

### Ejercicio 1

*¿Hacer un desplazamiento en un registro es más rápido que buscar el valor que se ha de desplazar en memoria?*

Sí, dado que cada uno de los registros están directamente conectados al Bus, entonces son más rápidos de leer y escribir mientras que los accesos a memoria consumen más energía ya que se accede a través de un controlador que selecciona una posición dada una dirección.

*A partir del inciso anterior, ¿qué habría de tener en cuenta a la hora de revisar el rendimiento de su programa?*

Dado que el acceso de datos en registros es mucho más rápido que en memoria, es crucial que los compiladores hagan una buena asignación de registros. Dicha tarea es más fácil entre más registros se tenga y esto mejoraría el rendimiento del programa. Esto es justamente lo que hace RISC-V, ofrece una cantidad generosa de 32 registros y además tiene instrucciones para tareas específicas las cuales son entre registros, ninguna es de registro a memoria, entonces solo se accede a memoria cuando es necesario manteniendo así simplicidad

### 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?*

El valor del registro (x0) no se puede modificar ya que este se encuentra directamente conectado a una constante cero. Tener un registro a cero sirve para simplificar el ISA ya que en las operaciones pueden ser creadas simplemente usando el registro cero como operando. Dado que esta conectado directamente a cero no se le puede hacer ninguna escritura y cuando lee el registro siempre lee cero. Además, hay pseudoinstrucciones que son configuraciones ingeniosas de instrucciones normales usando el registro x0, es decir, instrucciones que utilizan el registro x0 como uno de sus parámetros, las cuales dependen que el registro sea siempre 0. Esto simplifica significativamente el conjunto de instrucciones de RISC-V haciendo que muchas instrucciones se hagan con pseudoinstrucciones. (En la página 38 hay varios ejemplos de instrucciones usando el registro x0)

### Ejercicio 3

*¿Qué significa que la arquitectura RISC-V sea modular?*

- El autor no menciona que las ISAs incrementales no solo implementan las nuevas extensiones sino todas las instrucciones de ISAs anteriores, es decir, que las nuevas instrucciones se van apilando en las anteriores generando una ISA muy extensa y costosa. A diferencia de la arquitectura RISC-V que es modular, lo cual significa que el conjunto de instrucciones es fijo y cuando se quiere agregar una nueva instrucción usan módulos que vienen como extensiones opcionales estándar que el hardware puede incorporar de acuerdo a las necesidades de cada aplicación.

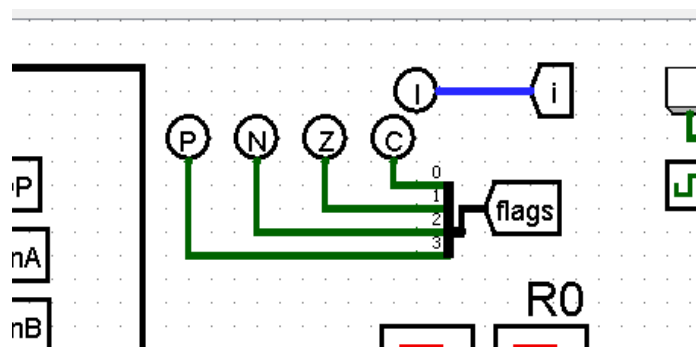
*¿Qué ventajas puede tener esto?*

- Las ventajas vienen dadas de que el usuario puede elegir que módulos agregar (como RV32M, RV32F, RV32D agrega punto flotante de precisión doble, etc.) a las instrucciones base obligatorias (RV32I) en base a qué función necesita y así se mantiene una ISA muy pequeña y de bajo consumo energético.

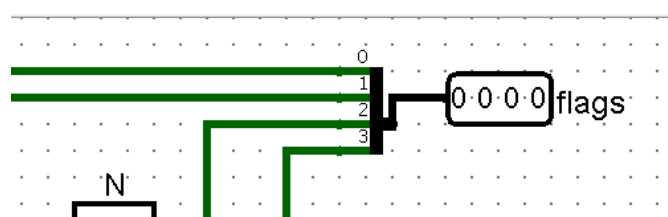
#### Ejercicio 4

*Queremos 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 Orga1SmallII para implementarlo? Descríbalos en detalle. ¿Cómo resolvería un salto condicional (JP) por paridad? Suponga que tiene espacio libre para las instrucciones en la memoria de la unidad de control y que puede agregar tantas señales a la misma como hagan falta*

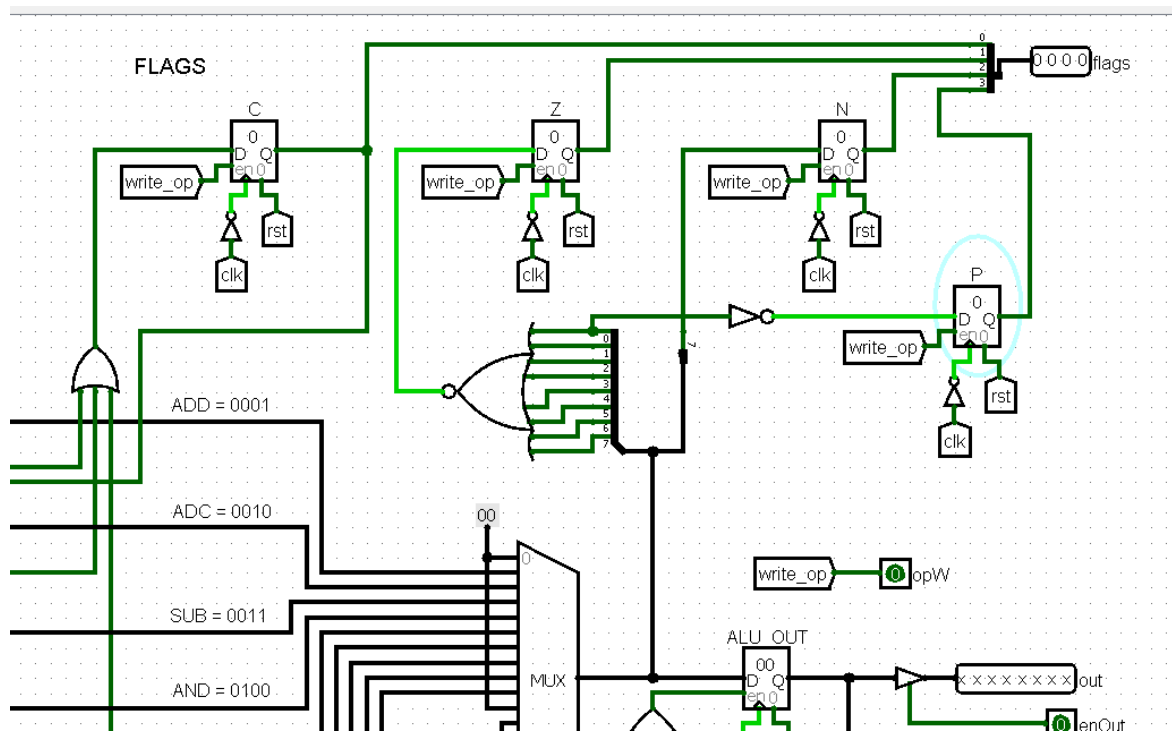
- Primero, en la microOrgaSmallII hay que cambiar el Splitter de las flags de 3 bits por uno de 4 bits, de manera tal que, en la posición 3 reciba la señal de paridad.



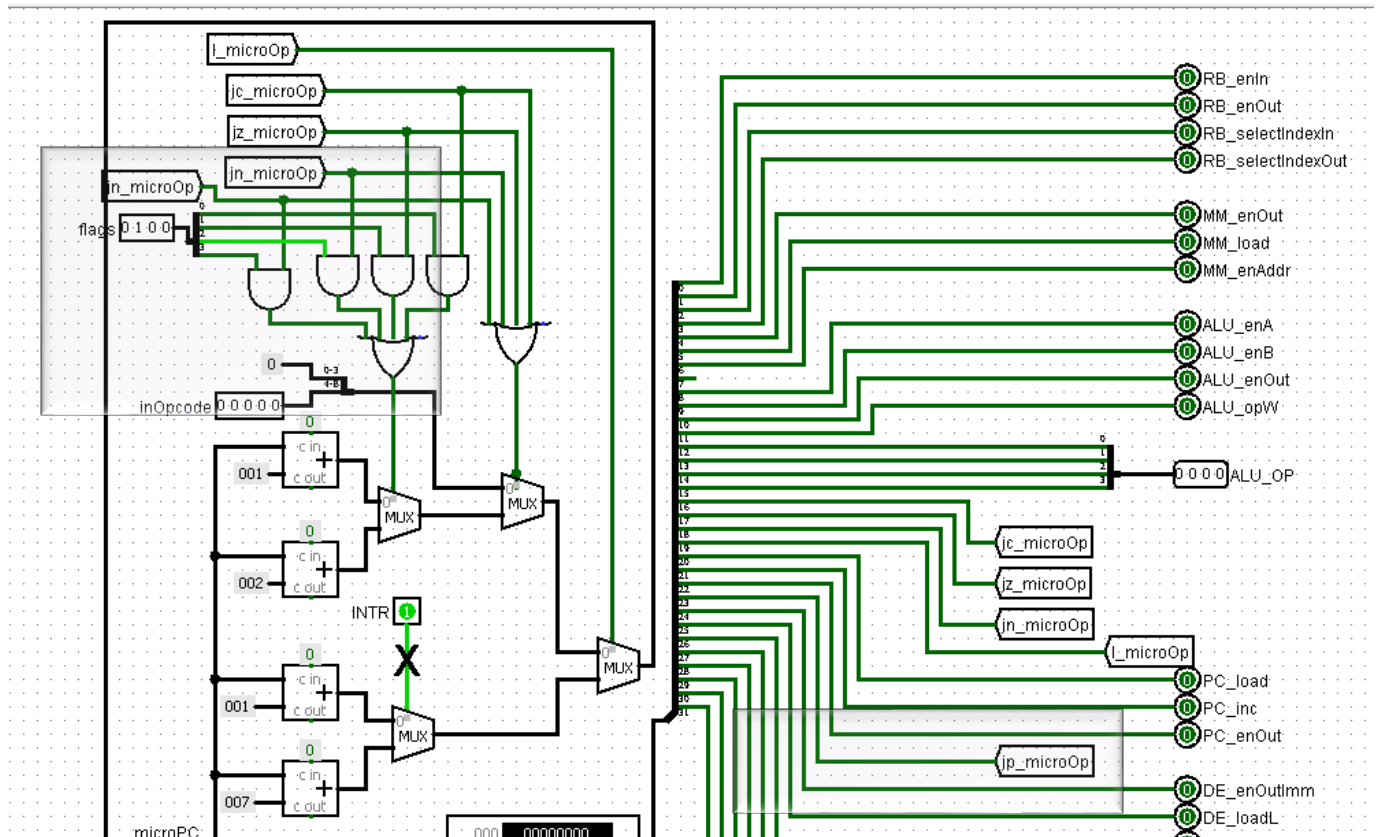
- Luego, en la ALU, cambiamos la salida de las flags a 4 bits, dejando la posición 3 para flag P.




- Notemos que el output es par cuando el primer bit es 0, e impar cuando es 1 (pues el primer bit es el único que suma 1 al momento de pasar el número en binario a base 10) Entonces, teniendo en cuenta esto, construimos la frag P:
  - Al bit 0 del Splitter de la salida de la ALU, lo comparamos con 0 usando la compuerta NOT, es decir, cuando el bit cero sea 0 (es par) entonces la flag se activa, cuando el bit cero es 1 (es impar) entonces la flag no se activa. Este problema de paridad, lo resolvemos antes de pasarlo por un Register, el cual necesitamos para que la flag se active en el momento indicado.



- Y por ultimo, cambiamos un poco la unidad de Control:  
Sabemos que el microPC se sobrescribe con la señal de load microOp en conjunto con 3 selectores de multiplexores. Cambiamos las flags a 4 bits. Y resolvemos el salto condicional por paridad.  
Tenemos que agregar una señal llamada jp\_MicroOp que nos va a indicar si hubo un salto por paridad, si este está habilitado y el bit 3 de las flags, que es el que tiene la flag P, también está habilitado entonces se fija si hay alguna otra flag habilitada y en base a eso el multiplexor indica si se incrementa el microPC en 1 o 2. De ahí, pasa a otro multiplexor que indica si se sobrescribe el microPC por una nueva instrucción o flags, y de ahí a otro multiplexor que dada la señal de interrupciones indica si se sobrescribe por una interrupción.



el de la derecha indicando si se sobrescribe por una nueva instrucción o flags; y el de la izquierda indicando si se incrementa el micro PC en 1 (flag habilitado) o 2 (flag en cuestión deshabilitado).

(  → cambios )

Agregamos el microprograma del JP:

JP

```
JP_microOp load_microOp
reset_microOp
DE_enOutImm PC_load
reset_microOp
```

(En la primera línea se fija si el JP está encendido o no y load\_microOp carga el valor de la micro operación. Entonces, si sí está encendido el JP\_microOp no lee la segunda línea, y pasa directamente a la tercera donde se habilita la salida de un valor inmediato del decode y se carga en el PC, luego se resetea el microOp)

## Ejercicio 5

Microprograma:

LU:467/21,

Alumna: Valeria Lucia Arratia Guillen

RB\_enOut mem\_op=001 RB\_selectIndexOut=0

RB\_enOut mem\_op=010 RB\_selectIndexOut=1

Mem\_op=111

Reset\_microOp