Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

IIC2343 – Arquitectura de Computadores

Ayudantía 3 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

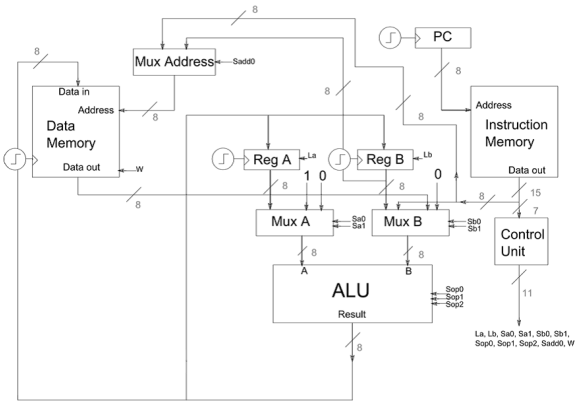
Ayudante: Germán Leandro Contreras Sagredo (glcontreras@uc.cl)

Nota al lector

El título dice "solución propuesta" por una razón bien sencilla: Estos ejercicios pueden tener más de un desarrollo correcto. Lo que se pretende hacer aquí es mostrar un camino a la solución, sin excluir la posibilidad de rutas alternativas igual de correctas.

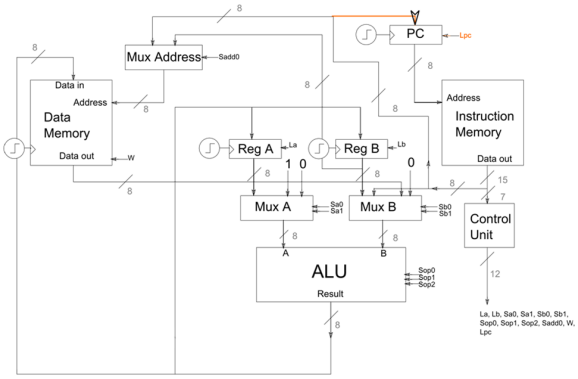
Preguntas

1. a. **(I1 - II/2017)** Considere el siguiente diagrama de bloques del computador básico, aún incompleto.



I. Explica el rol del multiplexor *Address*, da un ejemplo de su funcionamiento.
Su rol es poder seleccionar la dirección que se desea utilizar en la memoria de datos (tanto para lectura como escritura). Existen dos posibilidades, la dirección proveniente como literal de la memoria de instrucciones (direccionamiento directo) y la dirección obtenida desde el registro `B` (direccionamiento indirecto). Un ejemplo de esto es el contraste entre las instrucciones `MOV (Dir),A` y `MOV (B),A`. En el primer caso, sin pérdida de generalidad, la unidad de control le asignará el valor 0 a la señal S_{add0} para que el multiplexor `Address` seleccione el bus proveniente de la memoria de instrucciones (que será la dirección de `Dir`), mientras que en el segundo asignará el valor 1 para seleccionar el valor del registro `B`. No obstante, en ambos casos el valor escrito será el contenido en el registro `A`.

II. Explica qué es necesario agregar para permitir instrucciones de tipo salto incondicional; y explica cómo funcionaría en ese caso un salto incondicional.
Como los saltos incondicionales se realizan con la instrucción `JMP label` (siendo `label` la etiqueta de referencia de la instrucción que queremos ejecutar posteriormente), lo que se necesita es poder entregarle al registro `PC` el número de instrucción deseado y cargarlo para poder obtener el *opcode* y el literal correspondientes. Esto es posible transfiriendo la dirección de la instrucción[1] como literal desde la memoria de instrucciones[2] hacia `PC`. Como solo se transfiere una señal no es necesario un multiplexor por el momento, no obstante, sí se requiere una señal que permita decidir cuándo sobreescribir el valor del registro, que llamaremos L_{pc}. El diagrama resultante se muestra a continuación.



Entonces, su funcionamiento es el siguiente:

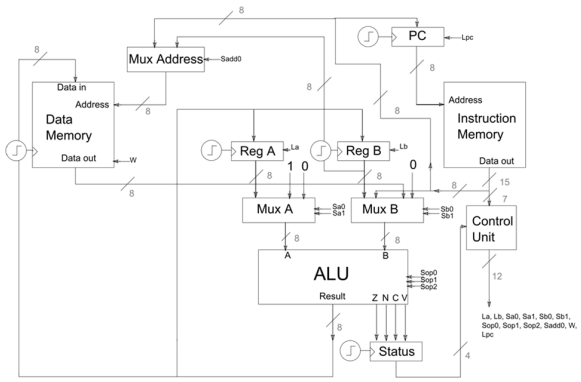
- Se obtiene el *opcode* y el literal de la instrucción `JMP label`.
- La unidad de control identifica la instrucción y habilita la escritura sobre el registro `PC`, es decir, $L_{pc} = 1$.
- El literal (correspondiente a la línea de instrucción que se desea saltar) llega al registro `PC` y es escrito en el mismo una vez que comienza el flanco de subida.

[1] En general, la dirección será equivalente al número de la línea de código del programa.
[2] En `JMP label`, el *Assembler* se encarga de convertir `label` en el literal correspondiente a la línea de la instrucción.

III. Explica qué es necesario agregar a tu respuesta en **II.** para permitir instrucciones de tipo salto condicional; y explica cómo funcionaría en ese caso un salto condicional.
La implementación de saltos condicionales implica dos tipos de instrucción esenciales: `CMP A,x` (comparación entre el valor del registro `A` y `x`) y `JEQ, JNE,` etc. (saltos según condición). En primer lugar, el objetivo de la instrucción `CMP` es poder obtener el resultado de $A - x$ sin tener que cambiar los valores de los registros, lo que en el siguiente ciclo nos ayudará a obtener información sobre la operación. Este resultado se interpreta a partir de *flags*, las que no son más que señales de un bit que nos permiten definir qué pasó con el resultado. Se suelen utilizar cuatro:

- **Z:** Indica si $A - x = 0$ o no.
- **N:** Indica si $A - x < 0$ o no.
- **C:** Indica si hubo *carry* o no.
- **O:** Indica si hubo *overflow* o no.

Por ejemplo, si queremos ejecutar la instrucción `JEQ label`, queremos ver si se activó la señal **Z** (ya que eso indicaría que la diferencia entre ambos es 0, siendo iguales). Para poder lograr esto entonces, primero se necesita añadir un registro `STATUS` que esté conectado al *clock* del computador básico y que reciba las *flags* provenientes de la `ALU`. Esto permitirá coordinar el estado del ciclo **anterior** con la unidad de control y la instrucción que esté leyendo en su momento, de forma que pueda decidir si habilitar o no la señal L_{pc} para permitir o no la escritura sobre el registro `PC`. El diagrama, entonces, se ve como sigue:



Finalmente, su funcionamiento es el siguiente:

- En el primer ciclo, se obtiene el *opcode* de la instrucción `CMP`, realizando la operación de resta en la `ALU` con el objetivo de actualizar las *flags* en `STATUS`.
- En el segundo ciclo, se obtiene el *opcode* y el literal de la instrucción de salto condicional.
- La unidad de control identifica la instrucción y habilita la escritura sobre el registro `PC` si, y solo si se cumplen las condiciones dadas por las *flags* de `STATUS`.
- Si se cumple la condición, el literal (correspondiente a la línea de instrucción a la que se desea saltar) llega al registro `PC` y es escrito en el mismo una vez que comienza el flanco de subida. En caso contrario, L_{pc} será igual a 0 y simplemente se obtendrá el *opcode* de la instrucción siguiente en el programa.

b. ¿Cuántos ciclos toma llamar una subrutina? ¿y cuántos toma retornarla? Justifique.

Para llamar a una subrutina, se siguen los siguientes pasos:

1) Guardar `PC+1` en la posición actual de `SP`.
2) Decrementar en 1 `SP`.
3) Guardar la dirección de la subrutina en `PC`.

En cambio, para retornarla, se siguen estos:

1) Incrementar en 1 `SP`.
2) Guardar el valor de memoria por `SP` incrementado en `PC`.

Si revisamos el computador básico, podemos ver que el llamado a la subrutina toma solo un ciclo, ya que:

1) `PC+1` ya se encuentra esperando en el `Mux DataIn`, espera a que la unidad de control habilite su paso a la RAM y, en el flanco de subida, se almacena en la dirección `SP`.
2) Se habilita la señal *DecSp* en el registro `SP` y, llegado el flanco de subida, decrece correspondientemente.
3) La dirección de la subrutina ya se encuentra en el `Mux PC`. Se habilita su paso a partir de *Spc*, y llegado el flanco de subida, se almacena en el registro `PC`.

Es decir, los 3 cambios se realizan simultáneamente dentro de un solo flanco de subida, por lo que un ciclo es más que suficiente.

Ahora, veamos el retorno:

1) Se habilita la señal *IncSP* en el registro `SP` y, llegado el flanco de subida, decrece correspondientemente.
2) Se guarda en el `PC` lo almacenado en `SP+1` (que sería la siguiente línea de código desde la que se saltó). Sin embargo, `SP` lo cambiamos recientemente, por lo que el valor final llega a `PC` una vez que acaba el flanco de subida y debe esperar al siguiente flanco para habilitar su escritura en el registro.

Por ende, el retorno es imposible realizarlo en un ciclo, son necesarios dos para el funcionamiento correcto de la instrucción.

c. **(Examen - II/2012)** ¿Qué implicancia tiene en el tamaño de los programas el eliminar la conexión entre memoria de datos y `PC` (*program counter*) en el computador básico?

Eliminar esta conexión repercute en la implementación de subrutinas, ya que no sería posible cargar en el registro `PC` las líneas del código que se almacenan en el *stack* para su posterior retorno. Esto implicaría un tamaño mucho mayor en los programas, al tener que repetir código constantemente (un ejemplo es el caso de la multiplicación).

d. **(Apuntes - Saltos y subrutinas)** ¿Cómo se podría implementar en el computador básico la opción de que este avise luego de realizar una operación cuando el resultado es par o impar?

Esto se puede hacer de forma muy sencilla si nos damos cuenta del siguiente hecho: Todo número par en representación binaria termina en 0 y todo número impar termina en 1. Esto, ya que todo bit representa una potencia de 2, salvo el último que representa un 1. Por lo tanto, bastaría con añadir a la salida de la `ALU` una señal P equivalente al último bit del resultado (o su negación) y que fuera almacenada dentro del registro Status de forma directa. Solo faltaría una nueva instrucción asociada a un nuevo *opcode* (que podríamos llamar `JEN` -*Jump Even Number*- o `JON` - *Jump Odd Number*-, por ejemplo).

e. **(I1 - I/2016)** ¿Qué pasaría si se quita el registro `STATUS` del computador básico y se conectaran directamente las señales `ZNCV` a la unidad de control?

Al no estar conectadas a la señal *clock*, estas señales perderán la sincronización con el flanco de subida. Esto implica que las señales que estén ingresadas en la unidad de control no necesariamente sean correspondientes a la última instrucción ejecutada, causando posibles errores en las instrucciones de salto (y por ende, en todo el programa).

f. **(I1 - I/2017)** 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.

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.

g. **(Examen - I/2017)** Modifique la arquitectura del computador básico para que el registro `STATUS` se actualice solo después de la ejecución de una instrucción `CMP`.

La modificación más simple que se puede hacer para lograr el objetivo consiste en crear una nueva señal (llamémosla L_{stat}). L_{stat} será la señal que habilita la carga de datos en el registro `STATUS`. Finalmente, basta con que la Unidad de Control se encargue de transmitir $L_{stat} = 0$ para todo *opcode* que no corresponda a la instrucción `CMP`.

h. **(I1 - I/2018)** Modifique el *hardware* del computador básico para que las instrucciones `RET` y `POP` tomen un solo ciclo.

Una opción es hacer uso de un sumador de entradas `SP` y 1. Este puede tener su salida conectada a un registro llamado "`SP+1`" conectado al *clock* del computador básico, de forma que esté sincronizado con el resto de los registros. Luego, su salida puede estar conectada al `Mux Address` y, en caso de las operaciones `RET` y `POP`, se configura *Sadd* para su selección. Luego, se puede obtener `Mem[SP+1]` de forma inmediata para su posterior escritura en el registro que corresponda (ya sea `PC`, `A` o `B`). El siguiente diagrama muestra esta idea:

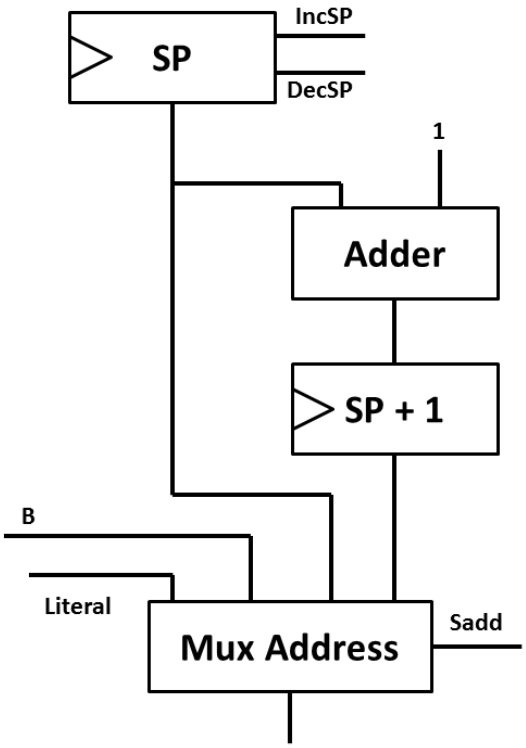


Figura 1: Diagrama del registro `SP+1`.

Esto, finalmente, permite reducir las operaciones `POP` y `RET` a un solo *opcode*, configurando correspondientemente al `Mux Address` para la selección del registro creado.

i. A partir del siguiente código `Assembly`, explique el flujo resultante dentro del computador básico, indicando el valor final de los registros `A` y `B`:

```
DATA:
  r 3 ;Resultado final
CODE:
  MOV A,(r)
  MOV B,2
  PUSH B
  CALL func
  POP B
  JMP finish
  func:
    shift:
      MOV A,B
      CMP A,0
      JEQ end
      DEC A
      MOV B,A
      MOV A,(r)
      SHL A,A
      MOV (r),A
      JMP shift
    end:
      RET
  finish: ;Termina el programa
```

1) `r 3:` Aquí se declara que el *label* `r` (una dirección de memoria) almacenará el literal 3. Al ser la primera (y única), se guarda por *default* en la dirección 0. Entonces, esto es equivalente a las instrucciones `MOV A,3` y `MOV (0),A`. Esto implica que, en primer lugar, se escribe el literal 3 en el registro `A` (habilitando su escritura previamente) y en el siguiente ciclo se habilita la escritura en la RAM y se escribe el valor del registro `A` en la dirección 0.
2) `MOV A,(r):` Se obtiene el dato de memoria en la dirección de `r` (que ya sabemos que es 0). Este dato se extrae de la RAM, lo seleccionamos para que pase por `Mux B` y permitimos la carga en el registro `A` para el resultado (que será la operación suma de la `ALU`, con `A = 0` y `B = Mem[0]`). Entonces, queda `A = Mem[0] = 3`.
3) `PUSH B:` Se almacena en la RAM el valor del registro `B`, en la posición `SP` (correspondiente a la última dirección). El valor de `SP` decrece (para almacenar el próximo valor).
4) `CALL func:` Guarda la posición `PC+1` en la dirección de memoria `SP`, esta señal decrece (para almacenar el siguiente valor), y se carga en `PC` el número correspondiente a la línea de código donde se encuentra la subrutina func (para ejecutar desde ahí).
5) `MOV A,B:` Se guarda en el registro `A` lo que está almacenado en `B` (en un comienzo, el valor igual a 2).
6) `CMP A,0:` Se realiza la operación `A-0` (en este caso, `2-0`) y se guardan las *flags* en el registro `Status` para el próximo comando a ejecutar.

7) `JEQ end:` Desde la unidad de control se revisa si la *flag* `Z` es igual a 0. Como esto no pasa, no se habilita la señal L_{pc} y el `PC` aumenta en uno para proceder con la siguiente línea de código.

8) `DEC A:` Equivalente a `SUB A,1`. El registro almacenado en `A` decrece en una unidad (en este caso entonces, queda `A = 1`).

9) `MOV B,A:` Se guarda en el registro `B` el valor almacenado en `A` (por lo que queda `B = 1`).

10) `MOV A,(r):` Se obtiene el dato de memoria en la dirección de `r = 0`. Este dato se extrae de la RAM, lo seleccionamos para que pase por `Mux B`, y permitimos la carga en el registro `A` para el resultado (que será la operación suma de la `ALU`, con `A = 0` y `B = Mem[0]`). Entonces, queda `A = Mem[0] = 3`.

11) `SHL A,A:` Almacenamos en `A` el resultado de hacerle un `shift left` al número. En este caso, al ser equivalente a una multiplicación por 2, queda `A = 6`.

12) `MOV (r),A:` Se habilita la escritura en la RAM y en la dirección `r = 0` se almacena el valor del registro `A` (entonces, `Mem[0] = 6`).

13) `JMP shift:` Se realiza un salto incondicional a la *label* `shift` (es decir, L_{pc} `= 1`, y se carga la línea de código correspondiente para la próxima ejecución).

14) `MOV A,B:` Se guarda en el registro `A` lo que está almacenado en `B` (ahora, un valor igual a 1).

15) `CMP A,0:` Se realiza la operación `A-0` (en este caso, `1-0`) y se guardan las *flags* en el registro `Status` para el próximo comando a ejecutar.

16) `JEQ end:` Desde la unidad de control se revisa si la *flag* `Z` es igual a 0. Como esto no pasa, no se habilita la señal L_{pc} y el `PC` aumenta en uno para proceder con la siguiente línea de código.

17) `DEC A:` Equivalente a `SUB A,1`. El registro almacenado en `A` decrece en una unidad (en este caso entonces, queda `A = 0`).

18) `MOV B,A:` Se guarda en el registro B el valor almacenado en `A` (por lo que queda `B = 0`).

19) `MOV A,(r):` Se obtiene el dato de memoria en la dirección de `r = 0`. Este dato se extrae de la RAM, lo seleccionamos para que pase por `Mux B` y permitimos la carga en el registro `A` para el resultado (que será la operación suma de la `ALU`, con `A = 0` y `B = Mem[0]`). Entonces, queda `A = Mem[0] = 6`.

20) `SHL A,A:` Almacenamos en `A` el resultado de hacerle un `shift left` al número. En este caso, al ser equivalente a una multiplicación por 2, queda `A = 12`.

21) `MOV (r),A:` Se habilita la escritura en la RAM y en la dirección `r = 0` se almacena el valor del registro `A` (entonces, `Mem[0] = 12`).

22) `JMP shift:` Se realiza un salto incondicional a la *label* `shift` (es decir, L_{pc} `= 1` y se carga la línea de código correspondiente).

23) `MOV A,B:` Se guarda en el registro `A` lo que está almacenado en `B` (ahora, un valor igual a 0).

24) `CMP A,0:` Se realiza la operación `A-0` (en este caso, `0-0`) y se guardan las *flags* en el registro `Status` para el próximo comando a ejecutar.

25) `JEQ end:` Desde la unidad de control se revisa si la *flag* `Z` es igual a 0. Como esto pasa, se habilita la señal L_{pc} y el registro `PC` carga la dirección de la línea de código del *label* `end`.

26) `RET: SP` aumenta en 1 y se carga en el registro `PC` lo almacenado en `Mem[SP]` (en este caso, la línea de código correspondiente a la siguiente de donde se hizo el llamado a la función).

27) `POP B: SP` aumenta en 1 y se carga en el registro `B` lo almacenado en `Mem[SP]` (en este caso, el valor original del registro `B` antes de ingresar a la función, `B = 2`).

28) `JMP finish:` Se realiza un salto incondicional a la *label* `finish` (es decir, L_{pc} `= 1`, y se carga la línea de código correspondiente). Como aquí ya no hay más código, es la última ejecución a realizar (la ROM suele tener después solo líneas de código que hacen que no se ejecute nada dentro del computador, manteniendo el estado final).

Finalmente, podemos ver que los valores finales de nuestros registros son A = 0 y B = 2. Notemos que el valor de A **no es** 12, ya que este cambió dentro de la rutina, y no terminamos por almacenarle el resultado escrito en memoria. Sí sabemos que en la memoria lo calculado se mantuvo (Mem[0] = 12).

2. a. **(Examen - II/2016)** Modifique la arquitectura del computador básico para que funcione con lógica ternaria en vez de binaria. Más específicamente, modifique los tamaños de los elementos (buses, registros, señales de control, etc.) de modo que el nuevo computador tenga una capacidad similar a la versión binaria. Asuma que existen todos los componentes vistos en clases en versión ternaria. **Nota:** No es válido utilizar los valores ternarios como si fueran binarios.

Aquí, lo relevante es ver cómo cambia la representación de datos que se tenía antes. Con un trit, podemos representar un número más que con un bit, con dos trits, podemos representar 4 números más que con dos bits, y así sucesivamente. Por lo tanto, iremos estudiando cómo cambiaría cada parte del computador básico:

- **Registros y buses de datos:** Estos almacenaban 8 bits. Para poder abarcarlos todos, resolvemos el siguiente problema para K trits y N bits:

$$\begin{array}{lcl} 3^K & = & 2^N \\ \log_3(3^K) & = & \log_3(2^N) \\ K & = & N * \log_3(2) \\ K & = & \lceil N * \log_3(2) \rceil \end{array} \tag{1}$$

Notar que como K debe ser un número entero, usamos la función techo para aproximar al entero mayor (si usáramos la función piso, podríamos perder números representables). Por lo tanto, para $N = 8$ bits, tenemos que $K = \lceil 5{,}047 \rceil = 6$.
Otra forma de obtener el resultado es simplemente ver el intervalo de representación con N bits:

$$-\frac{2^N}{2} \leq x \leq \frac{2^N}{2} - 1$$

Y tomando el intervalo para los trits:

$$-\left\lfloor \frac{3^K}{2} \right\rfloor \leq x \leq \left\lfloor \frac{3^K}{2} \right\rfloor$$

Buscamos el K tal que el intervalo de representación de bits (con $N = 8$) sea considerado **por completo**. (Notar que nos importa representar todo, da lo mismo si nuestro poder de representación aumenta, no queremos que disminuya). De esta forma, también se llega a $K = 6$.
Es importante ver que esto no lo cambiamos para el registro `STATUS`, ya que aquí nos interesa ver si cada *flag* cumple o no para generar los saltos (utilizar base trinaria dificultaría un poco el funcionamiento de este).

- **Señales de control:** Algunas de las señales de control se pueden abreviar de la siguiente forma:
  - Selector de operaciones de la `ALU`: Aquí tenemos 8 operaciones que se seleccionan a partir de 3 bits. Sin embargo, bastan 2 trits para tener 8 combinaciones diferentes, y así, escoger la operación correspondiente.
  - Incrementar/decrementar `SP`: Ahora, podemos usar un trit para las siguientes combinaciones: 0 (no incrementar ni decrementar), 1 (incrementar) y 2 (decrementar).
  - Selector del Mux `A` y Mux `Address`: Como escogemos entre tres señales para estos dos multiplexores, podemos quedarnos con un solo trit para obtener las 3 combinaciones. Notar que esto no sirve para el `Mux B`, ya que se selecciona entre 4 entradas (por lo que se siguen necesitando dos trits).

Estos corresponden a los cambios aplicables para mantener el funcionamiento de nuestro computador, utilizando trits en vez de bits.

b. **(I1 - II/2016)** En esta pregunta deberá diseñar un computador especializado en el manejo de matrices. El computador debe ser capaz de: i) copiar una matriz desde la memoria de datos a un registro y viceversa, ii) sumar 2 matrices almacenadas en registros distintos y almacenar el resultado en un registro.

   I. Haga el diagrama del computador, considerando que las matrices pueden tener como máximo $N \times N$ elementos, cada uno de 1 byte.
   En este caso, no es necesario modificar de forma significativa el diagrama del computador básico. Basta con aumentar el tamaño de los registros, buses de datos, `ALU` y las palabras de memoria a N^2 bytes para que puedan almacenar la matriz en su totalidad.

   II. Diseñe el *assembly* del computador. Cada instrucción debe estar asociada a un *opcode* y estos a sus respectivas señales de control.
   En base al diagrama anterior, basta utilizar el *assembly* del computador básico. La única diferencia se da en los literales, donde existen dos opciones: i) separar cada uno de los elementos de una matriz con coma (,) y ii) declarar la matriz completa como un único número de tamaño N^2 bytes (este último hace más sentido dada la representación anterior).

   III. Agregue tanto al *hardware* como al *assembly* soporte para una instrucción que permita modificar el valor de un elemento arbitrario de una matriz almacenada en la memoria de datos.
   Una posible solución es agregar unidades de *shifting* para aislar el elemento buscado y modificarlo y luego realizar operaciones lógicas para integrar este valor con la matriz completa. Notar que esto añade una gran complejidad en términos de *hardware*, pero logra cumplir el objetivo.

c. **(I1 - I/2018)** El computador básico solo trabaja con números enteros, sin embargo, para muchas aplicaciones es útil poder trabajar con números decimales. En consecuencia, usted deberá añadir soporte para números decimales al computador básico.

I Indique un esquema de números decimales que podría emplear en el computador básico, mencionando sus ventajas y desventajas.
Un esquema posible es el uso de precisión fija decimal. De este modo se utiliza un número fijo de bits para parte entera y parte decimal. Tiene la ventaja de ser relativamente simple y barato de implementar, pero la desventaja de estar limitado en el rango de números que puede abarcar. Otros esquemas pueden ser esquemas de punto flotante como `IEEE754`.

II Añada los componentes necesarios para trabajar con el esquema de números decimales argumentados en la parte **I.**, indicando qué hace cada componente y cómo se conecta a las partes existentes del computador básico, detallando su interacción con este. Si bien no es necesario hacer a nivel de compuertas todos los componentes, no se aceptarán componentes mágicos como "Unidad de cómputo decimal". Sí se aceptará la `ALU`, registros de n bits y otros componentes desarrollados en clases.
Es posible añadir 4 registros de 8 bits, iguales a los `A` y `B` del computador básico, que pueden ser llamados `DAi`, `DAd`, `DBi` y `DBd`, por `Decimal{A,B}(integer/decimal)`. Dichos registros tendrán sus `DataIn` conectados a la salida de la `ALU` y sus salidas conectadas a una `ALU` de partes decimales y otra de partes enteras. Además, el `CarryOut` de la `ALU` de partes decimales estará conectada al `CarryIN` de la `ALU` de partes enteras. Las salidas de dichas `ALUs` realimentarán las rutas que se alimentaban originalmente de la salida de la `ALU` normal. Este esquema hace que se requieran 4 cargas para cargar los operandos decimales y 2 ciclos para guardar sus resultados. Además, modifica las operaciones enteras para que pasen sin operar en las `ALUs` de decimales.

Otro acercamiento válido es usar los registros y `ALUs` al mismo nivel de la `ALU` normal, incluso pudiendo reutilizar esta con los nuevos registros, y usar un `Mux` de salida para seleccionar qué parte del resultado es la que se carga en registro o se guarda en memoria.