Nro. ord.	Apellido y nombre	L.U.

Organización del Computador I - **Parcial**

Primer Cuatrimestre 2022

Ej.1	Ej.2	Ej.3	Ej.4	Ej.5	Nota

Corrector:

Aclaraciones

- Anote apellido, nombre, LU en *todos* los archivos entregados.
- El parcial es domiciliario y todos los ejercicios deben estar aprobados para que el parcial se considere aprobado. Hay dos fechas de entrega, en ambos casos el conjunto de ejercicios a entregar es el mismo. En la primera instancia deberán defender su trabajo frente a su tutorx, quien les ayudará también a encaminar el trabajo de los ejercicios pendientes, si los hubiera.
- El link de entrega es: `https://forms.gle/re4UQkWJXKCvtYMA9`. Ante cualquier problema pueden comunicarse con la lista docente o preferentemente con el/la corrector/a.
- La fecha límite de entrega es el domingo 26 de Junio a las 21:00. El coloquio será el jueves 7 de Julio de cursada de los jueves (TM: 9 a 13hs - TT: 17 a 21hs) de **forma presencial** en un aula que confirmaremos en breve.
- Todas las respuestas deben estar correctamente justificadas y de corresponder hacer una comparación con la máquina Orga1.
- El archivo a adjuntar puede ser `.pdf` o `txt`, el nombre del archivo debe cumplir con el siguiente formato `orga1_nombre_apellido_parcial.pdf` o `orga1_nombre_apellido_parcial.txt`.

Introducción

Este parcial está dividido en seis preguntas, las primeras cuatro se relacionan con la arquitectura RISC-V. Se trata de una especificación y no una implementación de un procesador concreto. Si bien la arquitectura es diferente a la que utilizamos en clase, encontrarán los mismos conceptos que estudiamos y trabajamos aplicados al diseño de esta ISA. Toda la información necesaria está disponible en la `Guía Práctica de RISC-V` que se puede acceder libremente en: `http://riscvbook.com/spanish/guia-practica-de-risc-v-1.0.5.pdf`. Les recomendamos que hagan primero una lectura completa de los primeros tres capítulos de la guía y luego intenten responder las preguntas. Los ejercicios varían en temática y complejidad así que también les recomendamos ordenar la resolución de los mismos como les resulte más sencillo. Las últimas dos preguntas se relacionan con extensiones que introdujimos a nuestra implementación de `Orga1SmallI`

Ejercicios

Ejercicio 1 Sabiendo que `a1 = 0xffffffff`, ¿Cuánto queda almacenado en `a2` luego de realizar la operación: `andi a2,a1,0xf0f`?

Ejercicio 2 ¿Cómo se modifica el valor del registro (`x0`)? ¿Cómo maneja las escrituras a este registro y por qué lo hace de esa forma?

Ejercicio 3 ¿Cómo resuelve **BGT**(branch great than) con `RISC-V`?

Ejercicio 4 Quéremos agregar un flag llamado `P` (paridad) que nos indica si el resultado de una operación en la ALU es par. ¿Qué cambios hacen falta hacer a `Orga1SmallI` para implementarlo? Descríbalos en detalle. ¿Cómo resolvería un salto condicional (`JP`) por paridad? Suponga que tiene espacio libre para la instrucciones en la memoria de la unidad de control y que puede agregar tantas señales a la misma como hagan falta.

La necesidad de realizar numerosas operaciones sobre datos estructurados en memoria parece seguir creciendo mientras que el costo y la escala de las operaciones realizadas por el procesador se vuelven más costosas en tiempo e infraestructura. Debido a esto se están evaluando alternativas para realizar diversas operaciones en memoria sin tener la obligación de mover datos entre los registros y la memoria principal.

Proponemos el uso de un controlador de memoria con las siguientes señales de control:

- `mem_op` es una señal de tres bits que indican la operación a realizar
  - `000` no realiza ninguna operación
  - `001` permite guardar el valor del registro de destino (`wrAddr`)
  - `010` permite guardar el valor del registro de fuente (`rdAddr`)
  - `011` realiza una escritura en la dirección de destino desde el valor encontrado en inData (equivale al `load` de nuestra arquitectura)
  - `100` suma el valor que se encuentra en la dirección de fuente al valor que se encuentra en la direcicón de destino y lo guarda en la dirección de destino
  - `101` multiplica el valor que se encuentra en la dirección de fuente por el valor que se encuentra en la direcicón de destino y lo guarda en la dirección de destino
  - `110` limpia el valor que se encuentra en la dirección de destino
  - `111` mueve el valor que se encuentra en la dirección de fuente (`rdAddr`) y lo copia en la dirección de destino (`wrAddr`)
- `enOut` permite volcar el valor encontrado en la direción de fuente en el registro outData

Y las siguientes señales de datos:

- `wrAddr` indica la dirección de destino (la dirección de la palabra donde realizaremos una escritura)
- `rdAddr` indica la dirección de fuente (la dirección de la palabra donde realizaremos una lectura)
- `inData` indica el valor de la palabra a guardar en la dirección de destino
- `outData` indica el valor de la palabra a leer de la dirección de fuente

Ejercicio 5 Suponiendo que las señales de control están conectadas a la unidad de control (`mem_op`, `enOut`) y los registros de datos al bus (`wrAddr, rdAddr, inData, outData`):
Escriba un microprograma que permita mover el contenido del valor que se encuentra en la dirección referida por el registro indicado en el **operando Y del decoder** a su destino en la dirección referida el registro indicado en el **operando X del decoder**.

Organización del Computador I - **Parcial**

Jonathan Bekenstein - LU 348/11

2 de agosto de 2022

---

1. Ejercicio 1

La operación `ANDI rd,rs1,imm` realiza un AND lógico bit a bit entre rs1 y el valor inmediato imm, colocando el resultado en el registro destino rd.

Es importante destacar que todos los registros de RISC-V (RV32I específicamente) son de 32 bits, mientras que el valor inmediato es de 12 de bits. A diferencia de otras arquitecturas, RISC-V extiende el signo de los valores inmediatos de forma automática antes de realizar la operación.

Por lo tanto, en la instrucción `ANDI a2,a1,0xf0f` el valor inmediato `0xf0f` es primero extendido a `0xfffff0f`. Luego se realiza un AND entre éste valor y el valor del registro a1 que por el enunciado sabemos que es `0xffffffff`. Por lo tanto el resultado de la operación es `0xffffffff & 0xfffff0f = 0xfffff0f` y se guarda en el registro a2.

Comparación con Orga1

La máquina Orga1 posee una única instrucción AND, pero a diferencia de RISC-V soporta varios modos de direccionamiento. Esto permite utilizar una misma instrucción con distintas configuraciones de operandos (entre registros, registro a memoria, con valor inmediato, etc), sin necesidad de definir instrucciones distintas para cada combinación de tipos de operandos.

Otra diferencia es que las instrucciones de Orga1 aceptan como máximo 2 operandos, mientras que las de RISC-V pueden aceptar hasta 3. Notemos que si en RISC-V se usa algún registro fuente también como destino, en esencia estamos en la misma situación que ofrece Orga1.

Por lo tanto la instrucción ANDI de RV32I se puede realizar en Orga1 con la instrucción AND utilizando el modo de direccionamiento a registro para el operando destino, y el modo inmediato para el operando fuente, y asumiendo que en la versión de RISC-V se usó algún registro fuente también como registro destino.

Orga1 no tiene la necesidad de extender el signo de los valores inmediatos porque éstos siempre tienen el tamaño de una palabra (16 bits), al igual que los registros.

2. Ejercicio 2

El registro x0 está cableado a 0. Esto significa que no se puede modificar su valor. Por más que las instrucciones permitan usar al registro x0 como registro destino, su valor nunca cambia. Y al usar el registro x0 como operando de alguna instrucción, su valor siempre será 0.

Recordemos que RISC = reduced instruction set computer. Por lo tanto se podría decir que uno de los objetivos principales de RISC-V es mantener el ISA lo más chico posible. Esto quiere decir que hay muchísimas instrucciones "faltantes" en el ISA de RISC-V (puntualmente RV32I) por el simple hecho de que se pueden realizar con alguna otra instrucción existente, ya sea intercambiando el orden de los operandos o previamente seteando algún registro en un valor en particular.

Para simplificar la vida del programador, RISC-V posee pseudo-instrucciones, que son como un "alias" de una o más instrucciones. Estas pseudo-instrucciones se transforman en instrucciones durante el proceso de ensamblado, y básicamente agregan expresividad al lenguaje assembler de RISC-V (permiten escribir código más conciso, simplifican la cantidad de operandos de las instrucciones, o simplemente aportan mejor legibilidad al programa).

Algunas de las pseudo-instrucciones provistas por RV32I son saltos condicionales comparando con 0, o poner en 1 un registro en base a la comparación con 0. Estas operaciones se realizan utilizando las instrucciones provistas por RV32I que son más genéricas, es decir, permiten comparar con cualquier valor, no solo con el 0.

El registro x0 resulta muy práctico junto a las pseudo-instrucciones. En RV32I hay 32 de ellas que usan al registro x0 como operando (también podemos usarlo en nuestros propios programas). La ventaja del registro x0 en este contexto es que muchas instrucciones operan únicamente entre registros (por ejemplo las instrucciones de salto condicional BXX). Si necesitamos que uno de esos registros sea 0, tendríamos que primero cargar un 0 en algún registro y luego ejecutar la instrucción deseada. Gracias al registro x0 nos ahorramos ese paso ya que x0 siempre vale 0. No solo simplificamos el código (comparar con 0 requiere un operando menos que la comparación genérica) sino que también resulta más eficiente porque nos ahorramos una instrucción. Otras pseudo-instrucciones que usan el registro x0 simplemente mejoran la legibilidad pero no son más eficientes que haber usado las instrucciones directamente (como por ejemplo MV y RET).

Comparación con Orga1

Orga1 admite varios modos de direccionamiento, entre ellos el inmediato. Si utilizamos 0 como valor inmediato, podemos lograr algo muy parecido al registro x0 de RISC-V. No obstante, utilizar el valor inmediato tiene la penalidad de requerir cargar 1 palabra adicional de memoria (el 0) durante el fetch. Más allá del rendimiento de la máquina, desde la óptica del programador, estamos a un solo caracter de distancia entre escribir x0 ó 0x0.

3. Ejercicio 3

BGT es una pseudo-instrucción. El ensamblador transforma BGT (branch if greater than) por BLT (branch if less than) dando vuelta los operandos. Es fácil ver que la condición de salto de ambas instrucciones son equivalentes si se invierte el orden de los operandos.

Las instrucciones de salto condicional utilizan el formato B, que para aumentar el rango de salto, se multiplica el valor inmediato por 2 durante el ensamblado (shift lógico hacia la izquierda de 1 bit). Esto funciona de forma segura porque todas las instrucciones de RISC-V tienen un tamaño múltiplo de 2 bytes. Es más, la razón de multiplicar por 2 es porque existe una extensión compacta de RISC-V llamada RV32C donde las instrucciones tienen un tamaño de 2 bytes. Si no fuera por RV32C, se podría lograr un rango de salto aún mayor multiplicando por 4 ya que todas las instrucciones de RV32I tienen un tamaño de 4 bytes.

Al multiplicar por 2 el valor de offset, en esencia sabemos que el resultado será un número par y consecuentemente el bit menos significativo será siempre 0. Es por esto que el bit menos significativo del offset está omitido en el formato de instrucción B, y así ganamos 1 bit más de "resolución" (podemos saltar más lejos). Luego a nivel circuito se coloca siempre un 0 en el bit menos significativo del offset.

Los saltos condicionales en RISC-V se ensamblan en modo "PIC" = position independent code. Es decir, en vez de especificar un valor absoluto de memoria a dónde saltar, se especifica el offset (desplazamiento) relativo al PC. Si se cumple la condición de salto, tenemos que sumar offset al valor actual del PC.

Notemos que el PC es un registro de 32 bits, mientras que el offset está representado con 13 bits en complemento a 2 (12 bits embebidos en la instrucción más el 0 colocado a la fuerza en el bit menos significativo). Por esto es necesario extender el offset a 32 bits preservando el signo antes de realizar la suma, y así permitir correctamente saltos hacia adelante o atrás (offsets positivos o negativos).

Podemos describir la instrucción BGT de la siguiente forma:

```
BGT rs2,rs1,offset ≡ BLT rs1,rs2,offset ≡ if (rs1 < rs2) pc += signExt(offset << 1)
```

A expensas de un formato de instrucción un poco rebuscado (los bits del valor inmediato parecen tener una ubicación arbitraria dentro del formato de instrucción), el bit más significativo del valor inmediato es a su vez el bit más significativo de la instrucción. Es decir, el bit más significativo del valor inmediato, sin importar

su tamaño, aparece siempre en el mismo lugar de la instrucción. Esto simplifica el hardware ya que ahora el circuito de extensión de signo siempre tiene que mirar el mismo bit de la instrucción para decidir qué hacer.

Otro detalle relevante para los saltos condicionales es que éstos no revisan su condición de salto en base a los flags de la ALU. En cambio, las instrucciones de salto condicional reciben 2 registros los cuales son comparados para determinar si se debe saltar o no. La ventaja de esto es que permite optimizar la ejecución fuera-de-orden durante el proceso de pipelining, ya que al no usar los flags de la ALU, la condición de salto no sería afectada por otras operaciones realizadas fuera-de-orden.

Comparación con Orga1

Orga1 al igual que RISC-V posee instrucciones de salto condicional "redundantes" para simplificar la tarea del programador. No obstante, Orga1 no posee pseudo-instrucciones, así que todas las instrucciones que brinda requieren opcodes distintos.

Respecto a las condiciones de salto, Orga1 es fundamentalmente distinta a RISC-V en que las condiciones se basan exclusivamente en los flags de la ALU. Si bien a alto nivel es lo mismo y se pueden lograr las mismas funcionalidades, la arquitectura de Orga1 impone una dependencia temporal entre las instrucciones. Los flags que usa la instrucción de salto condicional fueron actualizados durante la última operación de la ALU (solo si la operación modificaba los flags). Esto quiere decir que hay que tener cuidado con lo que pasa entre la instrucción que modificó los flags y la instrucción de salto condicional. Si bien Orga1 no tiene interrupciones, éstas probablemente generen un problema si no se tiene en cuenta de salvar el estado de los flags y luego restaurarlos al terminar de atender la interrupción.

Respecto al offset, conceptualmente funciona igual que RISC-V. La instrucción de salto condicional contiene un valor de 8 bits en complemento a 2 que indica la cantidad de palabras a saltar. Al realizar la suma con el PC, se deberá extender el signo del offset a 16 bits.

4. Ejercicio 4

Para agregar el nuevo flag P tenemos que modificar la ALU de la siguiente forma:

1. Agregar un nuevo circuito en la ALU para determinar la paridad del resultado de la operación realizada. Este circuito utiliza la salida del multiplexor que selecciona el circuito de la operación realizada y lo conecta con la entrada de datos del registro ALU_OUT. Para detectar si un número es par podemos mirar el bit menos significativo. Si éste vale 0, entonces el número es par, caso contrario es impar. Por lo tanto, el circuito no es más que un splitter para obtener el bit menos significativo y luego negarlo ya que queremos que $P = 1 \iff b_0 = 0$.
2. Enviar el resultado del circuito de paridad a un nuevo registro interno de la ALU llamado P. Éste está configurado de la misma forma que los registros de los otros flags, se escribe solo si la señal opW=1. Así podemos usar la ALU para operaciones intermedias en un microprograma en donde no queremos modificar los flags.
3. Agregar 1 bit adicional al pin de salida flags de la ALU (ahora es una señal de 4 bits ya que hay 4 flags). Conectar el bit 3 de este pin a la salida del registro P.

Si bien nuestra ALU ya puede calcular el flag P que indica la paridad del resultado de alguna operación, no podemos usarlo ya que la unidad de control aún no está cableada de ninguna forma con este nuevo flag.

Agregamos entonces una nueva instrucción a la ISA que permite realizar un salto condicional en función del flag P. El formato de la instrucción es el siguiente:

Instrucción	CodOp	Formato	Acción
JP M	01100	C	Si flag_P=1 entonces PC ← M

Utilizamos el código de operación (CodOp) $01100_2 = 12_{10}$ ya que se encuentra disponible, y elegimos el formato C porque la operación JP necesita recibir un único parámetro de 8 bits: la dirección de memoria a donde saltar si se cumple la condición.

Las microinstrucciones (una línea del microprograma) son codificadas en una palabra de 32 bits que es guardada en la memoria de la unidad de control. Una microinstrucción contiene todas las señales de control que se deben activar (setear en 1) durante ese ciclo de clock. Para traducir una microinstrucción a su correspondiente palabra de 32 bits miramos cada señal que se debe activar durante ese ciclo de clock y colocamos un 1 en el bit correspondiente a cada señal. "Ejecutar" una microinstrucción es simplemente seleccionar su dirección en la memoria de la unidad de control. La salida de la memoria se descompone en 32 cables individuales, donde cada cable lleva una determinada señal de control (el cable del bit 0 lleva la señal de control asociada al bit 0 y así sucesivamente con los 32 bits).

Los microprogramas de los saltos condicionales de la máquina `Orga1SmallI` están implementados en sí mismos con saltos condicionales. Éstos saltos dentro de los microprogramas se computan con un circuito dedicado que modifica el microPC en función de la condición de salto. Para esto, se require una señal en la unidad de control que indica que queremos saltar condicionalmente en base a algún flag en particular. El circuito de salto realiza un AND entre ésta señal y la señal del flag que determina el salto (es decir algún bit de la entrada flags de la unidad de control). El resultado de este AND controla un multiplexor de 2 entradas para seleccionar entre las constantes `0x01` o `0x02`, que luego es sumada al microPC. Si se incrementa el microPC en 1, significa que no hay salto y termina el microprograma. Pero si se incrementa el microPC en 2, nos salteamos la microinstrucción de "fin"(reset_microOp) y ahora tenemos espacio para la secuencia de microinstrucciones necesarias para realizar el salto intencionado (en esencia, cargar el valor M en el PC). La estructura de los microprogramas de los saltos condicionales y el circuito descripto están fuertemente vinculados ya que trabajan en conjunto y se debe respetar ciertas convenciones para su correcto funcionamiento.

Para implementar el salto condicional por paridad (JP) tenemos que modificar la unidad de control de la siguiente forma:

1. Agregar una nueval señal jp_microOp en la unidad de control. Podemos utilizar el bit 23 ya que se encuentra disponible.
2. Agregamos un nuevo AND entre jp_microOp y el bit 3 de la señal de flags que corresponde al flag P. La salida de este nuevo AND se conecta con el OR entre los ANDs de los otros saltos condicionales.
3. Conectamos la señal jp_microOp en el OR entre las otras señales de salto condicional: jc_microOp, jz_microOp, jn_microOp. Esto no es necesario para el circuito de salto en sí mismo, existe porque el manejo de interrupciones (puntualmente la decisión de ejecutar la RAI o no) require de un salto condicional en el microprograma del fetch de 7 direcciones en vez de 2.

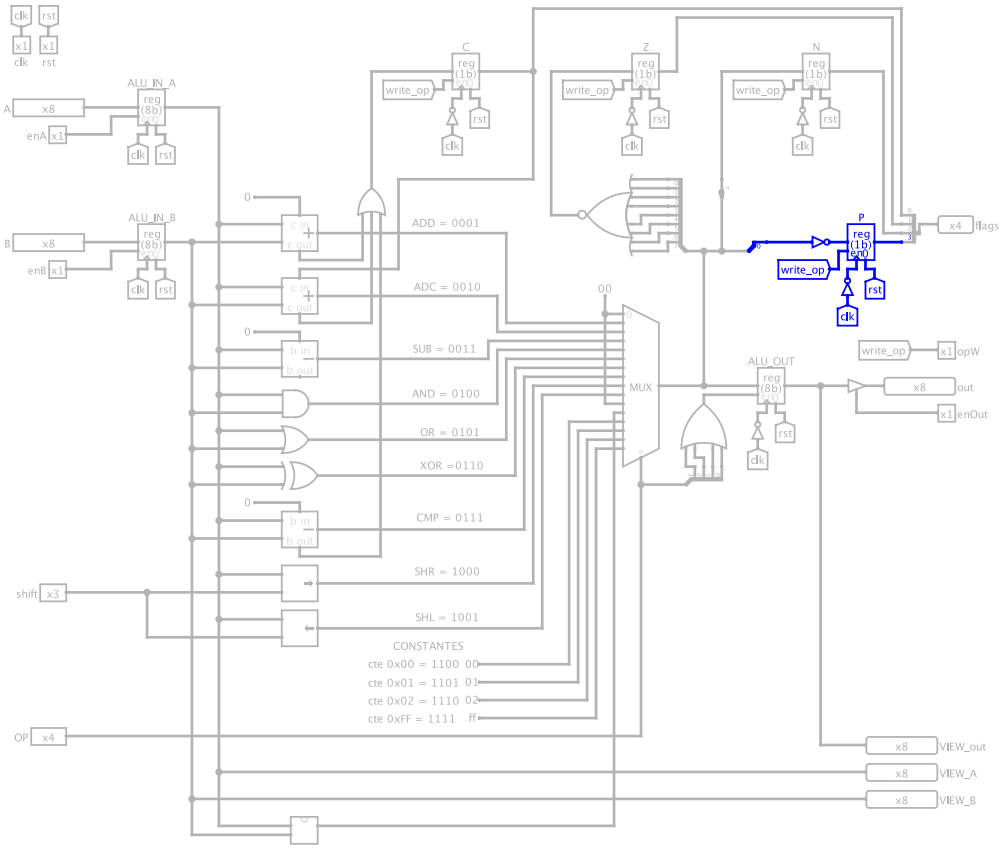
Por último necesitamos agregar el nuevo microprograma para la instrucción JP.

```
1: 01100:                                          ▷ Código de operación de la instrucción JP
2:   JP_microOp load_microOp          ▷ JP_microOp AND flag_P = 1 ⟹ microPC = microPC + 2
3:   reset_microOp               ▷ flag_P = 0 ⟹ no hay salto, terminamos el microprograma
4:   DE_enOutImm PC_load       ▷ flag_P = 1 ⟹ saltamos a la dirección M (valor inmediato)
5:   reset_microOp
```

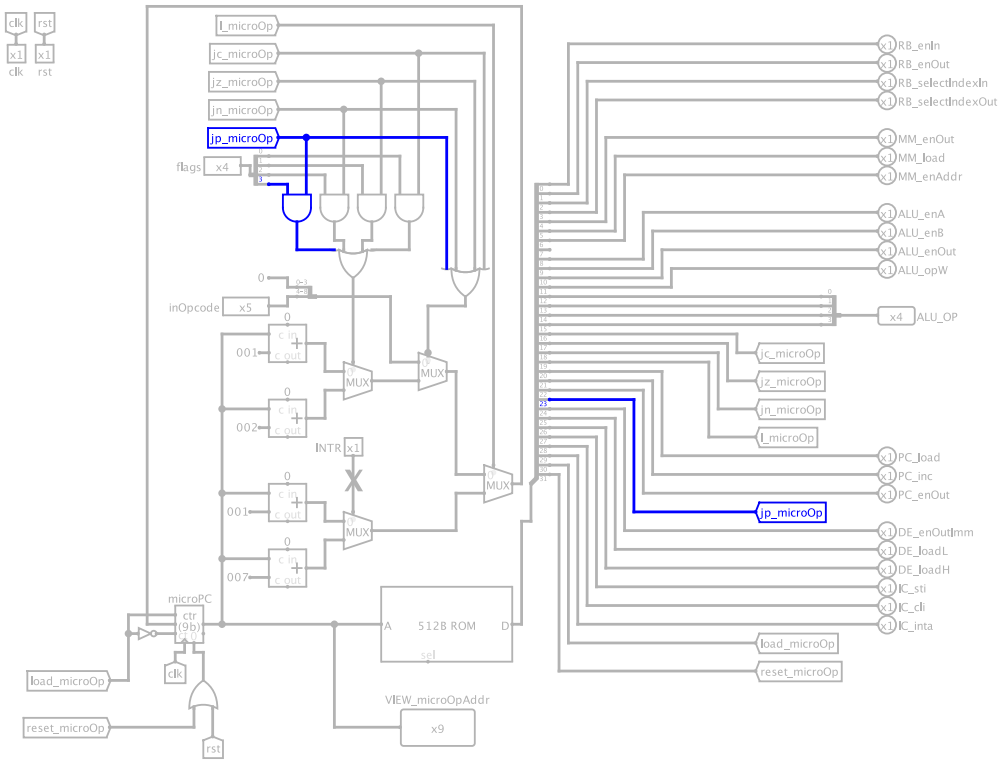
A modo de ejemplo y prueba de funcionamiento, el siguiente programa utiliza la nueva instrucción JP. Después del primer incremento R0 vale 1, y por lo tanto no se ejecuta el salto de la línea 5. Después del segundo incremento, R0 vale 2 y es par, por lo tanto sucede el salto de la línea 7 y termina la ejecución del programa. El tercer incremento nunca sucede, y al finalizar la ejecución R0 termina valiendo 2.

```
1: init:
2: SET R0, 0x00
3: main:
4: INC R0
5: JP halt                                                  ▷ No salta porque R0 = 1
6: INC R0
7: JP halt                                                ▷ Ahora sí salta porque R0 = 2
8: INC R0
9: halt:
10: JMP halt
```

Modificaciones en la ALU



Modificaciones en la Unidad de Control



En la carpeta “ej4” incluída en la entrega se encuentran todos los archivos de la máquina `Orga1SmallI` con las modificaciones detalladas para poder utilizar la instrucción JP (también se modificó el script Python para poder ensamblar los microprogramas y programas).

5. Ejercicio 5

El microprograma pedido podría corresponder a la siguiente instrucción.

Instrucción	CodOp	Formato	Acción
MOV [Rx], [Ry]	01100	A	Mem[Rx] ← Mem[Ry]

El microprograma de esta operación delega gran parte del trabajo al nuevo controlador de memoria. No obstante, es necesario ejecutar algunas microinstrucciones iniciales para configurar el controlador de memoria antes de pedirle que haga la copia. En esencia, le tenemos que decir desde qué dirección vamos a leer y a cuál escribir. Se utiliza la nomenclatura `bXXX` para indicar una constante en representación binaria.

```
1: 01100:                                  ▷ Código de operación de la instrucción MOV [Rx], [Ry]
2:    ; Colocamos el valor de Rx en el bus y lo guardamos en el registro destino wrAddr
3:    RB_enOut RB_selectIndexOut=0 mem_op=b001
4:    ; Colocamos el valor de Ry en el bus y lo guardamos en el registro fuente rdAddr
5:    RB_enOut RB_selectIndexOut=1 mem_op=b010
6:    ; Ejecutamos la operación de copia dentro del controlador de memoria
7:    mem_op=b111
8:    reset_microOp
```

Un detalle importante que asume este microprograma es que las operaciones del controlador de memoria requieren un único ciclo de clock. Caso contrario, se podrían agregar microinstrucciones nulas para esperar al controlador de memoria antes de terminar el microprograma y así asegurarse que las próximas instrucciones disponen de la memoria en su estado esperado.