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

IIC2343 – Arquitectura de Computadores

Saltos y Subrutinas

Profesor: Hans Löbel

```java
public static void promedio()
{
    int[] arreglo = new int[]{6,4,2,3,5};
    int n = 5;
    int i = 0;
    float promedio = 0;

    while(i < n)
    {
        promedio += arreglo[i];
        i++;
    }
    promedio /= n;
    System.out.println(promedio);
}
```

¿Cómo podríamos implementar un while?

```
while (i > 0)
{
        BLA BLA BLA
        i--;
}
```

```
while: CMP A,0
       JLE end
       BLA BLA BLA
       SUB A,1
       JMP while
end:
```

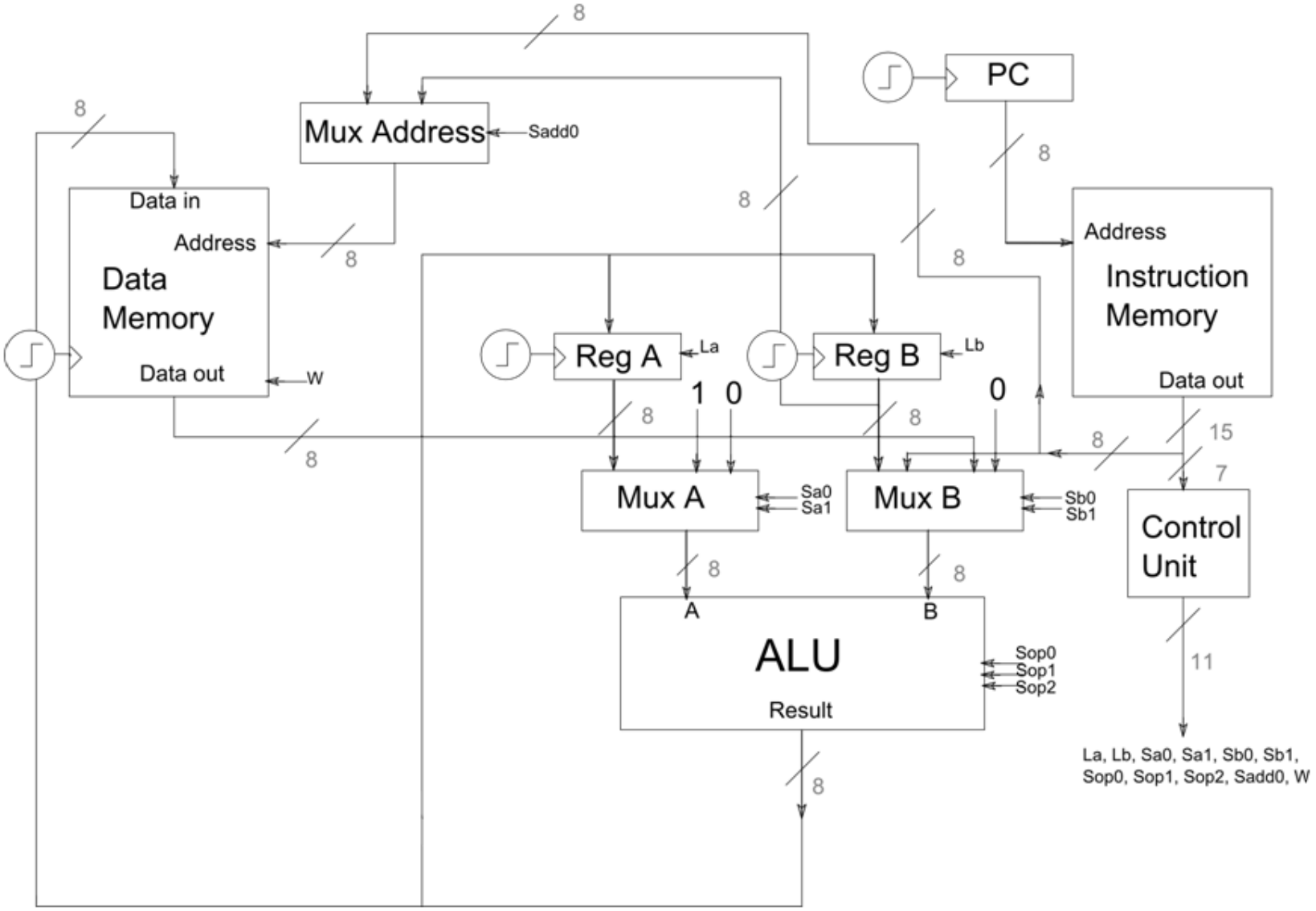
¿Cómo podríamos implementar un if?

```
if (x == 0)
{
        bla bla
}
else
{
        ble ble
}
```

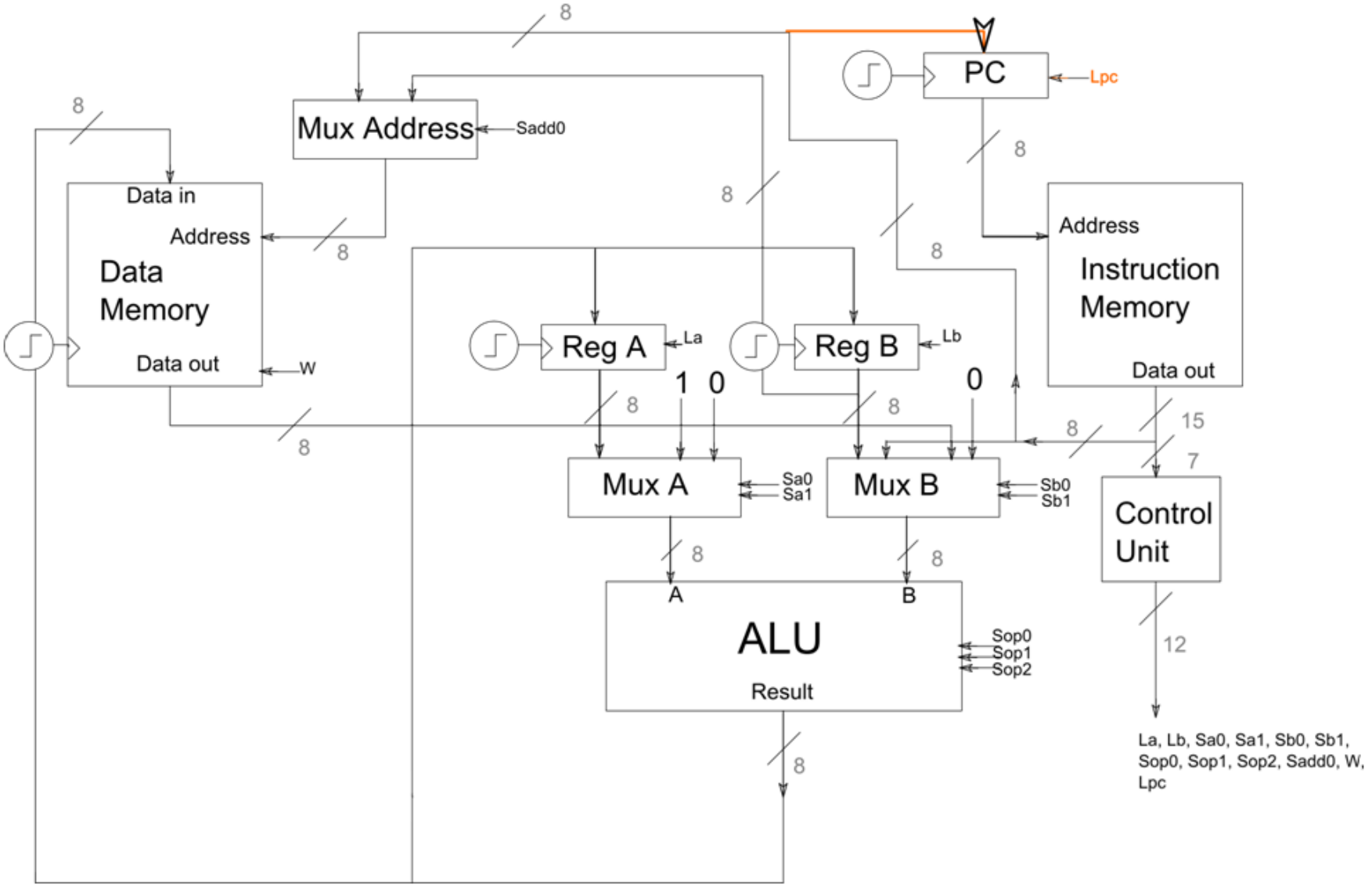
```
        CMP A,0
        JNE else
        bla bla
        JMP end
else:   ble ble
end:
```

Necesitamos saltos incondicionales y condicionales basados en una comparación

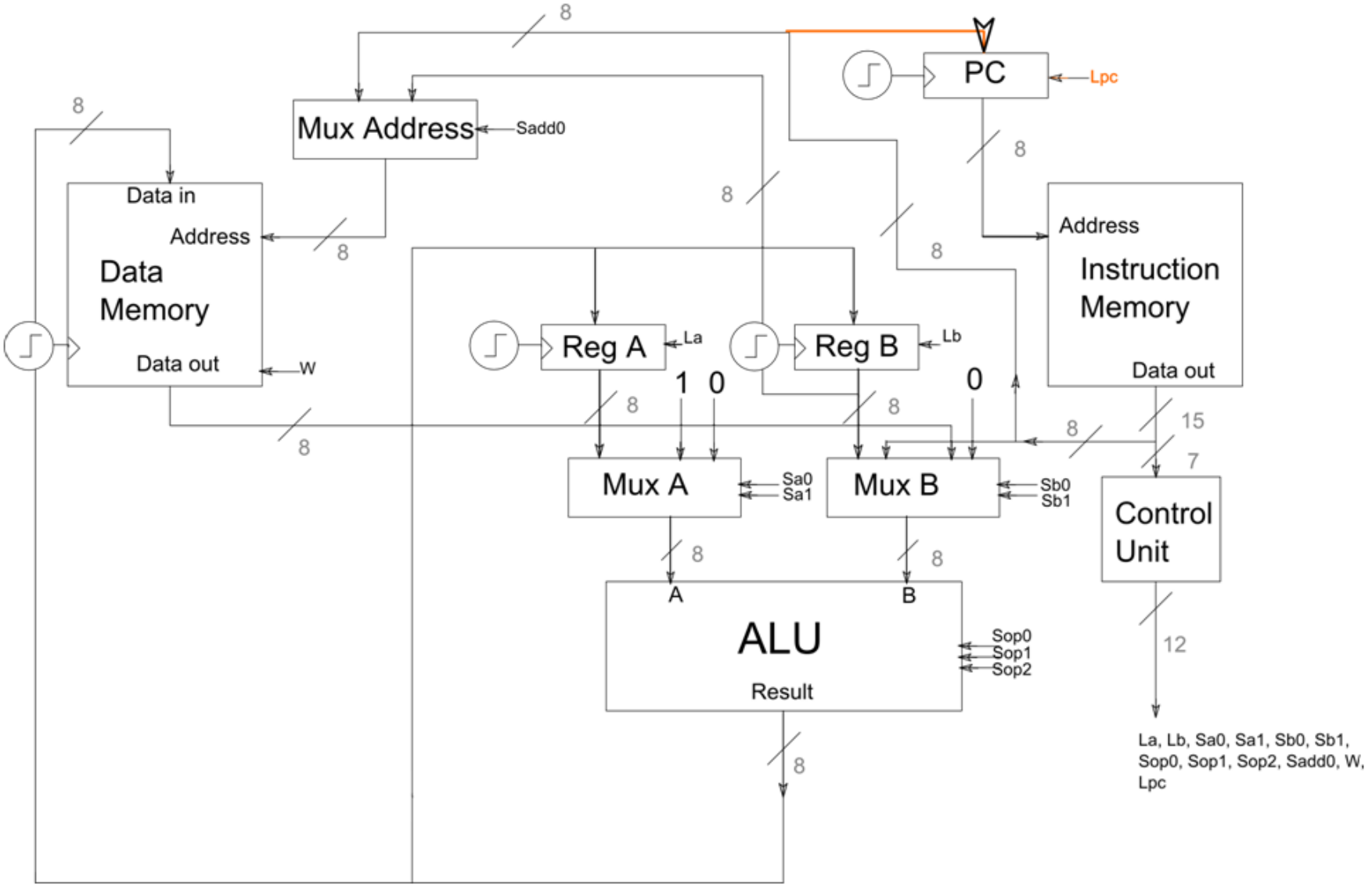
¿Qué debemos agregar para tener soporte en HW para saltos incondicionales?



¿Qué debemos agregar para tener soporte en HW para saltos incondicionales?



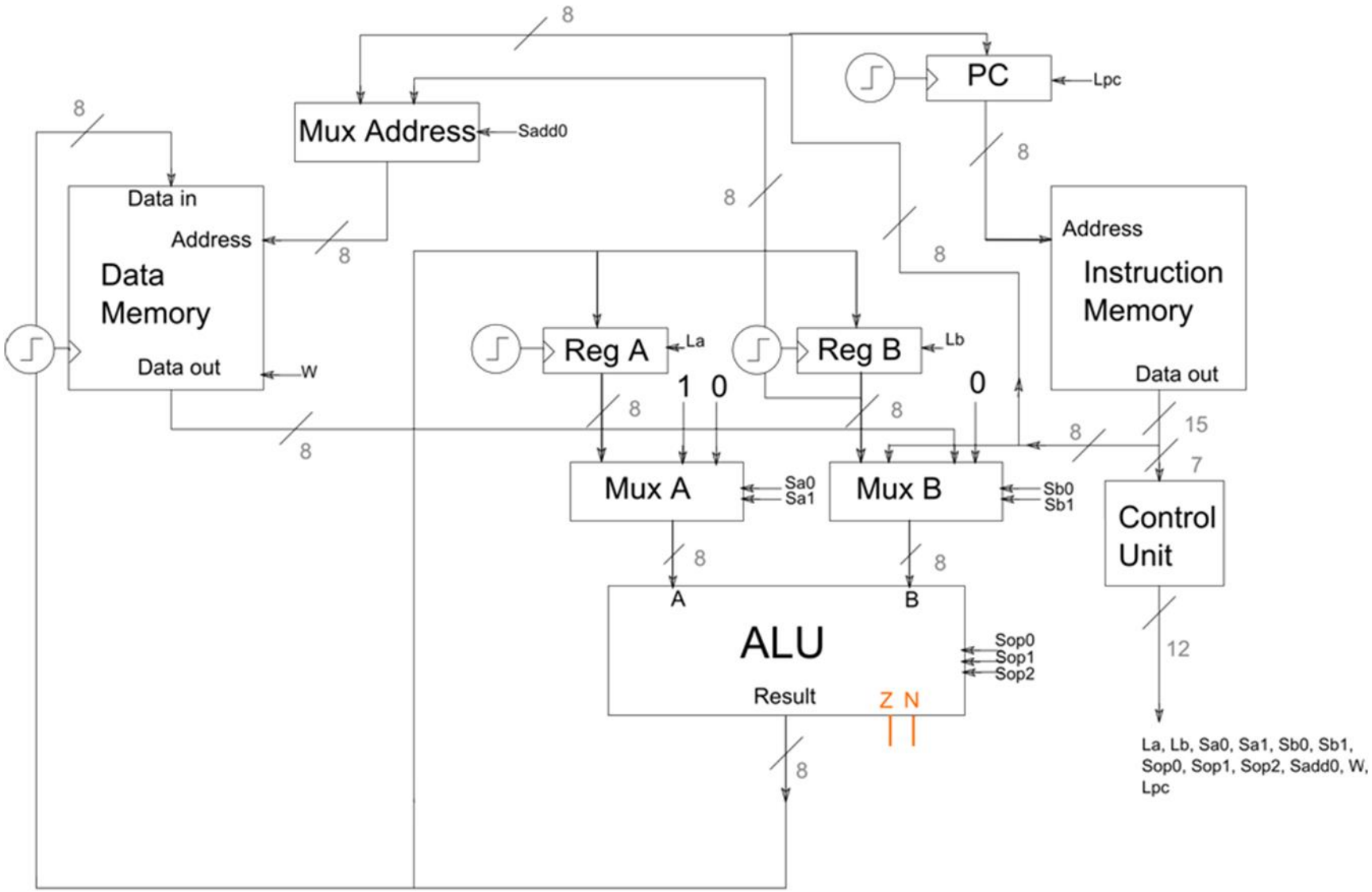
¿Y para saltos condicionales?



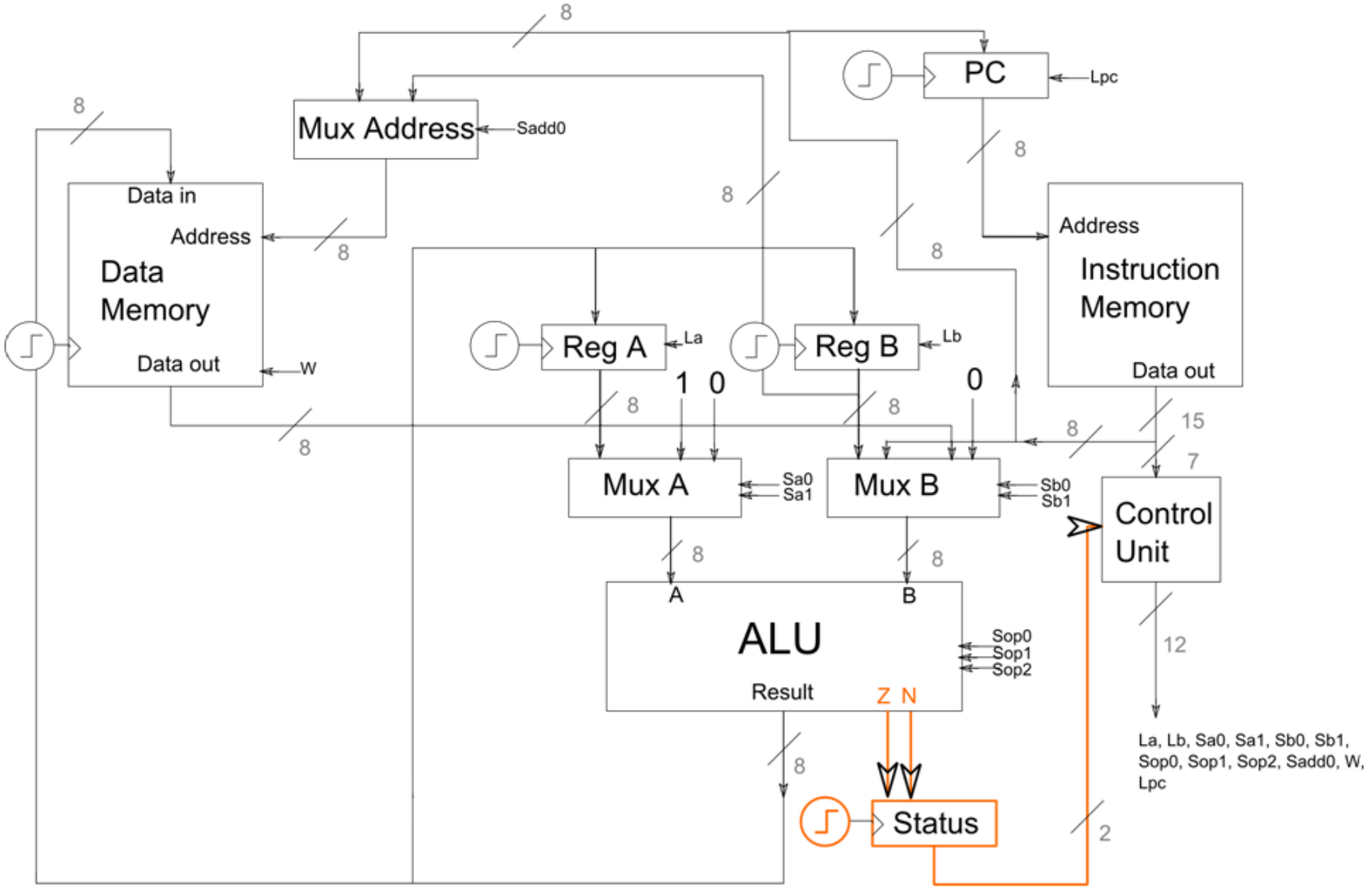
Lo primero es una instrucción de comparación, que no altere el contenido de los registros, ni de la memoria.

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CMP	A,B	A-B		
	A,Lit	A-Lit		CMP A,0

Lo segundo es extraer el estado de la ALU (flags, condition codes)



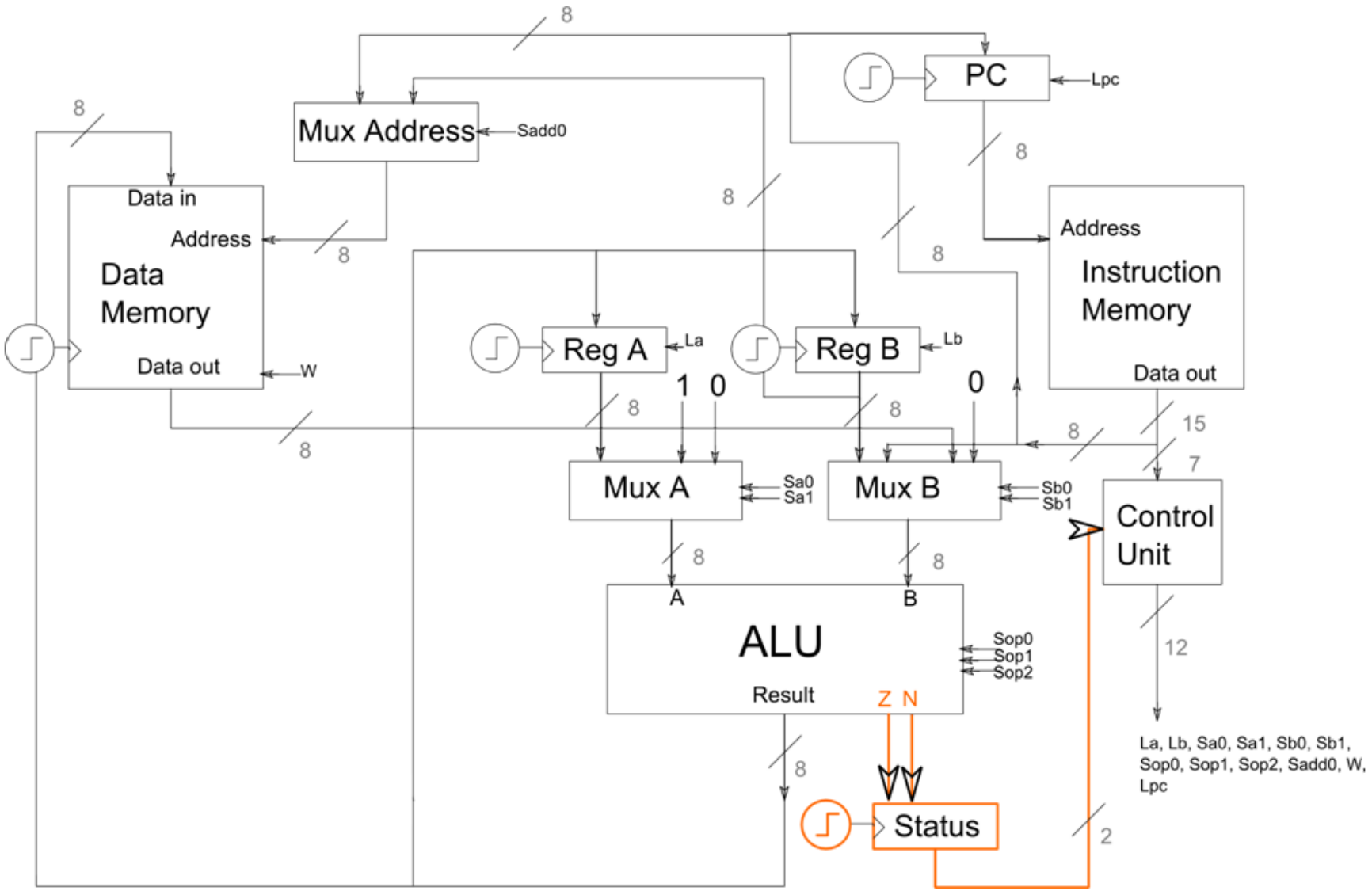
Agregamos el registro Status, que permite almacenar los condition codes para entregarlos a la CU en la siguiente instrucción (salto)



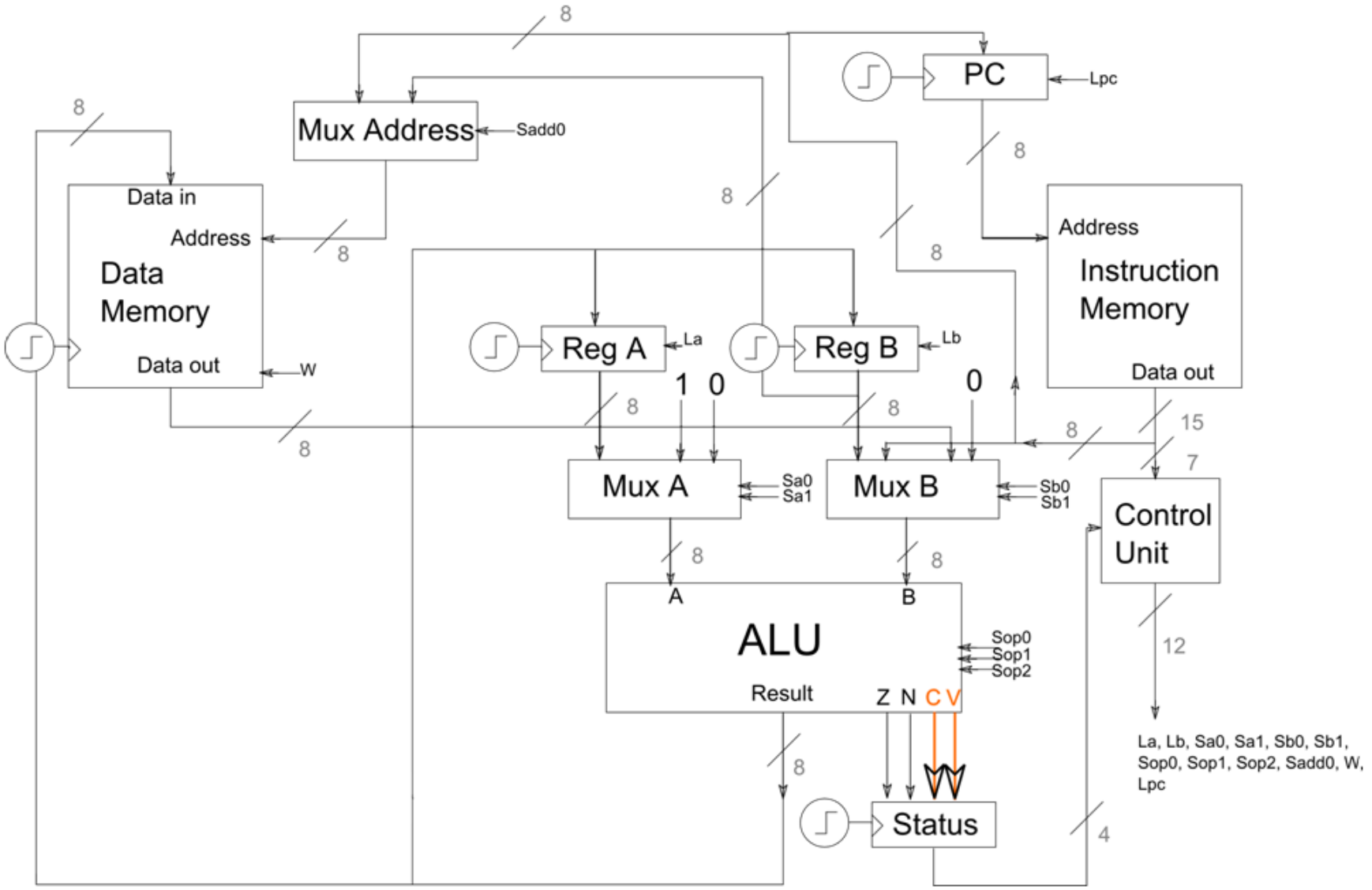
Finalmente agregamos al assembly las instrucciones de comparación necesarias para los distintos casos

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CMP	A,B	A-B		
	A,(B)	A-Mem[B]		
	A,Lit	A-Lit		CMP A,0
	A,(Dir)	A-Mem[Dir]		CMP A,(label)
JEQ	Dir	PC = Dir	Z=1	JEQ label
JNE	Dir	PC = Dir	Z=0	JNE label
JGT	Dir	PC = Dir	N=0 y Z=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

¿Existen más situaciones que nos gustaría controlar?



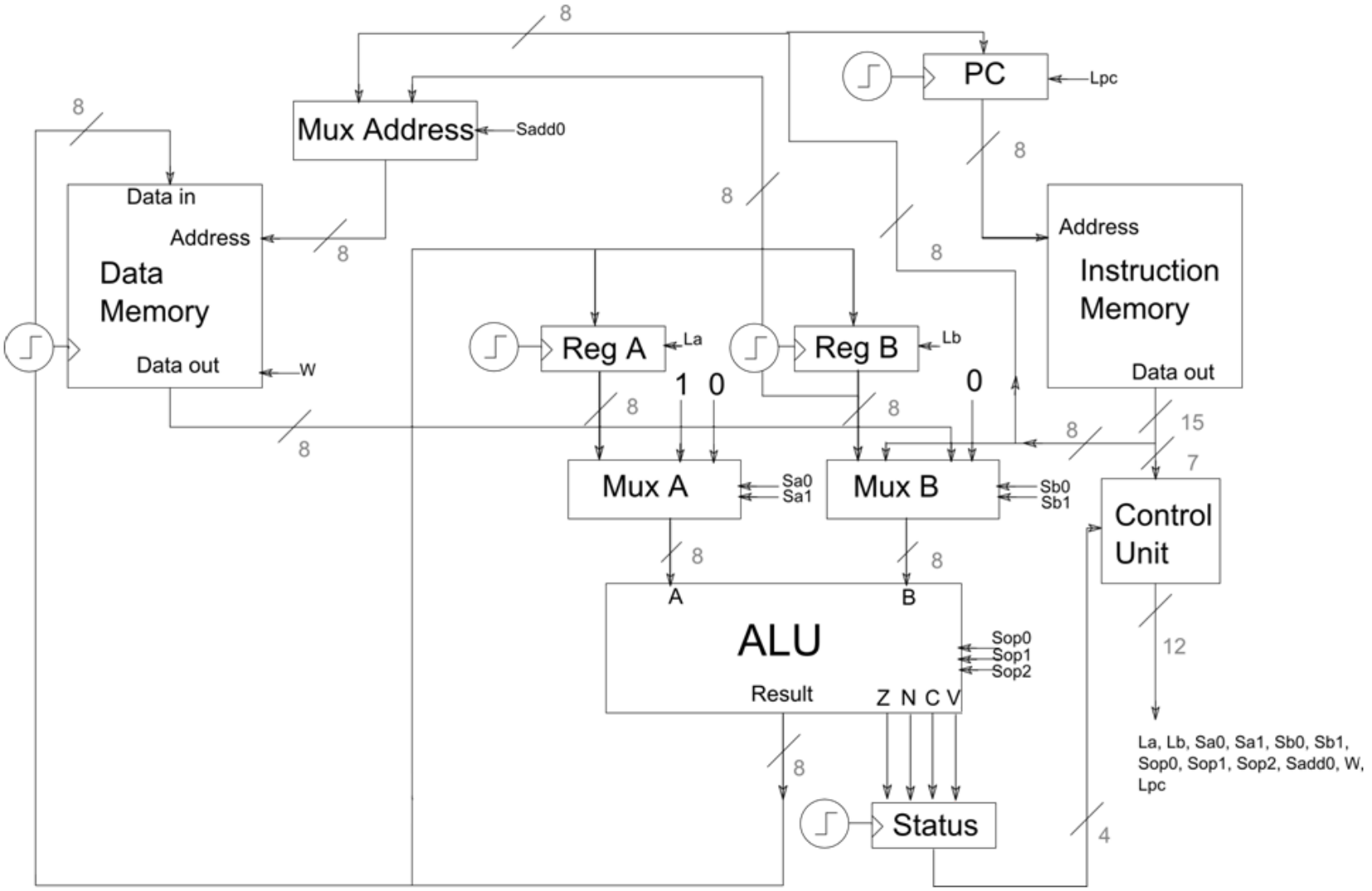
Agregamos bits para carry y overflow al registro Status



Operación	A	B	Resultado	Ejemplo (1 byte)
$A + B$	≥ 0	≥ 0	< 0	$127 + 4 = -125$
$A + B$	< 0	< 0	≥ 0	$-127 + -4 = 125$
$A - B$	≥ 0	< 0	< 0	$127 - -4 = -125$
$A - B$	< 0	≥ 0	≥ 0	$-127 - 4 = 125$

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
JCR	Dir	PC = Dir	C=1	JCR label
JOV	Dir	PC = Dir	V=1	JOV label

Si bien nuestro computador puede ahora hacer de todo, no tenemos aún soporte para modularizar código (funciones/subrutinas)



Agreguemos soporte para subrutinas primero desde el punto de vista del assembly

¿Qué elementos son necesarios para implementar subrutinas?

1. Parámetros de entrada
2. Valor de retorno
3. Llamada a la subrutina (salto y retorno)

Parámetros implican almacenamiento

- Necesitamos almacenar los parámetros en algún lugar que la subrutina pueda acceder
- Tenemos 2 opciones: registros y memoria
- Registros requieren especificar cual(es) se van a usar y son rápidos, pero limitan la cantidad de parámetros
- Memoria requiere especificar las direcciones, pero no pone limites a la cantidad de parámetros
- Tenemos todo el HW necesario para esto

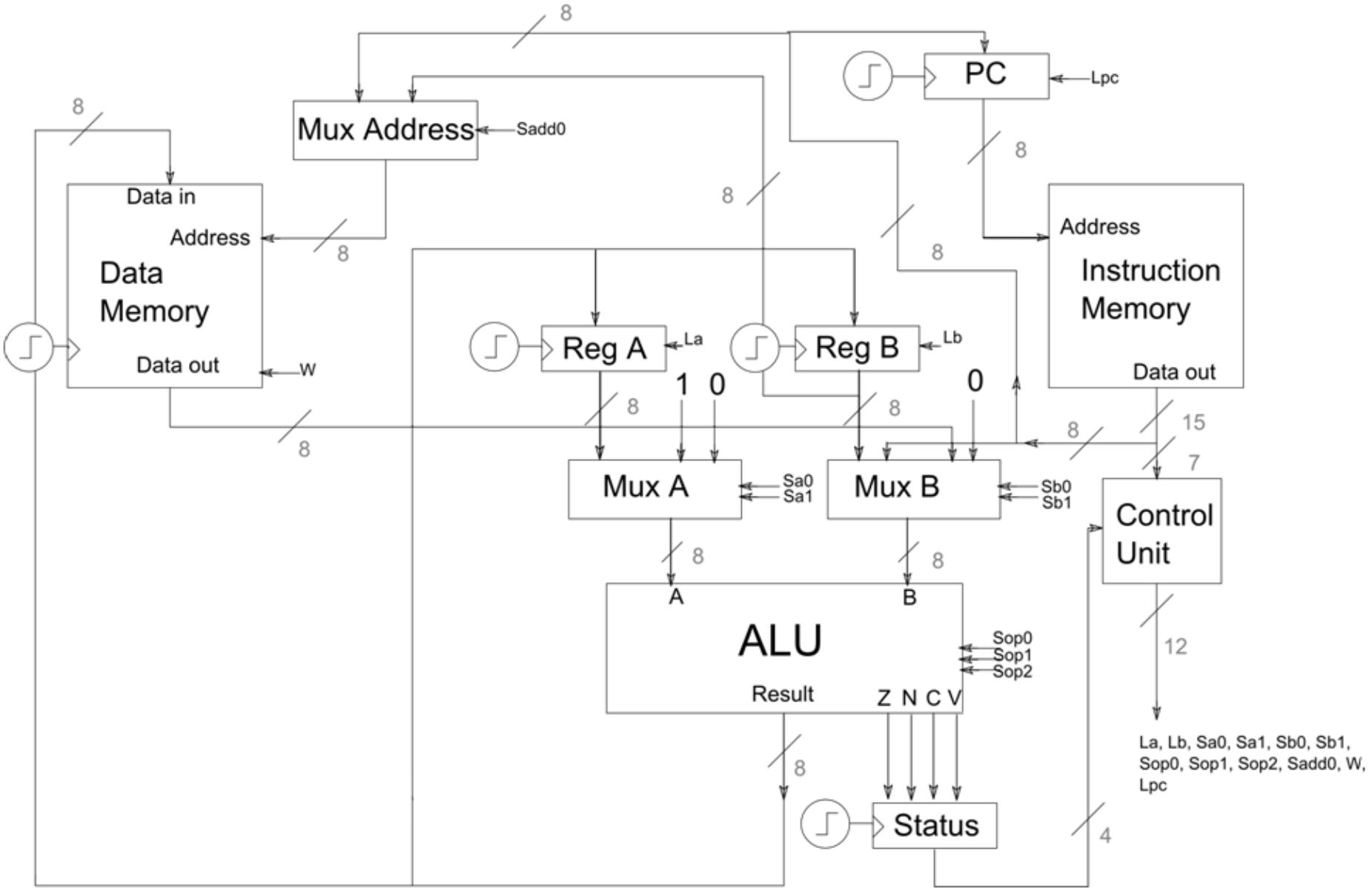
Soporte para valor de retorno es similar al de los parámetros

- Nuevamente tenemos 2 opciones: registros y memoria
- Registros requieren especificar cual(es) se van a usar, pero ahora no hay limitación de espacio, ya que es sólo un valor de retorno
- Memoria es igual al caso de los parámetros de entrada
- Nuevamente tenemos todo el HW necesario

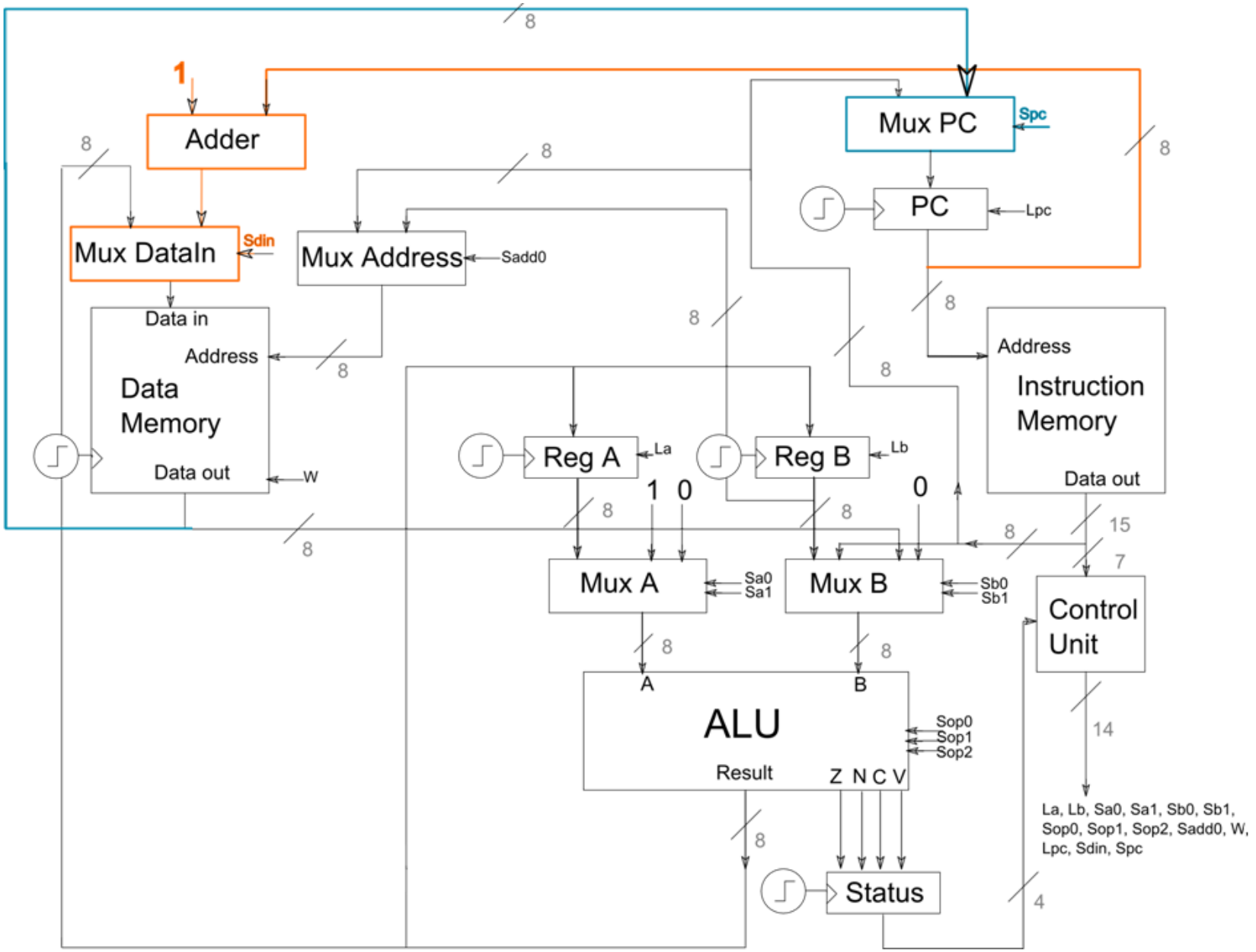
Llamada y retorno requieren más cuidado

- Llamada puede hacerse mediante un salto a la dirección de memoria de la subrutina (label)
- ¿Podemos hacer lo mismo con el retorno?
- Saltos no bastan, ya que no sabemos donde volver
- Necesitamos almacenar el valor del registro PC

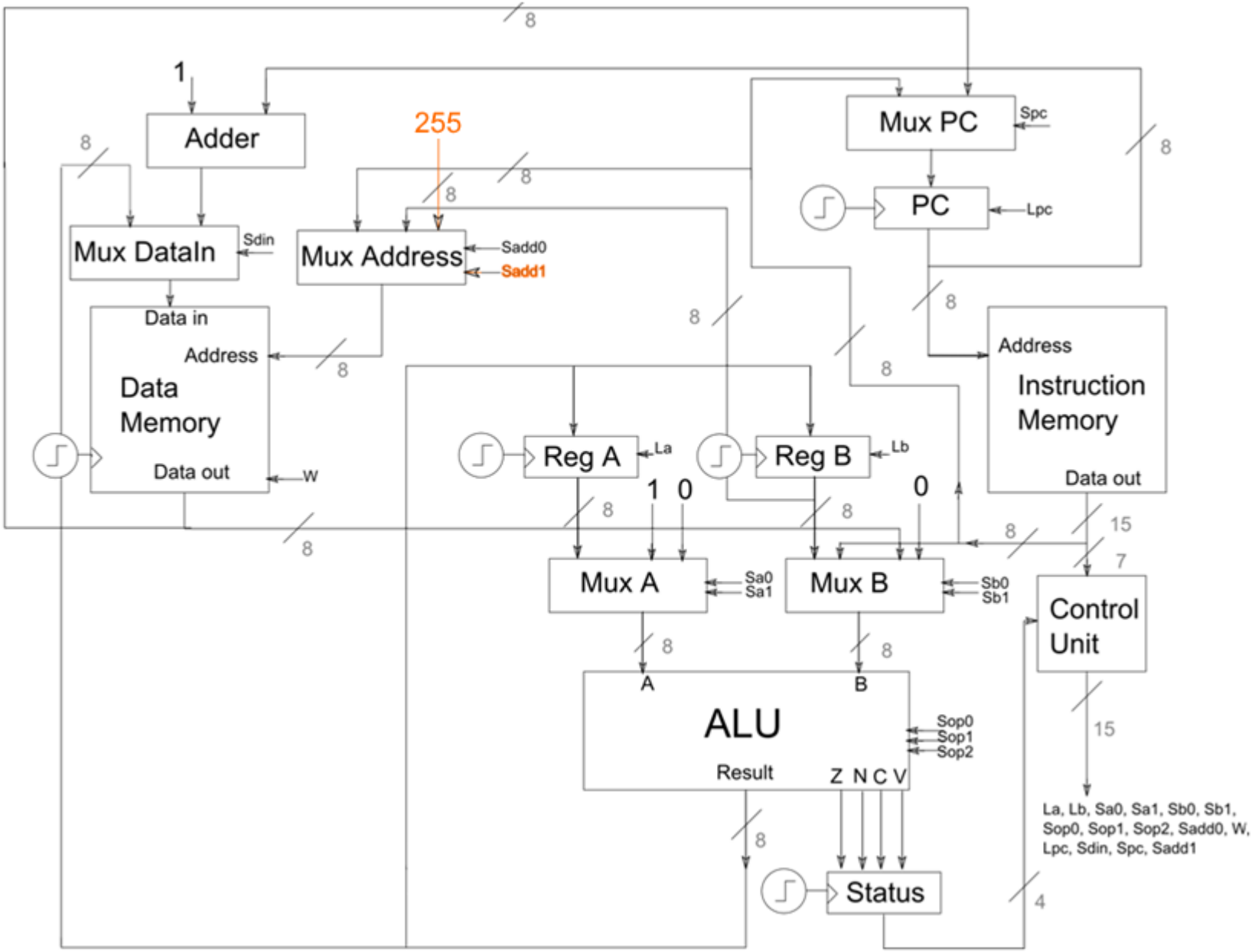
¿Tenemos soporte en HW para almacenar PC?



Agregamos conexión entre PC y memoria



Literal fijo (255) indica donde se almacena PC+1



Agregamos dos nuevas instrucciones al assembly del computador

1. CALL dir: almacena PC+1 en Mem[255] y salta a la dirección dir de la memoria de instrucciones

2. RET: extrae el valor de Mem[255] y lo guarda en PC, lo que conduce a un salto a la dirección siguiente al llamado de la subrutina. Debe ejecutarse siempre al final de esta.

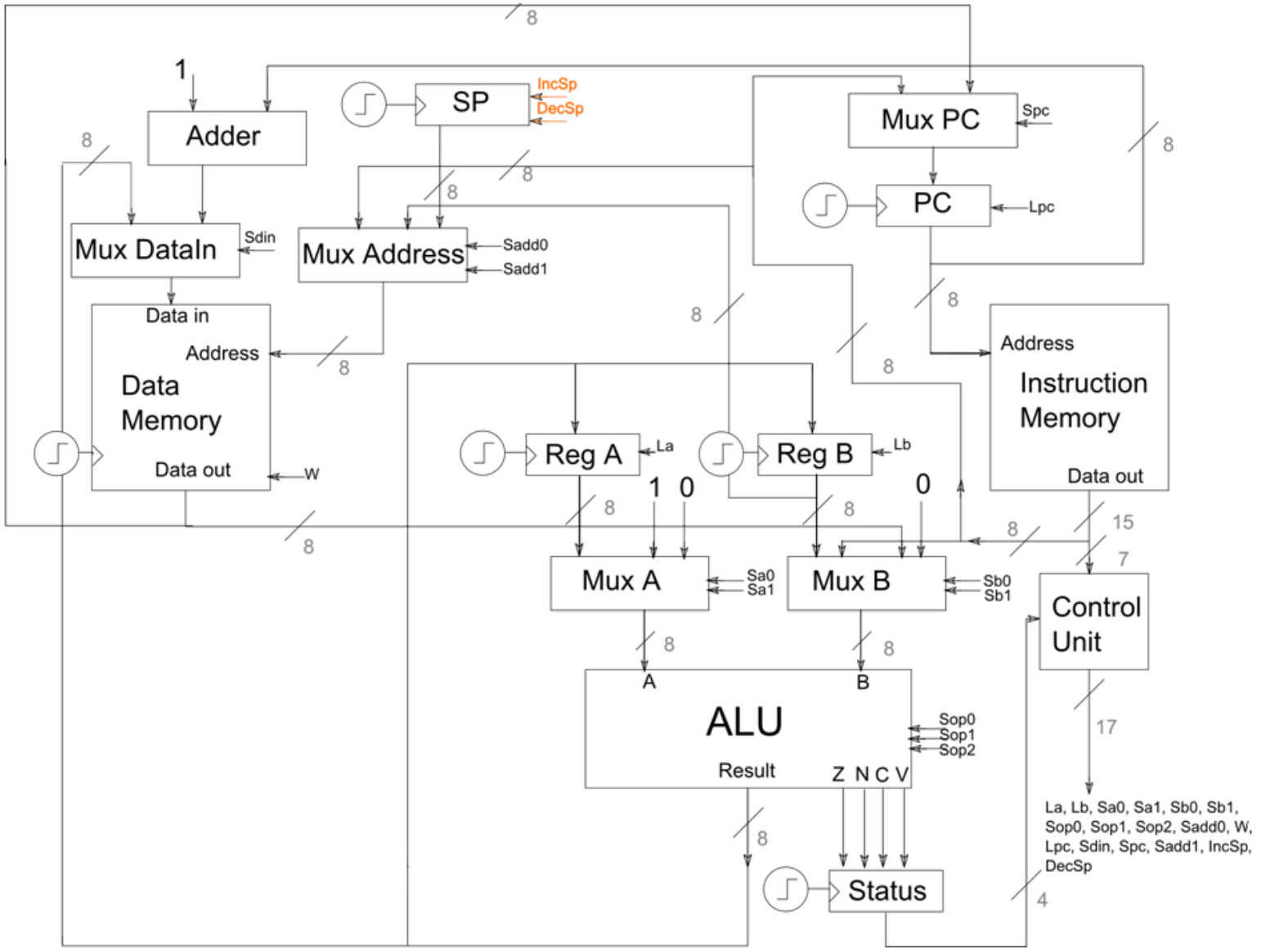
¿Qué pasa en este caso?

```
        JMP main
func1:  MOV A, (var1)
        MOV B, (var2)
        ADD A, B
        MOV (var1), A
        CALL func2
        RET

func2:  MOV A, var1
        MOV B, var2
        ADD A, B
        RET

main:
        MOV A, 5
        MOV B, 2
        MOV (var1), A
        MOV (var2), B
        CALL func1
        …
        …
```

Agregamos un registro con incremento y decremento (SP o stack pointer)



¿Qué pasa en este caso con A y B?

```
main:   …
        …
        MOV A, 5
        MOV B, 3
        CALL func
        ADD A, B
        …
        …

func:   INC B
        ADD A, B
        RET
```

Stack de uso general soluciona estos problemas

- Agregamos las instrucciones PUSH y POP
- PUSH Reg almacena en Mem[SP] el valor almacenado en el registro Reg y luego decrementa SP
- POP Reg primero incrementa SP y luego escribe en Reg el valor almacenado actualmente en Mem[SP]

Ahora no tenemos problemas

```
main:   …
        …
        MOV A, 5
        MOV B, 3
        PUSH A
        PUSH B
        CALL func
        POP B
        POP A
        ADD A, B
        …
        …

func:   INC B
        ADD A, B
        RET
```

En POP se invierte el orden de PUSH

Resumamos que pasa al llamar y retornar de una subrutina

Al llamar a una subrutina, debemos:

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

¿Cuánto ciclos del clock necesitamos para ejecutar estas 3 acciones?

Respuesta: 1

Resumamos que pasa al llamar y retornar de una subrutina

Al retornar de una subrutina, debemos:

1. Incrementar en 1 SP
2. Cargar en PC el valor de memoria apuntado por SP incrementado.

¿Cuánto ciclos del clock necesitamos para ejecutar estas 2 acciones?

Respuesta: 2

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CALL	Dir	Mem[SP] = PC + 1, SP- -, PC = Dir		CALL func
RET		SP++		-
		PC = Mem[SP]		-
PUSH	A	Mem[SP] = A, SP- -		-
PUSH	B	Mem[SP] = B, SP- -		-
POP	A	SP++		-
		A = Mem[SP]		-
POP	B	SP++		-
		B = Mem[SP]		-

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

IIC2343 – Arquitectura de Computadores

Saltos y Subrutinas

Profesor: Hans Löbel