

# Taller de arquitectura

*TDA 1819: Descripción y simulación de una computadora de 32 bits.*



Autor: Carballo Ormaeachea, Lucas.

Contacto: lucascarballo17@gmail.com

Facultad de Informática, UNLP, 50 y 120, La Plata.

## **Introducción:**

Antes de poder hablar del procesador en sí se debe tener en cuenta que el mismo es una parte en un conjunto de componentes que modelan o, mejor dicho, simulan una PC, casi por completo. Un aspecto a remarcar es la implementación de una memoria principal y un ensamblador, para asemejarse lo mayor posible al simulador MSX88 o WinMIPS64. La memoria para almacenar instrucciones y datos para ejecutar un programa, y el ensamblador para poder leer un archivo escrito en Assembler y traducirlo a código máquina. Otra parte importante de dicho conjunto es la máquina de estados que gestiona toda la pc. Su tarea será gestionar simultáneamente el correcto funcionamiento de la segmentación de varias instrucciones que se encuentren en distintas etapas del cauce. Siempre con una instrucción por etapa.

## **TDA 1819:**

Este procesador, si bien mantiene algunas características de sus versiones anteriores más orientadas a la arquitectura CISC, es casi por completo un procesador del tipo RISC, es decir un “híbrido” entre ambas arquitecturas. Al ser, en gran parte, un procesador de arquitectura RISC posee implementada la metodología de segmentación del cauce con el fin de acelerar y optimizar la labor a realizar, una gran cantidad de registros de uso general tanto enteros como de punto flotante y algunos modos de direccionamiento requieren solo de un ciclo de reloj para ejecutarse. Al planificar el TDA1819 se planteó inicialmente la labor de modelar tres componentes individuales para representar las tres secciones principales de la CPU(unidad de control, unidad aritmético-lógica y el banco de registros), cada una con su propia tarea particular y exclusiva; así como también se consideraron las cinco etapas que constituyen el ciclo de ejecución de la implementación tradicional de la segmentación del cauce ya que éste será al fin y al cabo el que terminará rigiendo el comportamiento general de la CPU, concluyendo así la necesidad de incorporar dos componentes adicionales, uno encargado exclusivamente de realizar la búsqueda de la siguiente instrucción y actualizar el Puntero de Instrucción (etapa de búsqueda) y otro para gestionar el acceso a la memoria principal desde la CPU para operaciones de lectura o escritura (etapa de acceso a memoria). Esta independencia de los componentes le da al procesador una gran predisposición a recibir mejoras o actualizaciones, por ejemplo en su repertorio de instrucciones, a tal punto que no es necesario modificar todos los componentes para que no se altere el funcionamiento del conjunto ( como sí ocurría en versiones anteriores del mismo procesador).

## Directivas

Las siguientes son las directivas que el ensamblador puede reconocer e interpretar y que definen las distintas secciones de nuestro programa escrito en Assembler:

Directiva	Comentario
.data	Comienzo de la sección de datos
.subr	Comienzo de la sección de subrutinas (*)
.text	Comienzo de la sección de código del programa principal
.code	Comienzo de la sección de código del programa principal (igual que .text)
.ascii <i>s1, s2, ...</i>	Almacena cadena/s ASCII
.asciiz <i>s1, s2, ...</i>	Almacena cadena/s ASCII terminada/s en cero
.byte <i>n1, n2, ...</i>	Almacena número/s de 8 bits con signo
.ubyte <i>n1, n2, ...</i>	Almacena número/s de 8 bits sin signo
.hword <i>n1, n2, ...</i>	Almacena número/s de 16 bits con signo
.uhword <i>n1, n2, ...</i>	Almacena número/s de 16 bits sin signo
.word <i>n1, n2, ...</i>	Almacena número/s de 32 bits con signo
.uword <i>n1, n2, ...</i>	Almacena número/s de 32 bits sin signo
.float <i>n1, n2, ...</i>	Almacena número/s en punto flotante (32 bits)

A continuación se detalla el repertorio de instrucciones de el procesador:

Tipo de instrucción	Instrucción	Comentario
Transferencia de Datos	lb $r_d, Inm(r_i)$	Copia en $r_d$ un byte (8 bits) desde la dirección $(Inm + r_i)$
	sb $r_f, Inm(r_i)$	Guarda los 8 bits menos significativos de $r_f$ en la dirección $(Inm + r_i)$
	lh $r_d, Inm(r_i)$	Copia en $r_d$ un half-word (16 bits) desde la dirección $(Inm + r_i)$
	sh $r_f, Inm(r_i)$	Guarda los 16 bits menos significativos de $r_f$ a partir de la dirección $(Inm + r_i)$
	lw $r_d, Inm(r_i)$	Copia en $r_d$ un word (32 bits) desde la dirección $(Inm + r_i)$
	sw $r_f, Inm(r_i)$	Guarda los 32 bits de $r_f$ a partir de la dirección $(Inm + r_i)$
	lf $f_d, Inm(r_i)$	Copia en $f_d$ un valor en punto flotante (32 bits) desde la dirección $(Inm + r_i)$
	sf $f_f, Inm(r_i)$	Guarda los 32 bits de $f_f$ a partir de la dirección $(Inm + r_i)$
	mff $f_d, f_f$	Copia el valor del registro $f_f$ (32 bits) al registro $f_d$
	mfr $f_d, r_f$	Copia los 32 bits del registro $r_f$ entero al

		registro $f_d$ de punto flotante
	mrf $r_d, f_f$	Copia los 32 bits del registro $f_f$ de punto flotante al registro $r_d$ entero
	tf $f_d, f_f$	Convierte a punto flotante el valor entero copiado al registro $f_f$ (32 bits), dejándolo en $f_d$
	ti $f_d, f_f$	Convierte a entero el valor en punto flotante contenido en $f_f$ (32 bits), dejándolo en $f_d$

Aritmética	dadd $r_d, r_f, r_g$	Suma $r_f$ con $r_g$ , dejando el resultado en $r_d$ (valores con signo)
	daddi $r_d, r_f, N$	Suma $r_f$ con el valor inmediato $N$ , dejando el resultado en $r_d$ (valores con signo)
	addir $r_d, r_f, r_g$	Suma $r_f$ con $r_g$ , dejando el resultado en $r_d$ (valores sin signo)
	addirui $r_d, r_f, N$	Suma $r_f$ con el valor inmediato $N$ , dejando el resultado en $r_d$ (valores sin signo)
	addf $f_d, f_f, f_g$	Suma $f_f$ con $f_g$ , dejando el resultado en $f_d$ (en punto flotante)
	dsub $r_d, r_f, r_g$	Resta $r_g$ a $r_f$ , dejando el resultado en $r_d$ (valores con signo)
	dsubu $r_d, r_f, r_g$	Resta $r_g$ a $r_f$ , dejando el resultado en $r_d$ (valores sin signo)
	subf $f_d, f_f, f_g$	Resta $f_g$ a $f_f$ , dejando el resultado en $f_d$ (en punto flotante)
	dmul $r_d, r_f, r_g$	Multiplica $r_f$ con $r_g$ , dejando el resultado en $r_d$ (valores con signo)
	dmulu $r_d, r_f, r_g$	Multiplica $r_f$ con $r_g$ , dejando el resultado en $r_d$ (valores sin signo)
	mulf $f_d, f_f, f_g$	Multiplica $f_f$ con $f_g$ , dejando el resultado en $f_d$ (en punto flotante)
	ddiv $r_d, r_f, r_g$	Divide $r_f$ por $r_g$ , dejando el resultado en $r_d$ (valores con signo)
	ddivu $r_d, r_f, r_g$	Divide $r_f$ por $r_g$ , dejando el resultado en $r_d$ (valores sin signo)
	divf $f_d, f_f, f_g$	Divide $f_f$ por $f_g$ , dejando el resultado en $f_d$ (en punto flotante)

Comparación		punto flotante)
	slt $r_d, r_f, r_g$	Compara $r_f$ con $r_g$ , dejando $r_d=1$ si $r_f$ es menor que $r_g$ (valores con signo)
	slti $r_d, r_f, N$	Compara $r_f$ con el valor inmediato $N$ , dejando $r_d=1$ si $r_f$ es menor que $N$ (valores con signo)
	ltf $f_f, f_g$	Compara $f_f$ con $f_g$ , dejando el flag FP=1 si $f_f$ es menor que $f_g$ (en punto flotante)
	lef $f_f, f_g$	Compara $f_f$ con $f_g$ , dejando el flag FP=1 si $f_f$ es menor o igual que $f_g$ (en punto flotante)
	eqf $f_f, f_g$	Compara $f_f$ con $f_g$ , dejando el flag FP=1 si $f_f$ es igual que $f_g$ (en punto flotante)
	negr $r_d, r_f$	Calcula el opuesto de $r_f$ , dejando el resultado en $r_d$ (valores con signo)

<b>Lógica</b>	<b>andr</b> $r_d, r_f, r_g$	Realiza un AND entre $r_f$ y $r_g$ (bit a bit), dejando el resultado en $r_d$
	<b>andi</b> $r_d, r_f, N$	Realiza un AND entre $r_f$ y el valor inmediato $N$ (bit a bit), dejando el resultado en $r_d$
	<b>orr</b> $r_d, r_f, r_g$	Realiza un OR entre $r_f$ y $r_g$ (bit a bit), dejando el resultado en $r_d$
	<b>ori</b> $r_d, r_f, N$	Realiza un OR entre $r_f$ y el valor inmediato $N$ (bit a bit), dejando el resultado en $r_d$
	<b>xorr</b> $r_d, r_f, r_g$	Realiza un XOR entre $r_f$ y $r_g$ (bit a bit), dejando el resultado en $r_d$
	<b>xori</b> $r_d, r_f, N$	Realiza un XOR entre $r_f$ y el valor inmediato $N$ (bit a bit), dejando el resultado en $r_d$
	<b>notr</b> $r_d, r_f$	Realiza un NOT sobre $r_f$ (bit a bit), dejando el resultado en $r_d$
<b>Desplazamiento de Bits</b>	<b>dsl</b> $r_d, r_f, r_N$	Desplaza a izquierda $r_N$ veces los bits del registro $r_f$ , dejando el resultado en $r_d$
	<b>dsli</b> $r_d, r_f, N$	Desplaza a izquierda $N$ veces los bits del registro $r_f$ , dejando el resultado en $r_d$
	<b>dsr</b> $r_d, r_f, r_N$	Desplaza a derecha $r_N$ veces los bits del registro $r_f$ , dejando el resultado en $r_d$
	<b>dsri</b> $r_d, r_f, N$	Desplaza a derecha $N$ veces los bits del

<b>Transferencia de Control</b>		registro $r_f$ , dejando el resultado en $r_d$
	<b>dsls</b> $r_d, r_f, r_N$	Igual que la instrucción <b>dsl</b> pero mantiene el signo del valor desplazado
	<b>dslsi</b> $r_d, r_f, N$	Igual que la instrucción <b>dsli</b> pero mantiene el signo del valor desplazado
	<b>dsrs</b> $r_d, r_f, r_N$	Igual que la instrucción <b>dsr</b> pero mantiene el signo del valor desplazado
	<b>dsrsi</b> $r_d, r_f, N$	Igual que la instrucción <b>dsri</b> pero mantiene el signo del valor desplazado
	<b>jmp</b> $offN_I$	Salta incondicionalmente a la instrucción etiquetada $offN_I$
	<b>beq</b> $r_f, r_g, offN_I$	Si $r_f$ es igual a $r_g$ , salta a la instrucción etiquetada $offN_I$
	<b>bne</b> $r_f, r_g, offN_I$	Si $r_f$ no es igual a $r_g$ , salta a la instrucción etiquetada $offN_I$
	<b>beqz</b> $r_f, offN_I$	Si $r_f$ es igual a 0, salta a la instrucción etiquetada $offN_I$
	<b>bnez</b> $r_f, offN_I$	Si $r_f$ no es igual a 0, salta a la instrucción etiquetada $offN_I$
	<b>bfpt</b> $offN_I$	Salta a la instrucción etiquetada $offN_I$ si el flag F=1
	<b>bfpf</b> $offN_I$	Salta a la instrucción etiquetada $offN_I$ si el flag F=0

<b>Subrutinas</b>	call <i>offNs</i>	Llama a subrutina cuyo inicio es la instrucción etiquetada <i>offNs</i> (*)
	ret	Retorna de la subrutina (*)
<b>Manejo de Interrupciones</b>	int <i>N<sub>I</sub></i>	Salva los flags y ejecuta la interrupción por software <i>N<sub>I</sub></i> (*)
	iret	Retorna de la interrupción y restablece los flags (*)
	cli	Inhabilita interrupciones enmascarables (*)
	sti	Habilita interrupciones enmascarables (*)
<b>Manejo de la Pila</b>	pushb <i>r<sub>f</sub></i>	Apila los 8 bits menos significativos de <i>r<sub>f</sub></i> (*)
	popb <i>r<sub>d</sub></i>	Desapila un byte (8 bits) y lo carga en <i>r<sub>d</sub></i> (*)

	pushhh $r_f$	Apila los 16 bits menos significativos de $r_f$ (*)
	pophh $r_d$	Desapila un half-word (16 bits) y lo carga en $r_d$ (*)
	pushhw $r_f$	Apila los 32 bits de $r_f$ (*)
	pophw $r_d$	Desapila un word (32 bits) y lo carga en $r_d$ (*)
	pushfl	Apila los 8 bits del registro FLAGS (*)
	popfl	Desapila un byte (8 bits) y lo carga en el registro FLAGS (*)
	pushfp	Apila los 8 bits del registro FLAGS (punto flotante) (*)
	popfp	Desapila un byte (8 bits) y lo carga en el registro FLAGS (punto flotante) (*)
	pushra	Apila los 32 bits del registro RA (*)
	popra	Desapila un word (32 bits) y lo carga en el registro RA (*)
Entrada/Salida	inb $r_d, offN_P$	Carga el valor del puerto etiquetado $offN_P$ en los 8 bits menos significativos de $r_d$ (*)
	outb $r_f, offN_P$	Carga los 8 bits menos significativos de $r_f$ en el puerto etiquetado $offN_P$ (*)
	inh $r_d, offN_P$	Carga el valor del puerto etiquetado $offN_P$ en los 16 bits menos significativos de $r_d$ (*)
	outh $r_f, offN_P$	Carga los 16 bits menos significativos de $r_f$ en el puerto etiquetado $offN_P$ (*)
	inw $r_d, offN_P$	Carga el valor del puerto etiquetado $offN_P$ en los 32 bits de $r_d$ (*)
	outw $r_f, offN_P$	Carga los 32 bits de $r_f$ en el puerto etiquetado $offN_P$ (*)
Control	nop	Operación nula (no realiza ninguna acción)
	halt	Detiene la ejecución del procesador

Si bien no todas las instrucciones se encuentran implementadas (como es el caso de las instrucciones de entrada/salida), la mayoría están habilitadas para su uso, y son usadas casi todas ellas. Por otro lado, las instrucciones “dsubi” y “xnorr”, son las que debemos implementar en este proyecto y se detalla el cómo más adelante en el informe.

## Memoria

En cuanto a la organización de la memoria principal, la estructura está basada en el diseño de la memoria del simulador MSX88 como se puede apreciar en la siguiente tabla.

Sección de memoria	Dirección inicial	Dirección final	Tamaño total
Interrupciones y Entrada/Salida (*)	0000	0FFF	4 kilobytes
Datos	1000	1FFF	4 kilobytes
Instrucciones	2000	2FFF	4 kilobytes
Subrutinas (*)	3000	6FFF	16 kilobytes
Pila (*)	7000	7FFF	4 kilobytes
Sistema Operativo (*)	8000	FFFF	32 kilobytes
<b>Total</b>	<b>0000</b>	<b>FFFF</b>	<b>64 kilobytes</b>

## CICLO DE EJECUCIÓN

Para esta característica del procesador, como en algunos casos anteriores, se tomó como referencia el procesador que proporcionaba el simulador WinMIPS64, es decir: en el TDA-1819 se divide el ciclo de ejecución en 5 etapas concretas con tareas específicas con el fin de implementar la segmentación del cauce del WinMIPS64.

Etapa	Tareas a ejecutar (en orden secuencial)
1. Búsqueda	Acceder al registro IP y obtener a partir de él la dirección inicial en memoria principal de la próxima instrucción a ejecutar.
	Acceder a la dirección de memoria adquirida en el paso anterior y obtener a partir de ella y las contiguas todos los campos de la próxima instrucción.
	Acceder al registro IR y actualizarlo con el código de operación de la nueva instrucción obtenida en el paso anterior.
	Transmitir todos los campos de la nueva instrucción, excepto la cabecera y el código de operación, a la unidad de control de la CPU para que ésta pueda continuar con la siguiente etapa del ciclo de ejecución.
	Incrementar el registro IP a partir del valor contenido en el campo cabecera de la instrucción adquirida en el paso anterior para que apunte a la próxima instrucción a ejecutar.

	<p>Acceder al registro IR y obtener a partir de él el código de operación de la nueva instrucción a ejecutar, actualizado en la etapa anterior del ciclo de ejecución.</p>
2. Decodificación	<p>Decodificar el código de operación obtenido en el paso anterior e interpretar a partir de él la denominación de la nueva instrucción a ejecutar.</p>
	<p>En caso de ser necesario, acceder a los registros de uso general de la CPU asociados con los identificadores recibidos de la etapa anterior del ciclo de ejecución y obtener a partir de ellos los operandos de la nueva instrucción.</p>
	<p>Enviar los operandos obtenidos en el paso anterior a las unidades de la CPU encargadas de ejecutar las etapas restantes del ciclo de ejecución junto con la información necesaria para indicarle a cada una de ellas, en caso de corresponder, la acción a realizar cuando le llegue el turno de ejecutar su etapa asignada del ciclo de ejecución de la nueva instrucción.</p>

3.1. Ejecución	Determinar a partir de la información recibida de la etapa de
	decodificación, en caso de corresponder, la operación aritmético-lógica requerida por la nueva instrucción para ser ejecutada en esta etapa del ciclo de ejecución.
	Ejecutar la operación aritmético-lógica determinada en el paso anterior.
	Acceder al registro FLAGS y actualizar las banderas que deban ser modificadas como consecuencia de la ejecución de la operación aritmético-lógica en el paso anterior.
	En caso de tratarse la nueva instrucción de un salto condicional, acceder nuevamente al registro FLAGS y obtener a partir de él el valor actual de cada una de sus banderas.
	A partir de las banderas obtenidas en el paso anterior, analizar aquella que corresponda para determinar si el salto debe ser tomado o no.
	Si la respuesta determinada en el paso anterior fue afirmativa, actualizar el registro IP con la dirección de salto indicada en la nueva instrucción.
	Enviar el resultado de la operación aritmético-lógica obtenido en esta etapa a las unidades de la CPU encargadas de ejecutar las etapas restantes del ciclo de ejecución para que puedan utilizarlo en caso de requerirlo para llevar a cabo sus respectivas tareas.

	Determinar a partir de la información recibida de la etapa de decodificación, en caso de corresponder, la operación aritmético-lógica en punto flotante requerida por la nueva instrucción para ser ejecutada en esta etapa del ciclo de ejecución.
3.2. Ejecución en punto flotante	Ejecutar la operación aritmético-lógica en punto flotante determinada en el paso anterior.
	Acceder al registro FPFLAGS y actualizar las banderas que deban ser modificadas como consecuencia de la ejecución de la operación aritmético-lógica en punto flotante en el paso anterior.
	En caso de tratarse la nueva instrucción de una comparación, acceder nuevamente al registro FPFLAGS y obtener a partir de él el valor actual de cada una de sus banderas.
	A partir de las banderas obtenidas en el paso anterior, analizar aquélla que corresponda para determinar el valor con el cual deberá

	actualizarse la bandera F.
	Acceder una vez más al registro FPFLAGS y actualizar la bandera F con el valor determinado en el paso anterior.
	Enviar el resultado de la operación aritmético-lógica en punto flotante obtenido en esta etapa a las unidades de la CPU encargadas de ejecutar las etapas restantes del ciclo de ejecución para que puedan utilizarlo en caso de requerirlo para llevar a cabo sus respectivas tareas.

	Determinar a partir de la información recibida de la etapa de decodificación, en caso de corresponder, el tipo de acceso a memoria de la computadora requerido por la nueva instrucción para ser llevado a cabo en esta etapa del ciclo de ejecución.
4. Acceso a memoria	Acceder a la dirección de memoria de datos o el puerto del periférico de E/S determinado en el paso anterior para realizar sobre él una operación de lectura o escritura según corresponda.
	En caso de haberse tratado de una operación de lectura, enviar el valor obtenido a la unidad de almacenamiento en registro para que pueda utilizarlo en caso de requerirlo para ejecutar la última etapa del ciclo de ejecución de la nueva instrucción.

	Determinar a partir de la información recibida de la etapa de decodificación, en caso de corresponder, el tipo de acceso a registro de uso general de la CPU requerido por la nueva instrucción para ser ejecutada en esta etapa del ciclo de ejecución.
5. Almacenamiento en registro	Acceder al registro de uso general determinado en el paso anterior y actualizarlo con el valor que corresponda según lo indicado en la instrucción actual. Dicho valor habrá sido recibido de alguna de las etapas anteriores del ciclo de ejecución: decodificación, ejecución/ejecución en punto flotante o acceso a memoria en función de la instrucción ejecutada.

Etapa	Unidad de la CPU encargada de su ejecución
1. Búsqueda	Unidad de búsqueda
2. Decodificación	Unidad de control
3.1. Ejecución	Unidad aritmético-lógica (ALU)
3.2. Ejecución en punto flotante	Unidad de punto flotante (FPU)
4. Acceso a memoria	Unidad de acceso a memoria
5. Almacenamiento en registro	Unidad de almacenamiento en registro

## Prueba de funcionamiento

Para analizar el funcionamiento del procesador, procedo a realizar la prueba, realizó la carga del *mem\_wb\_str\_wawl.asm*. Teniendo en cuenta el tamaño de cada instrucción indicado anteriormente, al costado de cada una colocamos un comentario representando la dirección de memoria (en hexadecimal) en la que debería encontrarse la instrucción (o dato).

```
.data
A: .float 5.0 ;1000
B: .float 6.0 ;1004
C: .word 20 ;1008
D: .word 6 ;100C
.code
lf f1, A(r0) ;2000
lf f2, B(r0) ;2006
lw r1, C(r0) ;200C
lw r2, D(r0) ;2012
nop ;2018
addf f3, f1, f2 ;201A
mulf f4, f1, f2 ;201F
lf f3, A(r0) ;2024
dsub r4, r1, r2 ;202A
addf f5, f1, f2 ;202F
ddiv r6, r1, r2 ;2034
sw r1, D(r2) ;2039
halt ;203F
```

Viendo los datos cargados en memoria, puede comprobarse que los mismos se guardan de la forma esperada (debe tenerse en cuenta que aquí las direcciones se encuentran en decimal, y

que cada posición en memoria guarda 1 byte).

<code>bx Data_Memory</code>	Limit	Limit
<code>bx Data_Memory[4096]</code>	0	0
<code>bx Data_Memory[4097]</code>	0	0
<code>bx Data_Memory[4098]</code>	160	160
<code>bx Data_Memory[4099]</code>	64	64
<code>bx Data_Memory[4100]</code>	0	0
<code>bx Data_Memory[4101]</code>	0	0
<code>bx Data_Memory[4102]</code>	192	192
<code>bx Data_Memory[4103]</code>	64	64
<code>bx Data_Memory[4104]</code>	20	20
<code>bx Data_Memory[4105]</code>	0	0
<code>bx Data_Memory[4106]</code>	0	0
<code>bx Data_Memory[4107]</code>	0	0
<code>bx Data_Memory[4108]</code>	?	6
<code>bx Data_Memory[4109]</code>	?	0
<code>bx Data_Memory[4110]</code>	?	0
<code>bx Data_Memory[4111]</code>	?	0

las instrucciones se guardarán en memoria de la siguiente manera:

- If f1, A(r0):

- Cabecera: un byte que indica la cantidad de bytes de la instrucción, incluyéndose. En este caso debería guardar 00000110b(6 decimal)
- Cod. de operación: en este caso, es 00001010b (10 decimal).
- Registro destino: en este caso es f1, cuyo ID en decimal es 17 (00010001b)
- Byte 0 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta A (1000h), contendrá 00000000b (0 decimal).
- Byte 1 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta A (1000h), contendrá 00010000b (16 decimal).
- Registro índice: en este caso es r0, cuyo ID en decimal es 0 (00000000b)

- If f2, B(r0):

- Cabecera: en este caso debería guardar 00000110b (6 decimal)
- Cod. de operación: en este caso, es 00001010b (10 decimal).
- Registro destino: en este caso es f2, cuyo ID en decimal es 18 (00010010b)
- Byte 0 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta B (1004h), contendrá 00000100b (4 decimal).
- Byte 1 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta B (1004h), contendrá 00010000b (16 decimal).
- Registro índice: en este caso es r0, cuyo ID en decimal es 0 (00000000b)

- lw r1, C(r0):

- Cabecera: en este caso debería guardar 00000110b (6 decimal)
- Cod. de operación: en este caso, es 00001000b (8 decimal).
- Registro destino: en este caso es r1, cuyo ID en decimal es 1 (00000001b)

- Byte 0 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta C (1008h), contendrá 00001000b (8 decimal).
- Byte 1 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta C (1008h), contendrá 00010000b (16 decimal).
- Registro índice: en este caso es r0, cuyo ID en decimal es 0 (00000000b)

- lw r2, D(r0):
  - Cabecera: en este caso debería guardar 00000110b (6 decimal)
  - Cod. de operación: en este caso, es 00001000b (8 decimal).
  - Registro destino: en este caso es r2, cuyo ID en decimal es 2 (00000010b)
  - Byte 0 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta D (100Ch), contendrá 00001100b (12 decimal).
  - Byte 1 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta D (100Ch), contendrá 00010000b (16 decimal).
  - Registro índice: en este caso es r0, cuyo ID en decimal es 0 (00000000b)

- nop:
  - Cabecera: en este caso debería guardar 00000010b (2 decimal)
  - Cod. de operación: en este caso, es 10000000b (128 decimal).

- addf f3, f1, f2:
  - Cabecera: en este caso debería guardar 00000101b (5 decimal)
  - Cod. de operación: en este caso, es 00011100b (28 decimal).
  - Registro destino: en este caso es f3, cuyo ID en decimal es 19 (00010011b)
  - Primer registro fuente: en este caso es f1, cuyo ID en decimal es 17 (00010001b)
  - Segundo registro fuente: en este caso es f2, cuyo ID en decimal es 18 (00010010b)

- mulf f4, f1, f2:
  - Cabecera: en este caso debería guardar 00000101b (5 decimal)
  - Cod. de operación: en este caso, es 00100010b (34 decimal).
  - Registro destino: en este caso es f4, cuyo ID en decimal es 20 (00010100b)
  - Primer registro fuente: en este caso es f1, cuyo ID en decimal es 17 (00010001b)
  - Segundo registro fuente: en este caso es f2, cuyo ID en decimal es 18 (00010010b)

- lf f3, A(r0):
  - Cabecera: en este caso debería guardar 00000110b (6 decimal)
  - Cod. de operación: en este caso, es 00001010b (10 decimal).
  - Registro destino: en este caso es f3, cuyo ID en decimal es 19 (00010011b)
  - Byte 0 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta A (1000h), contendrá 00000000b (0 decimal).
  - Byte 1 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta A (1000h), contendrá 00010000b (16 decimal).
  - Registro índice: en este caso es r0, cuyo ID en decimal es 0 (00000000b)

- dsub r4, r1, r2:

- Cabecera: en este caso debería guardar 00000110b (6 decimal)
- Cod. de operación: en este caso, es 00011101b (29 decimal).
- Registro destino: en este caso es f2, cuyo ID en decimal es 18 (00010010b)
- Byte 0 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta B (1004h), contendrá 00000100b (4 decimal).
- Byte 1 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta B (1004h), contendrá 00010000b (16 decimal).
- Registro índice: en este caso es r0, cuyo ID en decimal es 0 (00000000b)

- addf f5, f1, f2:

- Cabecera: en este caso debería guardar 00000101b (5 decimal)
- Cod. de operación: en este caso, es 00011100b (28 decimal).
- Registro destino: en este caso es f5, cuyo ID en decimal es 21 (00010101b)
- Primer registro fuente: en este caso es f1, cuyo ID en decimal es 17 (00010001b)
- Segundo registro fuente: en este caso es f2, cuyo ID en decimal es 18 (00010010b)

- ddiv r6, r1, r2:

- Cabecera: en este caso debería guardar 00000101b (5 decimal)
- Cod. de operación: en este caso, es 00100011b (35 decimal).
- Registro destino: en este caso es r6, cuyo ID en decimal es 6 (00000110b)
- Primer registro fuente: en este caso es r1, cuyo ID en decimal es 1 (00000001b)
- Segundo registro fuente: en este caso es r2, cuyo ID en decimal es 2 (00000010b)

- sw r1, D(r2):

- Cabecera: en este caso debería guardar 00000110b (6 decimal)
- Cod. de operación: en este caso, es 00001001b (9 decimal).
- Registro fuente: en este caso es r1, cuyo ID en decimal es 1 (00000001b)
- Byte 0 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta D (100Ch), contendrá 00001100b (12 decimal).
- Byte 1 inmediato: en este caso, como la dirección a la que hace referencia es a la que tiene la etiqueta D (100Ch), contendrá 00010000b (16 decimal).
- Registro índice: en este caso es r2, cuyo ID en decimal es 2 (00000010b)

- halt:

- Cabecera: en este caso debería guardar 00000010b (2 decimal)
- Cod. de operación: en este caso, es 10000001b (129 decimal).

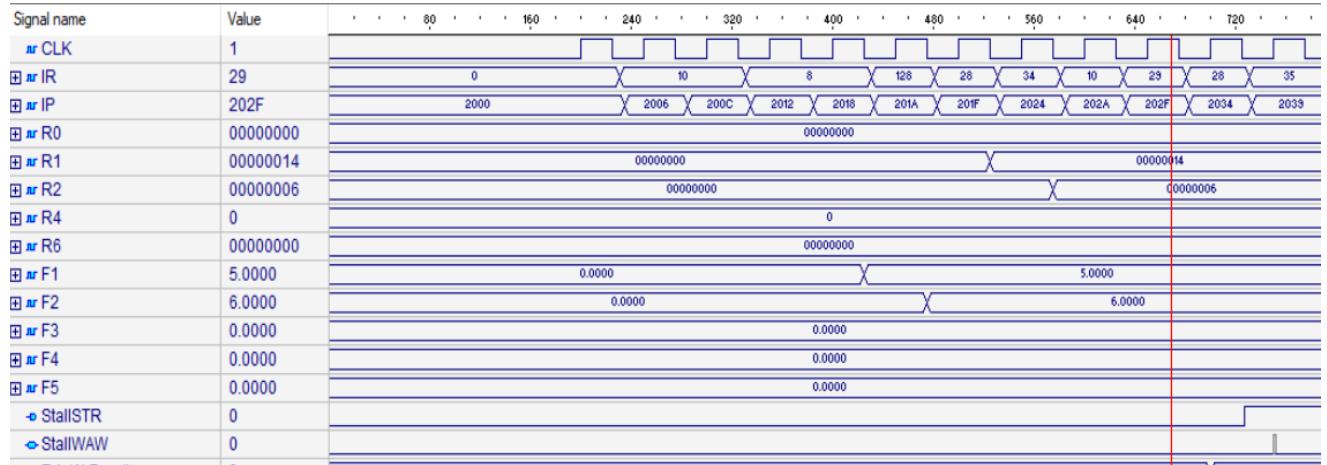
En las siguientes imágenes, puede comprobarse que las instrucciones fueron almacenadas en memoria (los datos guardados se encuentran en decimal para mayor legibilidad) :

Signal name	Value	240	320	400	480	560	640	720	800
⊕ ar_Inst_Memory[8192]	6				6				
⊕ ar_Inst_Memory[8193]	10				10				
⊕ ar_Inst_Memory[8194]	17				17				
⊕ ar_Inst_Memory[8195]	0				0				
⊕ ar_Inst_Memory[8196]	16				16				
⊕ ar_Inst_Memory[8197]	0				0				
⊕ ar_Inst_Memory[8198]	6				6				
⊕ ar_Inst_Memory[8199]	10				10				
⊕ ar_Inst_Memory[8200]	18				18				
⊕ ar_Inst_Memory[8201]	4				4				
⊕ ar_Inst_Memory[8202]	16				16				
⊕ ar_Inst_Memory[8203]	0				0				
⊕ ar_Inst_Memory[8204]	6				6				
⊕ ar_Inst_Memory[8205]	8				8				
⊕ ar_Inst_Memory[8206]	1				1				
⊕ ar_Inst_Memory[8207]	8				8				
⊕ ar_Inst_Memory[8208]	16				16				
⊕ ar_Inst_Memory[8209]	0				0				
⊕ ar_Inst_Memory[8210]	6				6				
⊕ ar_Inst_Memory[8211]	8				8				
⊕ ar_Inst_Memory[8212]	2				2				
⊕ ar_Inst_Memory[8213]	12				12				
⊕ ar_Inst_Memory[8214]	16				16				
⊕ ar_Inst_Memory[8215]	0				0				
⊕ ar_Inst_Memory[8216]	2				2				
⊕ ar_Inst_Memory[8217]	128				128				
⊕ ar_Inst_Memory[8218]	5				5				
⊕ ar_Inst_Memory[8219]	28				28				
⊕ ar_Inst_Memory[8220]	19				19				
⊕ ar_Inst_Memory[8221]	17				17				
⊕ ar_Inst_Memory[8222]	18				18				
⊕ ar_Inst_Memory[8223]	5				5				
⊕ ar_Inst_Memory[8224]	34				34				
⊕ ar_Inst_Memory[8225]	20				20				
⊕ ar_Inst_Memory[8226]	17				17				
⊕ ar_Inst_Memory[8227]	18				18				
⊕ ar_Inst_Memory[8228]	6				6				
⊕ ar_Inst_Memory[8229]	10				10				
⊕ ar_Inst_Memory[8230]	19				19				
⊕ ar_Inst_Memory[8231]	0				0				
⊕ ar_Inst_Memory[8232]	16				16				
⊕ ar_Inst_Memory[8233]	0				0				
⊕ ar_Inst_Memory[8234]	5				5				
⊕ ar_Inst_Memory[8235]	29				29				
⊕ ar_Inst_Memory[8236]	4				4				
⊕ ar_Inst_Memory[8237]	1				1				
⊕ ar_Inst_Memory[8238]	2				2				
⊕ ar_Inst_Memory[8239]	5				5				
⊕ ar_Inst_Memory[8240]	28				28				
⊕ ar_Inst_Memory[8241]	21				21				
⊕ ar_Inst_Memory[8242]	17				17				
⊕ ar_Inst_Memory[8243]	18				18				
⊕ ar_Inst_Memory[8244]	5				5				
⊕ ar_Inst_Memory[8245]	35				35				
⊕ ar_Inst_Memory[8246]	6				6				
⊕ ar_Inst_Memory[8247]	1				1				
⊕ ar_Inst_Memory[8248]	2				2				
⊕ ar_Inst_Memory[8249]	6				6				
⊕ ar_Inst_Memory[8250]	9				9				
⊕ ar_Inst_Memory[8251]	1				1				
⊕ ar_Inst_Memory[8252]	12				12				
⊕ ar_Inst_Memory[8253]	16				16				
⊕ ar_Inst_Memory[8254]	2				2				
⊕ ar_Inst_Memory[8255]	2				2				
⊕ ar_Inst_Memory[8256]	129				129				
⊕ ar_Inst_Memory[8257]	?				?				
⊕ ar_Inst_Memory[8258]	?				?				

En la siguiente imagen, puede verse el resultado de la simulación del programa.

El registro IP comenzará con el valor 2000 (inicio de la memoria de instrucciones), para luego ir tomando las direcciones de las próximas ejecuciones a ejecutar (respetándose el orden indicado en el código de más arriba).

Como en este programa no hay ningún salto, el registro IP se actualizará en cada etapa Fetch. Por otro lado, puede verse en el registro IR el código de operación (en decimal), de la instrucción que está siendo ejecutada en ese momento. Este registro, al igual que el anterior, se actualiza en la etapa Fetch.



Puede verse también como los registros f1, f2, r1 y r2 son cargados con los valores correctos y en el orden correcto, según el programa. Luego es posible ver dos señales de atascos: primero uno de tipo estructural, y luego otro de dependencias de datos, específicamente de tipo WAW.

El primero se debe a que tanto la instrucción addf f3, f1, f2 como la instrucción lf f3, A(r0) están intentando acceder a la etapa Memory Access. Esto se debe a que la primera tiene una etapa de ejecución de cuatro pasos, por ser una operación de punto flotante; mientras que la instrucción lf sólo cuenta con una etapa de ejecución. Por lo tanto, como la instrucción addf entró primero al cauce, esta tiene el “derecho” de entrar a la etapa Memory Access primero. El otro atasco es debido a que estas dos mismas instrucciones modifican al registro f3. Por lo tanto, por lo dicho anteriormente, al tener lf 5 etapas y addf 8 (por ser una operación en punto flotante), lf terminaría primero. Pero aquí, aparte del atasco estructural dicho anteriormente, ocurre el WAW, ya que lf escribirá f3 antes de que addf guarde su resultado en el mismo, habiendo esta última entrado primero al cauce. Por estos inconvenientes, el cauce se detendrá (en la instrucción ddiv específicamente) hasta que los atascos se resuelvan.

Signal name	Value	640	720	800	880	960	1040	1120	1200			
✓ CLK	0											
✗ IR	129	10	23	28	35	9	123					
✗ IP	2041	202A	X	202F	X	2034	X	2033	X	203F	X	2041
✗ R0	00000000					00000000						
✗ R1	00000014					00000014						
✗ R2	00000006					00000006						
✗ R4	14			0		X		14				
✗ R6	00000003				00000000		X		00000003			
✗ F1	5.0000					5.0000						
✗ F2	6.0000					6.0000						
✗ F3	5.0000		0.0000	X	11.000	X		5.0000				
✗ F4	30.000			0.0000	X			30.000				
✗ F5	11.000					0.0000		X	11.000			
✗ StallSTR	0											
✗ StallWAW	0											

Como puede verse, los atascos se resuelven satisfactoriamente, ya que f3 primero toma el valor 11.00, resultado de la instrucción addf, luego f4 toma el valor 30.00, resultado de la instrucción mulf, y luego de esto f3 toma el valor que le asigna la instrucción lf, siendo el mismo 5.00. Después de esto, el registro r4 toma el valor 14, debido a la instrucción dsub; el registro r6 toma el valor 3, producto de la instrucción ddiv (en realidad el resultado es 10/3, pero al ser un registro de punto fijo, en vez de almacenarse 3.33, se almacena 3). Cabe aclarar que esta instrucción terminó antes que su predecesora, la instrucción addf f5, f1, f2, ya que esta última es una operación de punto flotante. Por esta misma razón, la última instrucción antes de halt, sw r1, D(r2), también almacena su resultado (en la etapa Memory Access) antes de que el resultado de addf se vea reflejado en f5. Luego, se ejecuta la sentencia halt, terminando la simulación.

Por lo tanto, los resultados de la simulación fueron los esperados.

## dsubi rd, rf, N

Resta el valor inmediato N al registro rf, dejando el resultado en el registro rd (valores con signo)

## xnorr rd, rf, rg

Realiza un XNOR entre los registros rf y rg (bit a bit), dejando el resultado en el registro rd

Lo primero que hice para poder realizar las instrucciones DSUBI y XNORR fue ver cómo funcionaba el proyecto en general, qué señales eran de importancia y cuáles me podrían llegar a interesar, como también qué datos de “Packages” o constantes eran similares a lo que había que hacer, para luego agregar o modificar estos según mis necesidades. Lo primero que me di cuenta es que en “const\_ensamblador” se encontraban todas las instrucciones soportadas por el TDA18\_19, junto con

su código de operación y el tamaño total de cada instrucción en memoria, por lo tanto decidí agregar las constantes:

```
CONSTANT INSTAR_NAMES:           instar_name_array(1 to CANT_INSTAR) :=  
("dadd ", "daddi ", "daddu ", "daddui", "addf ", "dsub ", "dsu  
,"dsubu ", "subf ", "dmul ", "dmulu ", "mulf  
,"ddiv ", "ddivu ", "divf ", "slt ", "slti ", "ltf ", "lef ", "eqf ", "neg ");
```

```
CONSTANT INSTAR_CODES:          instar_code_array(1 to CANT_INSTAR) := (DADD,  
DADDI, DADDU, DADDUI, ADDF, DSUB, DSUBU, SUBF, DMUL, DMULU, MULF, DDIV,  
DDIVU, DIVF, SLT, SLTI, LTF, LEF, EQF, NEGR);
```

```
CONSTANT INSTAR_SIZES:          instar_size_array(1 to CANT_INSTAR) := (5, 8, 5,  
8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 8, 4, 4, 4, 4);
```

```
CONSTANT INSTLD_NAMES:          instld_name_array(1 to CANT_INSTLD) :=  
("and ", "andi ", "or ", "ori ", "xor ", "xori ", "not ", "notr ", "dsl ", "dsli ", "dsr ", "dsri ", "dls ", "dlsi ",  
"dsrs ", "dsrsi ", "xnorr");
```

```
CONSTANT INSTLD_CODES:          instld_code_array(1 to CANT_INSTLD) :=  
(ANDR, ANDI, ORR, ORI, XORR, XORI, NOTR, DSL, DSLI, DSR, DSRI, DSLS, DSLSI, DSRS,  
DSRSI, XNORR);
```

```
CONSTANT INSTLD_SIZES:          instld_size_array(1 to CANT_INSTLD) := (5, 8, 5,  
8, 5, 8, 5, 5, 8, 5, 8, 5, 8, 5, 8, 8);
```

```

CONSTANT INSTAR_NAMES:           instar_name_array(1 to CANT_INSTAR) := ("dadd ", "daddi ", "daddu ", "daddui", "addf ", "dsub ",  
CONSTANT INSTAR_CODES:          instar_code_array(1 to CANT_INSTAR) := (DADD, DADDI, DADDU, DADDUI, ADDF, DSUB, DSUBU, SUBF, DMUL,  
CONSTANT INSTAR_SIZES:          instar_size_array(1 to CANT_INSTAR) := (5, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 8, 4, 4, 4, 4)  
CONSTANT INSTLD_NAMES:          instld_name_array(1 to CANT_INSTLD) := ("and ", "andi ", "or ", "ori ", "xor ", "xori ", "not ",  
CONSTANT INSTLD_CODES:          instld_code_array(1 to CANT_INSTLD) := (ANDR, ANDI, ORR, ORI, XORR, XORI, NOTR, DSL, DSLI, DSR, DS  
CONSTANT INSTLD_SIZES:          instld_size_array(1 to CANT_INSTLD) := (5, 8, 5, 8, 5, 8, 5, 5, 5, 8, 5, 8, 5, 8, 5, 8, 8);  
  
CONSTANT INSTTC_NAMES:          insttc_name_array(1 to CANT_INSTTC) := ("jmp ", "beq ", "bne ", "beqz", "bnez", "bfpt", "bfpf");  
CONSTANT INSTTC_CODES:          insttc_code_array(1 to CANT_INSTTC) := (JMP, BEQ, BNE, BEQZ, BNEZ, BFPT, BFPF);  
CONSTANT INSTTC_SIZES:          insttc_size_array(1 to CANT_INSTTC) := (4, 6, 6, 5, 5, 4, 4);  
  
CONSTANT INCTOT_NAMES:          const_name_array(1 to CANT_INCTOT) := (None, None);

```

Console output:

```
# Compile success 0 Errors 0 Warnings Analysis time : 16.0 [ms]
acom -O3 -e 100 -work TDA_1819 -2002 sdsn/src/Packages/const_ensamblador.vhd
# Warning: DAGGEN_0523: The source is compiled without the -dbg switch. Line breakpoints and assertion debug will not be available.
# File: C:/My_Designs/TDA_1819/TDA_1819/src/Packages/const_ensamblador.vhd
# Compile Package "const_ensamblador"
# Compile Package Body "const_ensamblador"
# Compile success 0 Errors 0 Warnings Analysis time : 31.0 [ms]
```

Por lo que también tuve que modificar “tipos\_ensamblador”

gn Browser

gistros (REGISTROS\_ARCHITECTU... ▾

sorted

- report\_cpu.vhd
- tipos\_ascii.vhd
- const\_ascii.vhd
- func\_ensamblador.vhd
- const\_ensamblador.vhd
- Usuario

  - usuario.vhd
  - PC

    - pc.vhd
    - 1. Ensamblador

      - ensamblador.vhd

    - 2. CPU

      - cpu.vhd
      - 1. UI
      - 2. Etapas

        - etapas.vhd

      - 1. Fetch
      - 2. Decode

        - decode.vhd

      - 3. Execute

Structure Resource

```

16
17 PACKAGE tipos_ensamblador is
18
19   TYPE data_name_array IS ARRAY (POSITIVE RANGE <>) OF STRING(1 to 7);
20   TYPE data_type_array IS ARRAY (POSITIVE RANGE <>) OF INTEGER;
21   TYPE data_size_array IS ARRAY (POSITIVE RANGE <>) OF INTEGER;
22
23   TYPE insttd_name_array IS ARRAY (POSITIVE RANGE <>) OF STRING(1 to 3);
24   TYPE insttd_code_array IS ARRAY (POSITIVE RANGE <>) OF STD_LOGIC_VECTOR(7 downto 0);
25   TYPE insttd_size_array IS ARRAY (POSITIVE RANGE <>) OF INTEGER;
26
27   TYPE instar_name_array IS ARRAY (POSITIVE RANGE <>) OF STRING(1 to 6);
28   TYPE instar_code_array IS ARRAY (POSITIVE RANGE <>) OF STD_LOGIC_VECTOR(7 downto 0);
29   TYPE instar_size_array IS ARRAY (POSITIVE RANGE <>) OF INTEGER;
30
31   TYPE instld_name_array IS ARRAY (POSITIVE RANGE <>) OF STRING(1 to 5);
32   TYPE instld_code_array IS ARRAY (POSITIVE RANGE <>) OF STD_LOGIC_VECTOR(7 downto 0);
33   TYPE instld_size_array IS ARRAY (POSITIVE RANGE <>) OF INTEGER;
34
35   TYPE insttc_name_array IS ARRAY (POSITIVE RANGE <>) OF STRING(1 to 4);
36   TYPE insttc_code_array IS ARRAY (POSITIVE RANGE <>) OF STD_LOGIC_VECTOR(7 downto 0);
37   TYPE insttc_size_array IS ARRAY (POSITIVE RANGE <>) OF INTEGER;
38
39   TYPE instct_name_array IS ARRAY (POSITIVE RANGE <>) OF STRING(1 to 4);
40   TYPE instct_code_array IS ARRAY (POSITIVE RANGE <>) OF STD_LOGIC_VECTOR(7 downto 0);
41

```

sole

file: C:/My Designs/TDA\_1819/TDA\_1819/src/Usuario/PC/3. Buses\_Admin/buses\_admin.vhd

Actualizo el report\_cpu

```

92
93   CONSTANT NEGR:
94     -- NEG rd, rf
95     CONSTANT NEGR:           STD_LOGIC_VECTOR(7 downto 0) := "00101011";
96   CONSTANT DSUBI:
97     -- DSUBI rd, rf, N
98     CONSTANT DSUBI:          STD_LOGIC_VECTOR(7 downto 0) := "00101100";
99
100
101   CONSTANT NOTR:
102     --xnorr rd, rf, rg
103     CONSTANT NOTR:           STD_LOGIC_VECTOR(7 downto 0) := "00110110";
104   CONSTANT XNORR:
105     --xnorr rd, rf, rg
106     CONSTANT XNORR:          STD_LOGIC_VECTOR(7 downto 0) := "00110111";
107
108   -- Desplazamiento de bits
109

```

De modificar el Ensamblador, proseguí a modificar el Decode, ya que en esta etapa el procesador se encargará de decodificar cuál será la operación a realizar, por ello incorpore lo que el procesador deberá hacer luego de determinar qué se trata de una instrucción DSUBI o XNORR.

Se observó claramente que había un CASE con diferentes WHEN y que cada uno de ellos corresponden a una instrucción del procesador, por lo que hicimos lo mismo con las instrucciones a implementar en este trabajo.

## PROGRAMA ASSEMBLER

Rango de representar un número con signo de 16 bits

-32.768 a + 32.768

numero de alumno 01789/0 → 17890 → 0100 0101 1110 0010

dni 40656988 → 6988 → 0001 1011 0100 1100

.data

A: .hword 010001011100010

B: .hword 0001101101001100  
C: .word 10101110  
D: .word 1011010111001001 // B5C9  
E: .word 00000001

```
.subr
subr1:
dsub r3,r1,r2
pushh r3
slt r4,r3,r0
bne r4, r7 ,mayorUno
lb r6, C(r0)
xnorr r5,r3,r6
jmp finalUno
mayorUno: lh r6, D(r0)
xnorr r5, r3, r6
finalUno: pushh r5
ret
```

```
subr1:
dsub r3,r2,r1
pushh r3
slt r4,r3,r0
bne r4, r7 ,mayorDos
lb r6, C(r0)
xnorr r5,r3,r6
jmp finalDos
```

```
mayorDos: lh r6, D(r0)
xnorr r5, r3, r6
finalDos: pushh r5
ret
```

```
.code
lh r1, A(r0)
lh r2, B(r0)
lb r7, E(r0)
call subr1
call subr2
halt
```

---

El programa cuenta con la sección de datos donde:  
-A → número de alumno explicado anteriormente.  
-B → numero de documento.  
-C →número para hacer el xnorr en caso negativo.  
-D →número xnorr en caso positivo, B5C9 codificado en binario.

-E número 1.

En el registro R1 guardo A(Número Alumno), en el registro R2 guardo B(Número DNI), en r7 guardo en número 1, para después poder comparar.

Llamó a la subr1

Con dsub guardo en r3 la resta de r2 a r1, hago un push de r3.

Luego hago un slt entre el resultado r3 y el número 0(r0), y para después comparar ese resultado con un bne y si es igual a uno, nos da que el r4 es negativo.

Si r4 es mayor:

En r6 guardo el 1011010111001001, y realizó el xnorr implementado en el ejercicio1 del tp,

Si r4 es menor:

En r6 guardo 10101110, y realizó el xnorr.

Sea mayor o menor, se queda el resultado en r5 y luego hago un push del mismo.

Luego llamó a la subr2, que en lo único que cambia es:

Con dsub guardo en r3 la resta de r1 a r2, hago un push de r3.

Realiza los mismos procesos que en la subr1.

Al ser, en gran parte, un procesador de arquitectura RISC posee implementada la metodología de segmentación del cauce con el fin de acelerar y optimizar la labor a realizar.

Se consideran las cinco etapas que constituyen el ciclo de ejecución de la implementación tradicional de la segmentación del cauce y estas son:

- \* Etapa de búsqueda - Fetch
- \* Etapa de decodificación - Decode
- \* Etapa de ejecución - Execute
- \* Etapa de acceso a memoria - Memory access
- \* Etapa de escritura - Writeback

Decode : obtiene del Registro de Instrucción (IR) el código de operación de la instrucción a ejecutar, lo interpreta y en función del resultado procederá a obtener la información sobre los operandos a partir de las señales recibidas de la etapa de búsqueda. En caso de ser necesario, acceder a los registros de uso general de la CPU asociados con los identificadores recibidos de la etapa anterior del ciclo de ejecución. Las señales que son porteadas a esta etapa y son de nuestro interés son:

- \* IDtoMA -- Esta señal nos va a permitir acceder a memoria si es necesario.
- \* IDtoWB -- Esta señal nos va a permitir acceder a la etapa de writeback en el caso de que sea necesario.
- \* IDRegID -- Esta señal nos permite identificar el registro sobre el cual se quiere leer en la etapa decode
- \* SizeRegID -- El tamaño de lo que queremos leer
- \* EnableRegID -- Señal que nos va a permitir realizar la lectura, cuando hay un flanco de subida se realiza la lectura.
- \* DataRegInID -- Lo que voy a escribir en el registro que le mande el IDRegId
- \* DataRegOutID -- Va a contener lo que lee del registro que le mande el IDReigId

La señal CodOp va a ser la señal por la cual el decode va a reconocer de qué instrucción se trata, mediante un case a esta variable, va a reconocer todos los tipos de instrucciones en la etapa de decode.

Lo que hice fue agregar ambas instrucciones tanto XNORR como DSUBI, a este CASE WHEN y dependiendo de cual fuera la instrucción, realiza algunas operaciones u otras.