Práctica de diseño de un procesador monociclo

El objetivo de esta práctica de la asignatura de Tecnología de Computadores es la implementación de un procesador RISC-V reducido (nombre del procesador **TG00RISCV**).

Las características hardware principales del procesador TG00RISCV son:

- Palabra de 32 bits.
- Banco de 32 registros de propósito general de 32 bits. **El registro x0 será de solo lectura y contendrá el valor 0**.
- Unidad aritmético-lógica (ALU) de 32 bits con las siguientes características:
  - Suma y resta de operandos enteros representados en convenio complemento a 2 (C2)
  - Operaciones lógicas AND, OR y NOR.
  - Operación de comparación “menor que”.
  - Un flag de estado Z: indica (con un 1) que el resultado de la ALU es igual a 0.
- Memoria de instrucciones:
  - Bus de direcciones y de datos de 32 bits
  - Direccionable a nivel de byte.
- Memoria de datos:
  - Bus de direcciones y de datos de 32 bits
  - Direccionable a nivel de byte.

Para el diseño del procesador se proporcionan ya implementados los siguientes elementos: una ALU, un banco de 32 registros de 32 bits, las memorias de instrucciones y de datos, un módulo de extensiones de inmediatos y un módulo de control de la ALU.

En las figuras y tablas del Anexo I, al final de este documento, se muestra la forma, interfaz, funcionamiento y ejemplo de uso de los elementos proporcionados.

En el Anexo II se muestra un resumen de las instrucciones que deben implementar el procesador TG00RISCV implementado.

Tareas a desarrollar por el alumno

1. Diseño e implementación de las instrucciones JAL y HALT

Empezarás la implementación del procesador consiguiendo que sea capaz de ejecutar estas instrucciones. Debes utilizar como bloques básicos de diseño: ALU, banco de registros, registros, sumador/restador, biestables, multiplexores, codificadores, decodificadores y puertas lógicas (todos estos elementos con el ancho de bits que necesites).

Debes bajarte de la plataforma de enseñanza virtual un prototipo de proyecto que irás modificando para diseñar tu procesador. El nombre del prototipo es TG00RISCV.ZIP. Dentro de dicho fichero comprimido encontrarás el fichero con el circuito base TG00RISCV.circ. **Dicho nombre deberá ser cambiado antes de empezar a trabajar con él**. Deberás cambiar los 4 primeros caracteres (**TG00**) por los que correspondan:

- **T**: Titulación del alumno (**I** Ing. Informática, **S** Ing. del Software, **C** Ing. de Computadores)
- **G**: Grupo (**A, B, …**)
- **00**: Número de equipo del alumno con dos dígitos (**01, 02, …**)

Así por ejemplo el equipo 13 del grupo A de la ingeniería informática, deberá trabajar sobre el proyecto con nombre: **IA13RISCV.circ.**

a. Implementación del hardware para el fetching de las instrucciones:

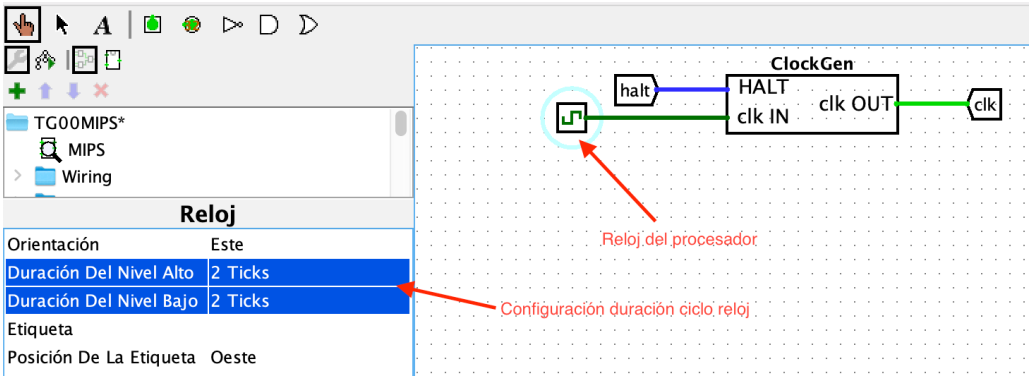
Debes implementar en tu circuito el hardware necesario para la lectura secuencial de las instrucciones de la memoria de instrucciones. Para ello tienes que usar le registro PC que hace la función de contador de programa: debe direccionar la memoria de instrucciones y actualizarse en cada flanco activo del reloj con la dirección de la siguiente instrucción (PC+4), añadiendo para ello el hardware adicional que sea necesario. **Recuerda que para que un registro cargue un valor necesita recibir un flanco por su señal de sincronismo (clk).**

Comprobación del funcionamiento: Para comprobar el funcionamiento del hardware de lectura de las instrucciones, debes introducir un programa en la memoria de instrucción y comprobar que se van leyendo en secuencia, una a una, las instrucciones almacenadas en la memoria, con cada flanco activo del reloj.

Introduce el siguiente código en la memoria de instrucciones (edita el fichero **instmem.txt** con cualquier editor de texto introduciendo el contenido después de la línea "**v2.0 raw**" con la que empieza dicho fichero):

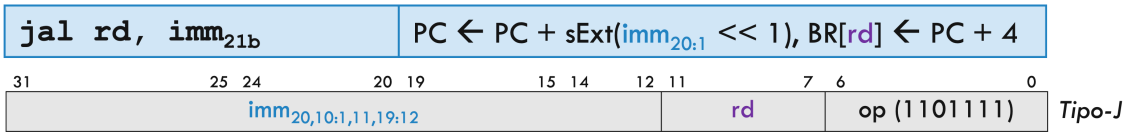
Dir.(hex)	Instrucciones	Código binario (hex) (contenido en instmem.txt)
0	nop	00000000
4	nop	00000004
8	jal x0, cinco	00c0006f
C	tres: nop	0000000C
10	jal x0, siete	00c0006f
14	cinco: nop	00000014
18	jal x0, tres	ff5ff06f
1C	siete: nop	0000001C
20	halt	ffffffff

Para poder simular el procesador y poder ver su correcto funcionamiento, antes es necesario configurar la correcta duración del ciclo de reloj. **Ya que el retardo de lectura de la memoria es de 4 ticks, asegúrate que el ciclo del reloj del procesador tenga al menos 4 ticks entre dos flancos activos para que de tiempo a leer una instrucción de memoria entre dos flancos activos de la señal de reloj.** En la siguiente imagen se ve como el reloj está configurado con 2 ticks para el nivel alto de la señal y 2 ticks para el nivel bajo, lo que suma un total de 4 ticks en total para el ciclo de reloj.



Comprueba que se van leyendo las instrucciones en la misma secuencia que están almacenadas en memoria (aparecen a la salida de la memoria de instrucción). Para ello tienes que conmutar el reloj del procesador (*Simular→Commutar Reloj o ctrl-t*), de forma que, en cada flanco activo de la señal del reloj, el contenido de PC habrá tenido que incrementarse en 4 unidades y por la salida de la memoria de instrucciones deben desfilar las instrucciones que has introducido en la misma (en el fichero instmem.txt). Para inspeccionar el valor de PC o el de la salida de la memoria de instrucciones basta con pulsar con la herramienta  sobre el bus de salida de cada uno de ellos o utilizar la herramienta sonda (en librería Wiring).

b. Implementación de la instrucción JAL



Implementa el hardware necesario para que, una vez leída la instrucción de memoria, si es una instrucción JAL (**op 1101111**) cargue en PC, en vez de PC+4, la dirección de salto de 32 bits construida sumándole a PC la extensión de signo del operando inmediato (imm).

Recuerda que, si la instrucción no es una JAL, PC se debe cargar con PC+4 como hacía en el apartado anterior.

En la unidad de datos deberás crear el camino HW que permita construir la dirección de destino y llevarla a la entrada de datos de PC sin que se deje de poder cargar en PC lo que se le había conectado previamente. Para construir la dirección de destino vas a necesitar el **circuito extensor** (consulta ANEXO I) para realizar una extensión tipo-J (ya que este es el formato de la instrucción JAL) y un **sumador** para sumarla a PC (no se puede utilizar la ALU para hacer esta suma).

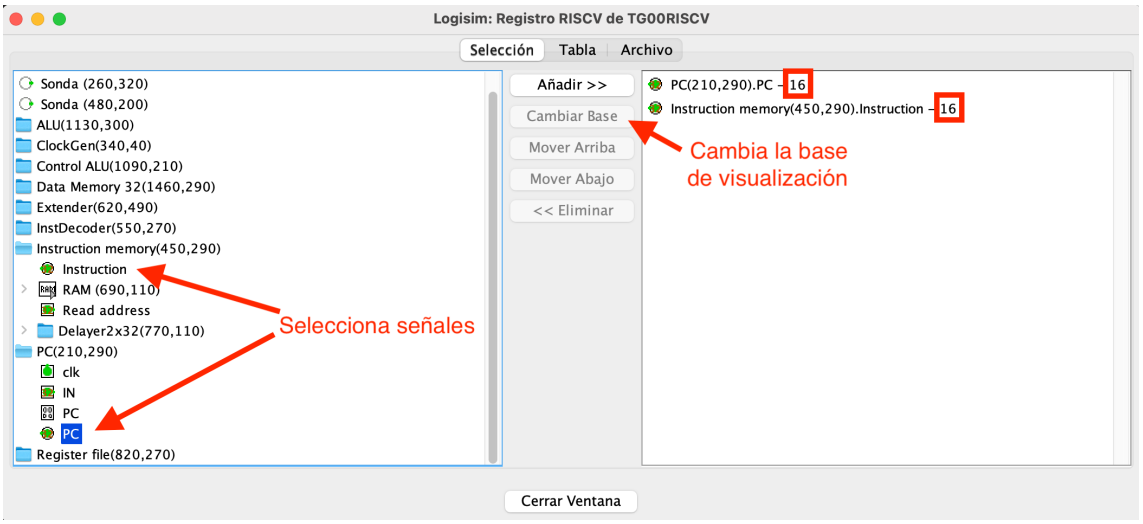
En la unidad de control deberás reconocer (decodificar) que es una instrucción JAL y activar el/los punto/s de control que activan el camino de datos HW que lleva la dirección de salto a PC (en vez de PC+4).

Comprobación del funcionamiento: Para comprobar el funcionamiento de la instrucción JAL introduce el mismo programa anterior en la memoria de instrucciones:

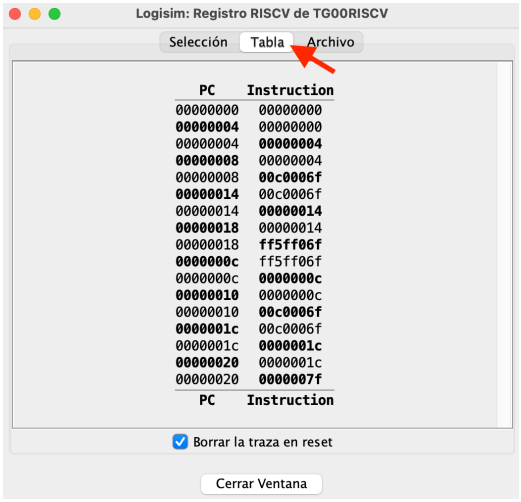
Dir.(hex)	Instrucciones	Código binario (hex) (contenido instmem.txt)
0	nop	00000000
4	nop	00000004
8	jal x0, cinco	00c0006f
C	tres: nop	0000000C
10	jal x0, siete	00c0006f
14	cinco: nop	00000014
18	jal x0, tres	ff5ff06f
1C	siete: nop	0000001C
20	halt	ffffffff

Conmutando el reloj varias veces, debes comprobar que las instrucciones JAL se están ejecutando (ahora PC no avanzará secuencialmente y por tanto las instrucciones no se leerán en orden secuencial). Para hacer esta comprobación, visualiza en la traza de simulación el valor de PC y de la salida de la memoria de instrucción (para ver la instrucción leida).

En Logisim, en el menú *Simular* selecciona *Registro*. Luego en la pestaña *Selección*, selecciona las señales PC y la salida de la memoria de instrucción, y cambia la base a 16 (para mostrar los valores en hexadecimal):



Luego selecciona la pestaña Tabla y conmuta la señal de reloj (ctrl-t) para visualizar la traza de la simulación.

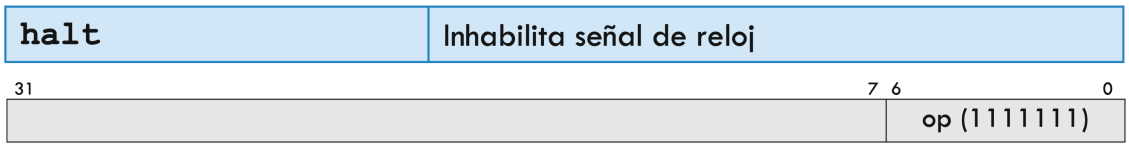


La traza (traza = secuencia de valores que van tomando los elementos) para PC y la salida de la memoria de instrucción deberían ser estos:

PC	Salida mem instrucción
00000000	00000000
00000004	00000000
00000008	00c0006f
00000014	00000014
00000018	ff5ff06f
0000000C	0000000c
00000010	00c0006f
0000001C	0000001C
00000020	ffffffff

Fíjate que la secuencia de instrucciones que aparecen en la traza de simulación no es el mismo en el que se han introducido en la memoria de instrucción, ¿Por qué no?

c. Implementación de la instrucción HALT



Observando la traza del apartado anterior te habrás dado cuenta de que tras la instrucción HALT de la posición $20_{(16}$ de memoria, el procesador ha seguido leyendo instrucciones (ceros) secuencialmente (porque PC ha seguido incrementándose con cada flanco activo del reloj). Con la implementación de la instrucción HALT podremos parar el procesador.

Para dejar de generar flancos en la señal de reloj (aunque sigamos conmutándolo) basta con activar la entrada "halt" del circuito *ClockGen* que hace que su salida "clk" se mantenga a 1 para siempre y por tanto parando la ejecución del procesador (ya no llegan más flancos).

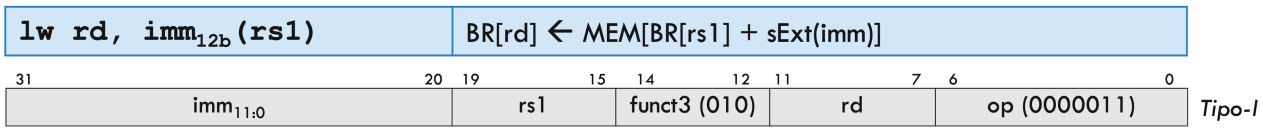
Por tanto, cuando se detecte/decodifique que la instrucción a ejecutar es una instrucción HALT (la unidad de control decodifica esta instrucción con op 1111111), lo que tiene que hacer el procesador es activar (poner a 1) la señal "halt" del circuito (túnel "halt").

Comprobación del funcionamiento: Simula el mismo programa anterior y comprueba como ahora tras la instrucción HALT (PC = $20_{(16)}$) ya no se ejecutan más instrucciones (PC se queda con el valor $24_{(16}$ por más que sigamos conmutando el reloj).

2. Implementación de las instrucciones LW y SW

Modifica el camino de datos y la unidad de control que has desarrollado hasta el momento para añadir las instrucciones LW y SW. Recuerda que debes añadir el nuevo HW y conexiones que consideres necesarios en tu procesador, pero manteniendo la posibilidad de ejecutar las instrucciones que has implementado previamente en el mismo (JAL y HALT).

a. Implementación de la instrucción LW



Esta instrucción lee de la memoria de datos un valor para escribirlo en el registro destino indicado por el campo de operando ***rd***. La dirección de acceso a memoria se calcula sumando **en la ALU** el contenido del registro especificado por el campo de operando ***rs1*** con el resultado de realizar la extensión de signo a 32 bits del campo de operando ***imm***.

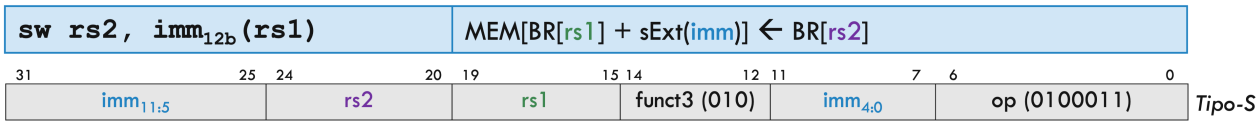
Para realizar la extensión de signo, puedes utilizar el circuito **Extensor** (ver ANEXO I) seleccionando la **extensión tipo-I**, ya que este es el formato de la instrucción LW.

La salida de la ALU que ha realizado la suma del registro más el inmediato, es la dirección de memoria que hay que leer de la memoria de datos.

El dato leído de memoria, deberá ser llevado al banco de registros y almacenado en el registro ***rd***.

Recuerda que el banco de registros y la memoria de datos necesitan una señal de reloj (clk) para funcionar.

b. Implementación de la instrucción SW



Esta instrucción lee el contenido del registro indicado por el campo de operando fuente ***rs2*** para escribirlo en la memoria de datos. La dirección de acceso a memoria se calcula sumando **en la ALU** el contenido del registro especificado por el campo de operando ***rs1*** con el resultado de realizar la extensión de signo a 32 bits del campo de operando ***imm***.

Para realizar la extensión de signo, puedes utilizar el circuito **Extensor** (ver ANEXO I) seleccionando la **extensión tipo-S**, ya que este es el formato de la instrucción SW.

Recuerda que el banco de registros y la memoria de datos necesitan una señal de reloj (clk) para funcionar.

c. Comprobación del funcionamiento de las instrucciones LW y SW

Introduce el siguiente programa en la memoria de instrucciones (**instmem.txt**) y los siguientes datos en la memoria de datos (**datamem.txt**).

Memoria de instrucciones

Dir.(hex)	Instrucciones	Contenido instmem.txt
0	`lw x1, 0(x0)`	00002083
4	`lw x2, 0(x1)`	0000a103
8	`sw x1, 4(x0)`	00102223
C	`sw x2, 0(x0)`	00202023
10	`lw x2, 8(x1)`	0080a103
14	`lw x1, 8(x0)`	00802083
18	`sw x2, 8(x0)`	00202423
1C	`sw x1, 0xC(x0)`	00102623
20	`halt`	ffffffff

Memoria de datos

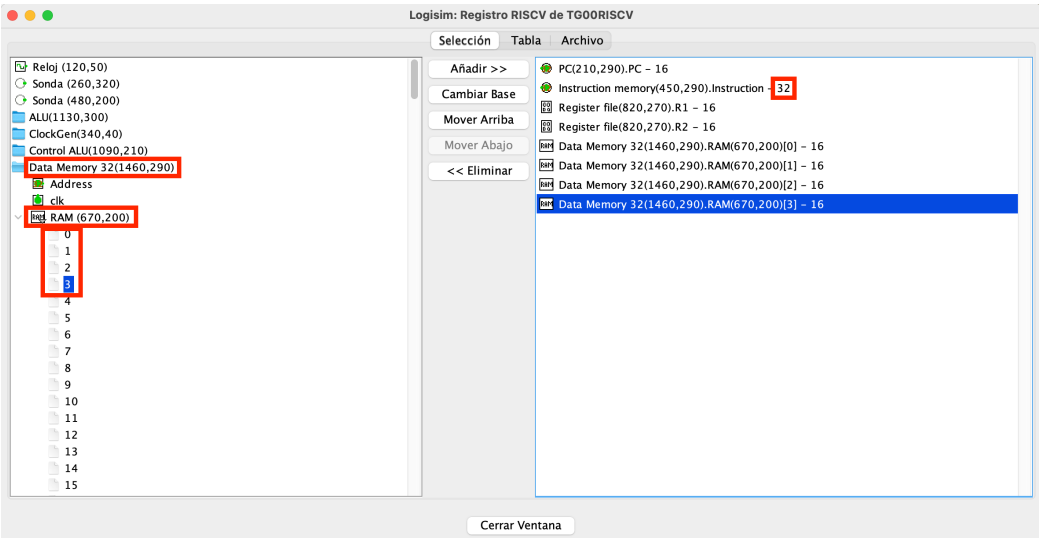
Dir.(hex)	Contenido datamem.txt
0	00000004
4	00000003
8	00000002
C	00000001
10	00000000
14	
18	
1C	
20	

Antes de poder simular correctamente, tienes que calcular el nuevo valor correcto de ticks que asignarle al reloj: debe darle tiempo a ejecutarse a la instrucción más lenta de las que has implementado. Comprueba el retardo sufrido en el camino que activa cada instrucción para su ejecución. Los retardos (en ticks) para las unidades funcionales que se te han proporcionado en el prototipo (ver ANEXO I) son los siguientes (considera 0 el retraso del resto de elementos que hayas añadido tú en el diseño):

- **Memoria de instrucciones: 4** (desde que se pone una dirección válida en la entrada de direcciones, hasta que aparece correctamente el dato almacenado en dicha dirección, hay que esperar 4 ticks)
- **Memoria de datos: 4** (tanto en lectura como en escritura. Para lectura hay que esperar 4 ticks desde que se pone la dirección hasta que está disponible el dato. En escritura hay que esperar 4 ticks desde que se pone la dirección y el dato a escribir, hasta que se ha llevado a cabo la escritura correctamente)
- **ALU: 4** (desde que se cambia alguna de sus entradas hasta que aparece el resultado correcto en la salida)
- **Banco de registros: 2** (tanto para lectura como para escritura)

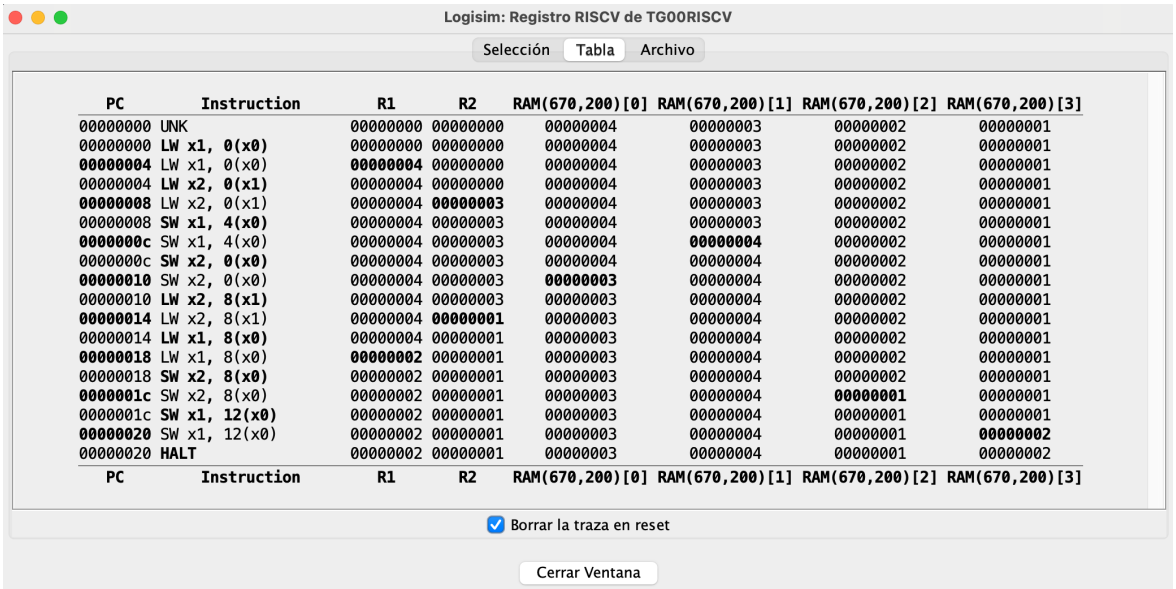
Si, por ejemplo, el retardo calculado para LW es X y el de SW es Y, tenemos que coger el mayor (la instrucción más lenta determina el tiempo de ciclo) y **configurar el reloj del procesador para que la suma de ticks en nivel alto más los ticks en nivel bajo sea igual a MAX(X, Y).**

Una vez configurado correctamente el reloj, simula el procesador usando la opción tabla de Logisim, mostrando el contenido de los registros PC, R1, R2 y las 4 primeras posiciones de memoria de datos (todos en hexadecimal). Puedes mostrar también la instrucción que se está ejecutando en cada momento (salida de la memoria de instrucción) y si cambias a Base “32” se mostrará la decodificación de las instrucciones en formato ensamblador RISC-V.



Analiza la traza siguiente (todos los valores están visualizados en hexadecimal) para comprender cómo se ejecuta cada instrucción. Comprueba que lo que tú imaginas que debe hacer la instrucción se ve reflejado en el contenido de los registros o la posición de la memoria afectados por la instrucción (ten en cuenta que el efecto de una instrucción no se ve hasta el ciclo siguiente, es decir, en la fila correspondiente al ciclo de instrucción/máquina de la siguiente instrucción). Comprueba que la tabla de tu simulación se corresponde con la traza dada.

	PC	R1	R2	RAM[0]	RAM[1]	RAM[2]	RAM[3]
`lw x1, 0(x0)`	00000000	00000000	00000000	00000004	00000003	00000002	00000001
`lw x2, 0(x1)`	00000004	**00000004**	00000000	00000004	00000003	00000002	00000001
`sw x1, 4(x0)`	00000008	00000004	**00000003**	00000004	00000003	00000002	00000001
`sw x2, 0(x0)`	0000000c	00000004	00000003	00000004	**00000004**	00000002	00000001
`lw x2, 8(x1)`	00000010	00000004	00000003	**00000003**	00000004	00000002	00000001
`lw x1, 8(x0)`	00000014	00000004	**00000001**	00000003	00000004	00000002	00000001
`sw x2, 8(x0)`	00000018	**00000002**	00000001	00000003	00000004	00000002	00000001
`sw x1, 0xC(x0)`	0000001c	00000002	00000001	00000003	00000004	**00000001**	00000001
`halt`	00000020	00000002	00000001	00000003	00000004	00000001	**00000002**

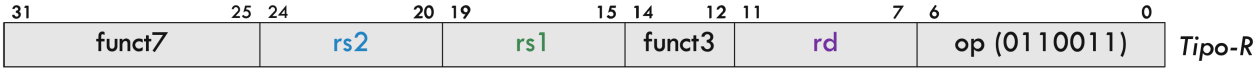


3. Implementación de las instrucciones ALU y ALUI

Modifica el camino de datos y la unidad de control que has desarrollado hasta el momento para añadir las instrucciones tipo ALU y ALUI. Recuerda que tus cambios no deben afectar a la correcta ejecución de las instrucciones que implementaste previamente (JAL, HALT, LW y SW).

a. Implementación de las instrucciones tipo ALU

`ALU rd, rs1, rs2`	BR[rd] ← BR[rs1] opALU BR[rs2]



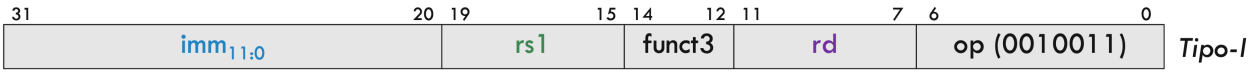
ALU	opALU	funct7	funct3
ADD	+	0000000	000
SUB	-	0100000	000
AND	&	0000000	111
OR	|	0000000	110
SLT	slt	0000000	010

Estas instrucciones deben llevar al registro indicado por el campo de operando ***rd***, el resultado de **operar en la ALU** el contenido de los registros indicados por los campos de operando ***rs1*** y ***rs2***. La operación que debe realizar la ALU se viene especificada en los campos ***funct7*** y ***funct3*** de la instrucción máquina, según la tabla anterior.

Para que la ALU realice la operación adecuada a la instrucción, habrá que proporcionarle al circuito Control ALU los campos funct7 y funct3 e indicarle que la instrucción es de tipo-R.

b. Implementación de las instrucciones tipo ALUI

`alui rd, rs1, imm`$_{12b}$	BR[rd] ← BR[rs1] opALU sExt(imm)



ALU	opALU	funct3
ADDI	+	000
ANDI	&	111
ORI	|	110
SLTI	slt	010

Estas instrucciones llevan al registro especificado por el campo de operando ***rd***, el resultado de **operar en la ALU** el contenido del registro especificado por el campo de operando ***rs1*** con el dato inmediato

especificado en el campo de operando ***imm***, extendido en signo hasta los 32 bits.

Para realizar la extensión de signo, puedes utilizar el circuito **Extensor** (ver ANEXO I) seleccionando la **extensión tipo-I**, ya que este es el formato de estas instrucciones.

Para que la ALU realice la operación adecuada a la instrucción, habrá que proporcionarle al circuito Control ALU el campo funct3 e indicarle que la instrucción es de tipo-I.

c. Comprobación del funcionamiento de las instrucciones ALU y ALUI

Introduce el siguiente programa en la memoria de instrucciones (**instmem.txt**) y los siguientes datos en la memoria de datos (**datamem.txt**).

Memoria de instrucciones

Dir.(hex)	Instrucciones	Contenido instmem.txt
0	lw x1, 4(x0)	00402083
4	lw x2, 4(x1)	0040a103
8	add x3, x2, x1	001101b3
C	sub x3, x3, x2	402181b3
10	addi, x3, x2, -500	e0c10193
14	ori x2, x2, 0x7a0	7a016113
18	andi x3, x3, 0x7ff	7ff1f193
1C	and x1, x2, x3	003170b3
20	or x1, x2, x3	003160b3
24	slt x3, x1, x2	0020a1b3
28	slt x3, x2, x1	001121b3
2C	slti x3, x1, -0x7ff	8010a193
30	slti x3, x1, 0x7ff	7ff0a193
34	sw x1, 0(x0)	00102023
38	halt	ffffffff

Memoria de datos

Dir.(hex)	Contenido datamem.txt
0	3
4	-4
8	0

Simula el procesador usando la opción tabla de Logisim, mostrando el contenido de los registros PC, R1, R2, R3 y la primera posición de memoria de datos (todos en hexadecimal).

Analiza la traza siguiente (todos los valores están visualizados en hexadecimal) para comprender como se ejecuta cada instrucción. Comprueba que lo que tú piensas que debe hacer la instrucción se ve reflejado en el contenido de los registros o la posición de la memoria afectados por la instrucción (ten en cuenta que el efecto de una instrucción no se ve hasta el ciclo siguiente, es decir, en la fila de la siguiente instrucción). Comprueba que la tabla de tu simulación se corresponde con la traza dada.

A la hora de analizar el resultado de las operaciones aritméticas (ADD, SUB, ADDI), ten en cuenta que la representación de los números enteros en el procesador utiliza el convenio C2 y que la

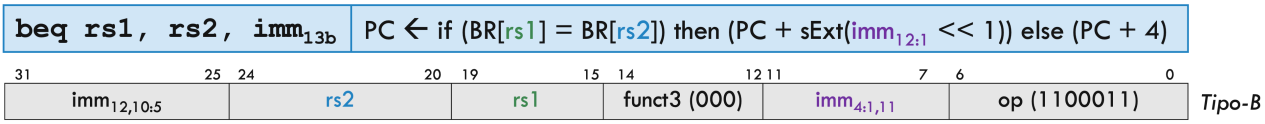
visualización en la tabla está en hexadecimal (así por ejemplo el número -1, sería una cadena binaria de 32 1's, que representado en hexadecimal sería ffffffff)

	PC	R1	R2	R3	RAM[0]
lw x1, 4(x0)	00000000	00000000	00000000	00000000	00000003
lw x2, 4(x1)	00000004	**fffffffc**	00000000	00000000	00000003
add x3, x2, x1	00000008	fffffffc	**00000003**	00000000	00000003
sub x3, x3, x2	0000000c	fffffffc	00000003	**ffffffff**	00000003
addi, x3, x2, -500	00000010	fffffffc	00000003	**fffffffc**	00000003
ori x2, x2, 0x7a0	00000014	fffffffc	00000003	**fffffe0f**	00000003
andi x3, x3, 0x7ff	00000018	fffffffc	**000007a3**	fffffe0f	00000003
and x1, x2, x3	0000001c	fffffffc	000007a3	**0000060f**	00000003
or x1, x2, x3	00000020	**00000603**	000007a3	0000060f	00000003
slt x3, x1, x2	00000024	**000007af**	000007a3	0000060f	00000003
slt x3, x2, x1	00000028	000007af	000007a3	**00000000**	00000003
slti x3, x1, -0x7ff	0000002c	000007af	000007a3	**00000001**	00000003
slti x3, x1, 0x7ff	00000030	000007af	000007a3	**00000000**	00000003
sw x1, 0(x0)	00000034	000007af	000007a3	**00000001**	00000003
halt	00000038	000007af	000007a3	00000001	**000007af**

4. Implementación de la instrucción BEQ

Modifica el camino de datos y la unidad de control que has desarrollado hasta el momento para añadir la instrucción de salto condicional BEQ. Recuerda que deben seguirse ejecutando correctamente las instrucciones que implementaste previamente en el mismo (JAL, HALT, LW, SW, ALU y ADDI).

a. Implementación de la instrucción BEQ



Esta instrucción comprobará si el contenido de los registros especificados por los campos de operando **rs1** y **rs2** es el mismo. Si no es igual, la instrucción no hará nada (simplemente se incrementará PC: PC = PC + 4). En el caso de que el contenido sea igual, se ejecutará el salto, es decir, se cargará un valor en PC distinto de PC+4. El valor a cargar se calcula sumándole a PC el valor del campo de operando ***imm*** extendido en signo hasta 32.

Para realizar la extensión de signo, puedes utilizar el circuito **Extensor** (ver ANEXO I) seleccionando la **extensión tipo-B**, ya que este es el formato de estas instrucciones.

Para comprobar si el contenido de los dos registros es igual, **OBLIGATORAIMENTE hay que utilizar la operación resta de la ALU** y comprueba si el resultado es 0 observando la salida Zero de la ALU que indica con su activación si el resultado de la operación realizada en la misma da como resultado 0.

b. Comprobación del funcionamiento de la instrucción BEQ

Para comprobar el funcionamiento de esta instrucción utilizaremos un código que va a utilizar todas las instrucciones del procesador que hemos ido implementando, de manera que podamos comprobar que no ha dejado de funcionar ninguna de ellas con las modificaciones que se han ido realizando.

Introduce el siguiente programa en la memoria de instrucciones (**instmem.txt**) y los siguientes datos en la memoria de datos (**datamem.txt**). El programa consta de un bucle que recorre secuencialmente un vector leyendo los valores almacenados para irlos sumando. Al terminar el recorrido del bucle, el resultado de la acumulación se almacena en memoria. Analiza el código, asegurándote de entender bien la función de cada uno de los registros (contador de elementos, acumulador de la suma, dirección del vector, ...) y qué hace exactamente cada una de las instrucciones para conseguir el valor de la suma total.

Memoria de instrucciones

Dir. (hex)	Instrucciones	Contenido instmem.txt
0	lw x1, 0x14(x0)	01402083
4	addi x2, x0, 4	00400113
8	sub x3, x3, x3	403181b3
C	add x4, x0, x0	00000233
10	lo: beq x4, x1, sl	00120c63
14	lw x5, 0(x2)	00012283
18	add x3, x3, x5	005181b3
1C	addi x2, x2, 4	00410113
20	addi x4, x4, 1	00120213
24	jal x0, lo	fedff06f
28	sl: sw x3, 0(x0)	00302023
2C	halt	ffffffff

Memoria de datos

Dir. (hex)	Contenido datamem.txt	
0	00000000	Result
4	fffffffe	Vector
8	00000004	
C	fffffffa	
10	00000008	
14	00000004	Size

Simula el procesador usando la opción tabla de Logisim, mostrando el contenido de los registros PC, R1, R2, R3, R4, R5 y la primera posición de memoria de datos (todos en hexadecimal).

Analiza la traza siguiente (todos los valores están visualizados en hexadecimal) para comprender cómo se ejecuta cada instrucción. Comprueba que lo que tú piensas que debe hacer la instrucción se ve reflejado en el contenido de los registros o la posición de la memoria afectados por la instrucción (ten en cuenta que el efecto de una instrucción no se ve hasta el ciclo siguiente, es decir, en la fila de la siguiente instrucción). Comprueba que la tabla de tu simulación se corresponde con la traza dada.

Ten en cuenta de nuevo, a la hora de interpretar la traza, que los datos del vector son números enteros de 32 bits representados en C2 y visualizados en hexadecimal.

		PC	R1	R2	R3	R4	R5	RAM[0]
	lw x1, 0x14(x0)	00000000	00000000	00000000	00000000	00000000	00000000	00000000
	addi x2, x0, 4	00000004	**00000004**	00000000	00000000	00000000	00000000	00000000
	sub x3, x3, x3	00000008	00000004	**00000004**	00000000	00000000	00000000	00000000
	add x4, x0, x0	0000000c	00000004	00000004	**00000000**	00000000	00000000	00000000
lo:	beq x4, x1, sl	00000010	00000004	00000004	00000000	**00000000**	00000000	00000000
	lw x5, 0(x2)	00000014	00000004	00000004	00000000	00000000	00000000	00000000
	add x3, x3, x5	00000018	00000004	00000004	00000000	00000000	**fffffffe**	00000000
	addi x2, x2, 4	0000001c	00000004	00000004	**fffffffe**	00000000	fffffffe	00000000
	addi x4, x4, 1	00000020	00000004	**00000008**	fffffffe	00000000	fffffffe	00000000
	j lo	00000024	00000004	00000008	fffffffe	**00000001**	fffffffe	00000000
lo:	beq x4, x1, sl	**00000010**	00000004	00000008	fffffffe	00000001	fffffffe	00000000
	lw x5, 0(x2)	00000014	00000004	00000008	fffffffe	00000001	fffffffe	00000000
	add x3, x3, x5	00000018	00000004	00000008	fffffffe	00000001	**00000004**	00000000
	addi x2, x2, 4	0000001c	00000004	00000008	**00000002**	00000001	00000004	00000000
	addi x4, x4, 1	00000020	00000004	**0000000c**	00000002	00000001	00000004	00000000
	j lo	00000024	00000004	0000000c	00000002	**00000002**	00000004	00000000
lo:	beq x4, x1, sl	**00000010**	00000004	0000000c	00000002	00000002	00000004	00000000
	lw x5, 0(x2)	00000014	00000004	0000000c	00000002	00000002	00000004	00000000
	add x3, x3, x5	00000018	00000004	0000000c	00000002	00000002	**fffffffa**	00000000
	addi x2, x2, 4	0000001c	00000004	0000000c	**fffffffc**	00000002	fffffffa	00000000
	addi x4, x4, 1	00000020	00000004	**00000010**	fffffffc	00000002	fffffffa	00000000
	j lo	00000024	00000004	00000010	fffffffc	**00000003**	fffffffa	00000000
lo:	beq x4, x1, sl	**00000010**	00000004	00000010	fffffffc	00000003	fffffffa	00000000
	lw x5, 0(x2)	00000014	00000004	00000010	fffffffc	00000003	fffffffa	00000000
	add x3, x3, x5	00000018	00000004	00000010	fffffffc	00000003	**00000008**	00000000
	addi x2, x2, 4	0000001c	00000004	00000010	**00000004**	00000003	00000008	00000000
	addi x4, x4, 1	00000020	00000004	**00000014**	00000004	00000003	00000008	00000000
	j lo	00000024	00000004	00000014	00000004	**00000004**	00000008	00000000
lo:	beq x4, x1, sl	**00000010**	00000004	00000014	00000004	00000004	00000008	00000000
sl:	sw x3, 0(x0)	**00000028**	00000004	00000014	00000004	00000004	00000008	00000000
	halt	0000002c	00000004	00000014	00000004	00000004	00000008	**00000004**

5. Programación del procesador

El objetivo de este apartado es realizar un programa que se ejecute en el procesador diseñado.

El programa que tenéis que implementar debe calcular la suma de las diferencias en valor absoluto de los elementos de dos vectores (SAD), operación que se utiliza para medir cuanto se parecen dos vectores A y B.

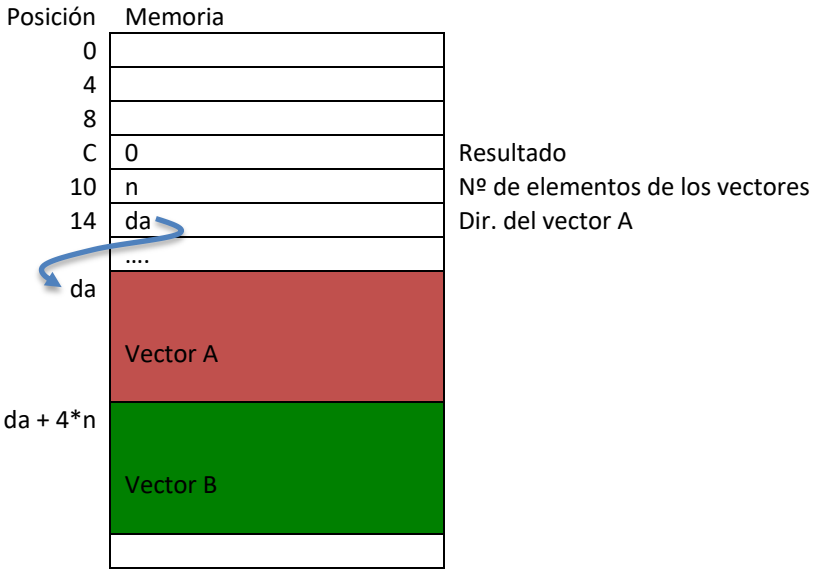
$$SAD(A,B) = \sum_i |A[i] - B[i]|$$

Para ello ten en cuenta lo siguiente:

- El tamaño de los vectores (número de elementos, n) estará almacenado en la posición de memoria $10_{(16}$.
- La posición de memoria donde comienza el vector A estará almacenada en la posición de memoria $14_{(16}$.
- La posición de memoria donde comienza el vector B es la posición siguiente a la última del vector A (B está almacenado justo a continuación del vector A).
- Tendrás que realizar la resta elemento a elemento de los vectores (el primero de A menos el primero de B, el segundo menos el segundo, …). A dicha resta tendrás que calcular su valor absoluto, e ir acumulando la suma de todos los valores absolutos calculados.

- Debes almacenar el resultado en la posición de memoria $C_{(16}$.
- Las posiciones de memoria $0_{(16}$, $4_{(16}$, y $8_{(16}$ las tienes libres para usarlas como mejor estimes oportuno. Se pueden usar para variables que necesite el programa.

  El esquema de la memoria de datos para este programa sería:



Diseña el programa propuesto utilizando las instrucciones disponibles en la implementación monociclo del procesador. Introdúcelo a partir de la posición 0 de la memoria de instrucciones (fichero instmem.txt).

Para la programación, simulación y generación del contenido de los ficheros asociados a las memorias del procesador (instmem.txt y datamem.txt) puedes utilizar la herramienta RARS utilizada para la programación en ensamblador RISC-V.

El siguiente segmento de datos es un ejemplo para dos vectores de 4 elementos cada uno (para los cuales el resultado del programa debe ser 8).

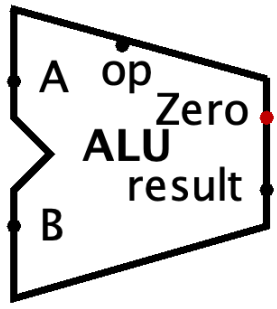
```
.data
.word 0, 0, 0            //direcciones libres
.word 0                  //resultado
.word 4                  //n - número de elementos de A y B
.word 24                 //dirección del vector A
.word 25, -3, -6, 1      //datos del vector A
.word 23, -1, -4, -1     //datos del vector B

.text
//tu código aquí
```

ANEXO I – Elementos hardware

ALU

Retardo: 4 unidades de tiempo (ticks)

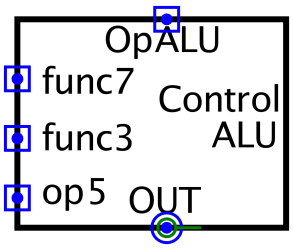


Señales	
A[31:0]	Dato de entrada
B[31:0]	Dato de entrada
op[3:0]	Operación a realizar por la ALU
result[31:0]	Dato de salida
Zero	Se activa (1) cuando el resultado es cero

op[3:0]	Operación ALU	
0000	result = A & B	And (bit a bit)
0001	result = A | B	Or (bit a bit)
0010	result = A + B	Suma
0110	result = A – B	Resta
0111	If (A<B) result = 1 else result = 0	Activar si A menor que B
1100	result = $\overline{A \mid B}$	Nor (bit a bit)

Control ALU

Retardo: 0 unidades de tiempo (ticks)

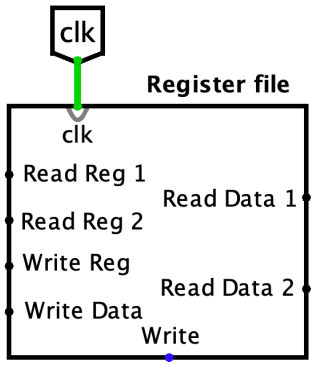


Señales	
OpALU[1:0]	Selección de operación de la ALU
func7[6:0] func3[2:0] op5	Códigos de función de las instrucciones ALU/ALUI y bit 5 del opcode (op)
OUT[3:0]	Código de operación para la ALU (op), **conectar directamente a la ALU**

OpALU[2:0]	OUT
00	0010 (ALU suma)
01	0110 (ALU resta)
10	Operación indicada por func3 y func7 (instrucciones ALU y ALUI)

BANCO DE REGISTROS (Register file)

Retardo: 4 unidades de tiempo (ticks)

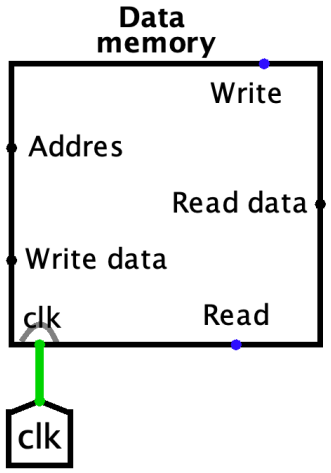


Señales	
Clk	Señal de reloj para sincronizar la escritura
Write	Señal de escritura (1)
Read Data 1[31:0]	Dato leído del registro indicado en Read Reg 1
Read Data 2[31:0]	Dato leído del registro indicado en Read Reg 2
Write Data[31:0]	Dato a escribir en el registro indicado en Write Reg
Read Reg 1[5:0]	Código del registro a leer por salida Read Data 1
Read Reg 2[5:0]	Código del registro a leer por salida Read Data 2
Write Reg[5:0]	Código del registro a escribir

Write	Operación del registro
0	Lectura: Read Data 1[31:0] = Contenido almacenado en el registro indicado por Read Reg 1[5:0] Read Data 2[31:0] = Contenido almacenado en el registro indicado por Read Reg 2[5:0]
1	Escritura por flanco de subida de CLK: Registro indicado por Write Reg[5:0] se carga con Write Data[31:0] Además se realizan las dos lecturas al igual que con Write=0

Memoria de datos (Data Memory 32)

Retardo: 4 unidades de tiempo (ticks)

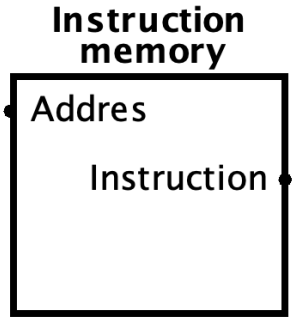


Señales	
clk	Señal de reloj para sincronizar las escrituras en memoria
Write	Señal de escritura
Read	Señal de lectura
Read data[31:0]	Bus de datos de salida (lectura)
Write data[31:0]	Bus de datos de entrada (escritura)
Address[31:0]	Bus de direcciones

Write	Read	Operación de la memoria
0	1	Lectura: Read data[31:0] = M[Address[31:0]]
1	0	Escritura por flanco de subida de clk: M[Address[31:0]] ← Read data[31:0]

Memoria de Instrucciones (Instruction memory)

Retardo: 4 unidades de tiempo (ticks)

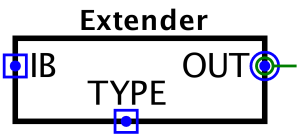


Señales	
Instruction[31:0]	Bus de datos de salida (lectura)
Address[31:0]	Bus de direcciones

Operación de la memoria
Continuamente: Instruction[31:0] = M[Address[31:0]]

Extensor (Extender)

Retardo: 0 unidades de tiempo (ticks)

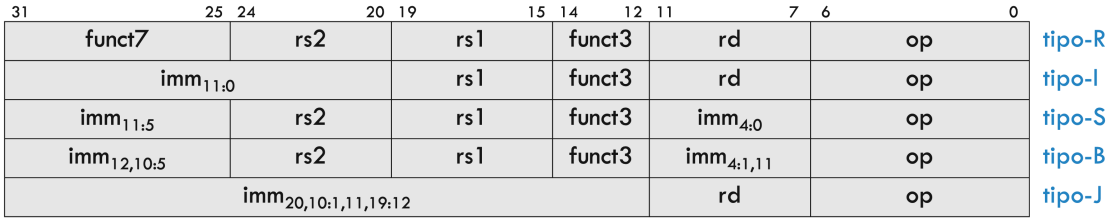


Señales	
IB[31:0]	Instrucción con inmediato a extender
TYPE[2:0]	Tipo de extensión
OUT[3:0]	Inmediato extendido a 32 bits

TYPE[2:0]	OUT
000	Extensión **tipo-I**: OUT[31:0] = ExtSigno(IB[31:20])
001	Extensión **tipo-S**: OUT[31:0] = ExtSigno(IB[31:25],IB[11:7])
010	Extensión **tipo-B**: OUT[31:0] = ExtSigno(IB[31],IB[7],IB30:25],IB[11:8],0)
011	Extensión **tipo-U**: OUT[31:0] = IB[31:12],000000000000
100	Extensión **tipo-J**: OUT[31:0] = ExtSigno(IB[31],IB[19:12],IB[20],IB[30:21],0)

ANEXO II – Instrucciones

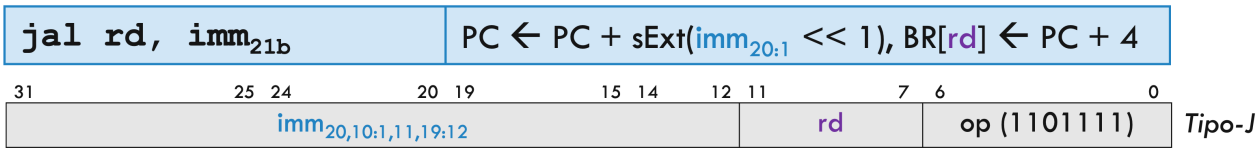
FORMATOS DE INSTRUCCIÓN



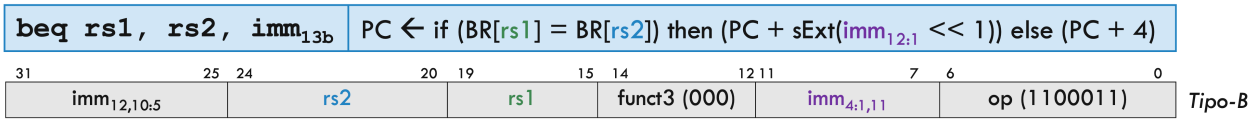
INSTRUCCIONES

El conjunto mínimo de instrucciones que debe soportar el procesador es:

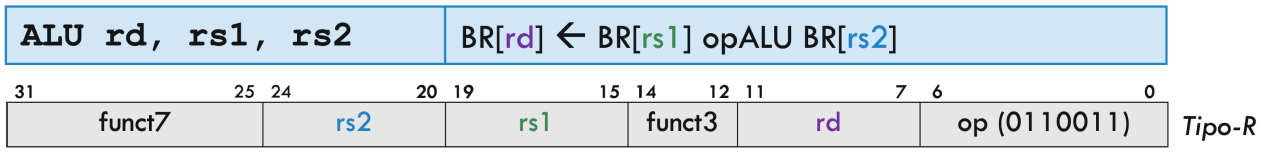
- **Instrucciones de Bifurcación:**
  - Salto incondicional: **JAL imm**



  - Salto condicional: **BEQ rs1, rs2, imm**



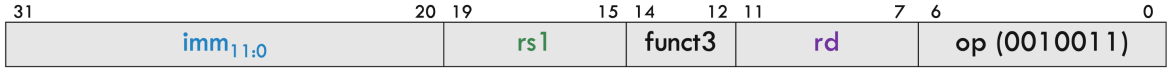
- **Instrucciones de transformación de información:**
  - Operaciones con la ALU: **ALU rd, rs, rt**



ALU	opALU	funct7	funct3
ADD	+	0000000	000
SUB	-	0100000	000
AND	&	0000000	111
OR	|	0000000	110
SLT	slt	0000000	010

- Operaciones inmediatas con la ALU: **ALUI rt, rs, imm**

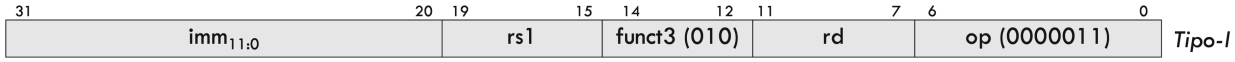
alui rd, rs1, imm_{12b}	BR[rd] ← BR[rs1] opALU sExt(imm)



ALU	opALU	funct3
ADDI	+	000
ANDI	&	111
ORI	|	110
SLTI	slt	010

- **Instrucciones de transferencia:**
  - Carga: **LW rt, desp.(rs)**

lw rd, imm_{12b}(rs1)	BR[rd] ← MEM[BR[rs1] + sExt(imm)]



  - Almacenamiento: **SW rt, desp.(rs)**

sw rs2, imm_{12b}(rs1)	MEM[BR[rs1] + sExt(imm)] ← BR[rs2]

- **Instrucción especial:**
  - **HALT**

halt	Inhabilita señal de reloj