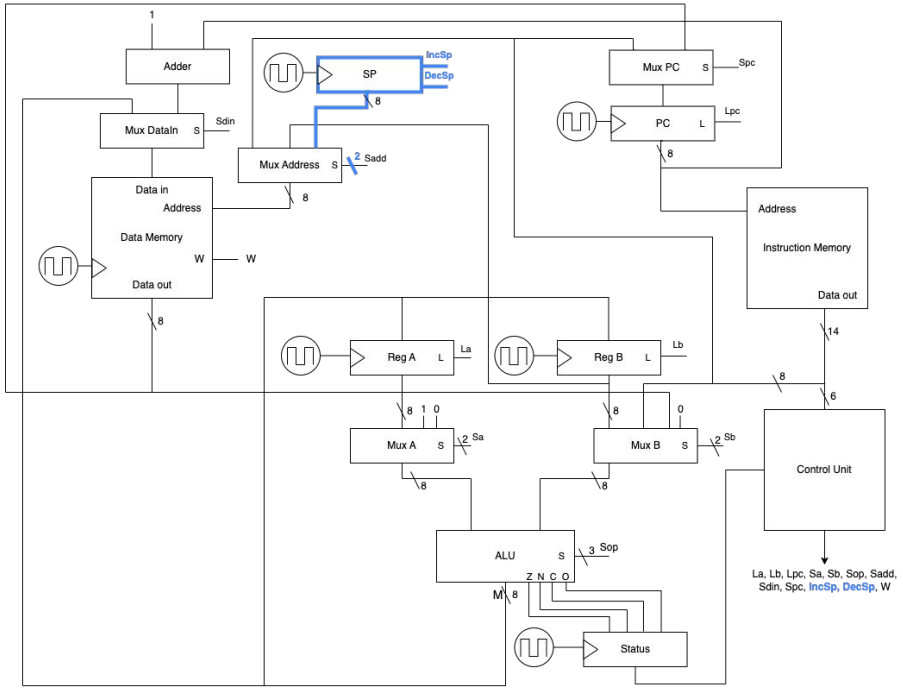
Resolvemos esto a partir de la **memoria de *stack***.



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*.

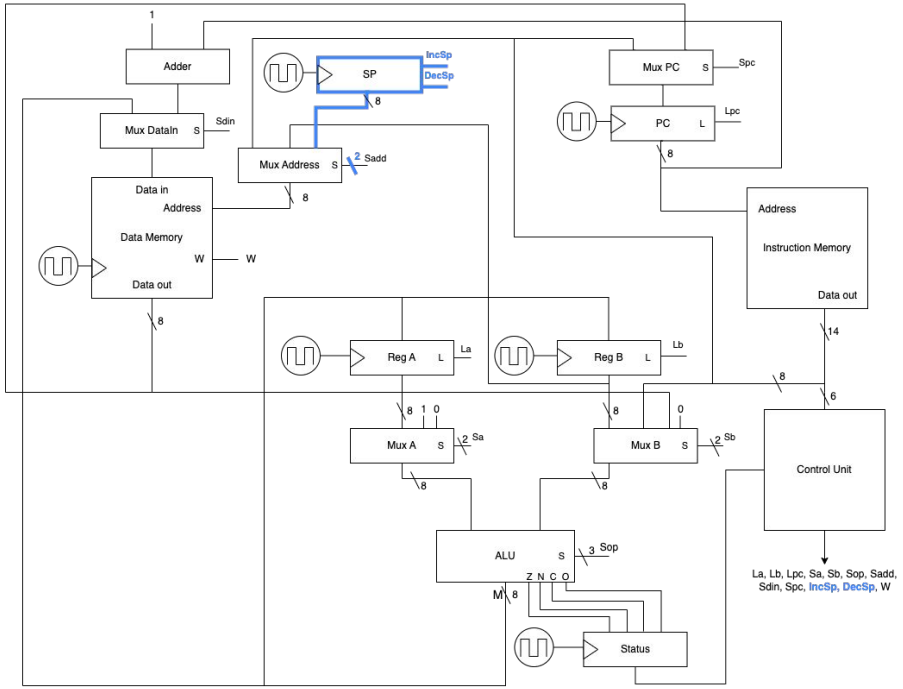
Se accede a estas direcciones desde el ***Stack Pointer*** (SP), contador cuyo valor parte con la última dirección (valor 255 para memorias con bus de 8 bits de direccionamiento).



Computador básico - Subrutinas

- La señal $\texttt{Dec}_{SP}$ decrementa en una unidad el contador SP, lo que **aumenta en una unidad el tamaño de la memoria de *stack***.

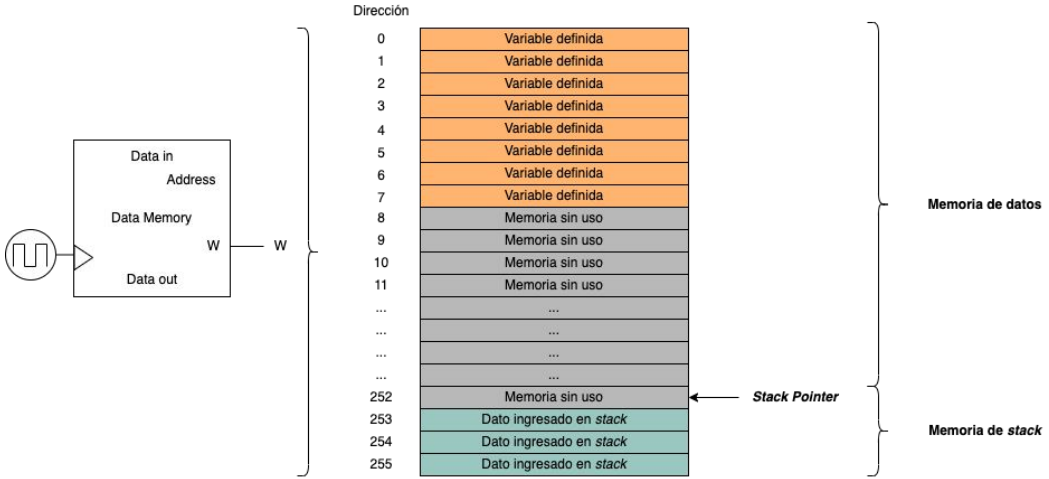
- La señal $\texttt{Inc}_{SP}$ incrementa en una unidad el contador SP, lo que **disminuye en una unidad el tamaño de la memoria de *stack***.



Como Mux Address ahora recibe 3 entradas, el bus de selección *Sadd* debe ser de dos bits.

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.



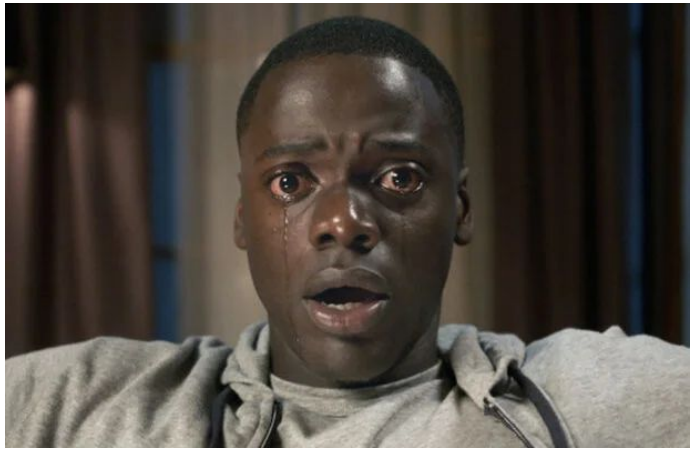
Por construcción, el *stack pointer* apuntará siempre a **una dirección mayor en una unidad** respecto al tamaño del *stack*. Esto puede ser distinto en otras arquitecturas.

Computador básico - Memoria de *stack*

¿Qué pasa si la memoria de *stack* crece lo suficiente para **chocar** con la definición de 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} = 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: $\texttt{S}_{\texttt{Add}} = \texttt{SP};\ \texttt{S}_{\texttt{PC}} = \texttt{Data Memory DOUT};\ \texttt{L}_{\texttt{PC}} = 1$
  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

```
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.

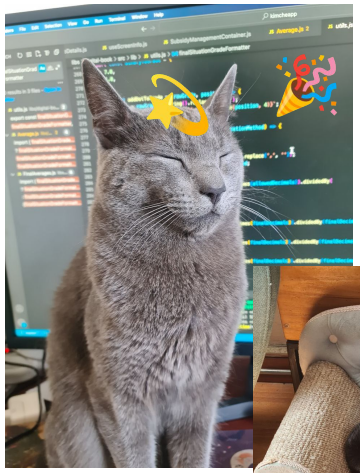
Computador básico - Subrutinas en Assembly

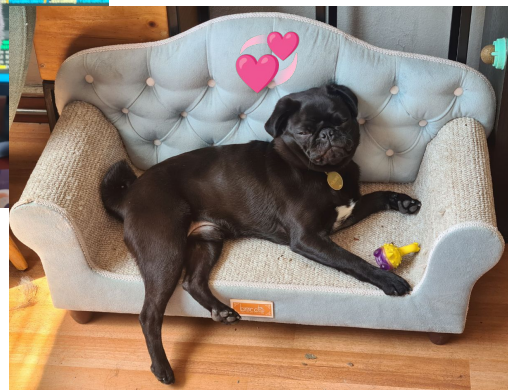
```
DATA:
  fibOne 1
  fibTwo 0
  iterations 3
  currentIteration 0
CODE:
  MOV B,(fibTwo)          ;B = Mem[fibTwo] = Elemento N - 1
  JMP loop
  next_fibonacci:
    MOV (fibTwo),A        ;Mem[fibTwo] = Elemento N
    ADD A,B               ;A = A + B = Elemento N + 1
    MOV (fibOne),A        ;Mem[fibOne] = 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
```

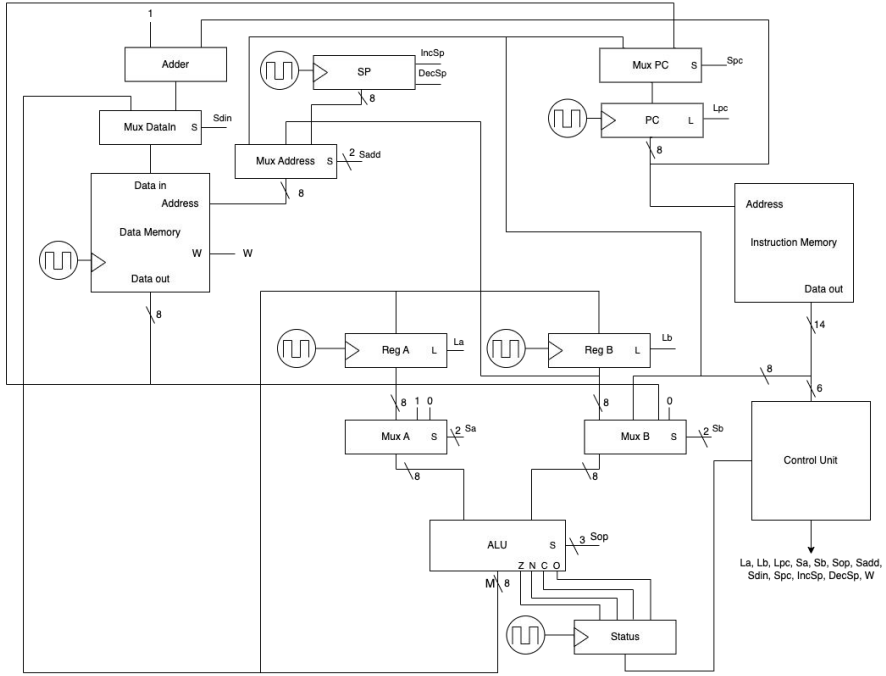
También podemos incorporar ciclos para controlar mejor la cantidad de ejecuciones.

Computador básico - Versión final

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







Computador básico - 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.

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.

Ejercicios

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

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.

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 $L_{Stat} = 0$.