Trabajo Práctico 10 - Taller de Arquitectura

Procesador TDA 1819



Ringuelet Pedro

02960/7

**Índice**

[1. Introducción 3](#_Toc182561163)

[2. Implementación de instrucciones 4](#_Toc182561164)

[2.1 DSUBI 4](#_Toc182561165)

[2.2 XNORR 5](#_Toc182561166)

[3. Desarrollo de programa de prueba 10](#_Toc182561167)

[4. Análisis de Resultados 12](#_Toc182561168)

[4.1 Pipeline desactivado 12](#_Toc182561169)

[4.2 Pipeline activado 14](#_Toc182561170)

[5. Conclusión 16](#_Toc182561171)

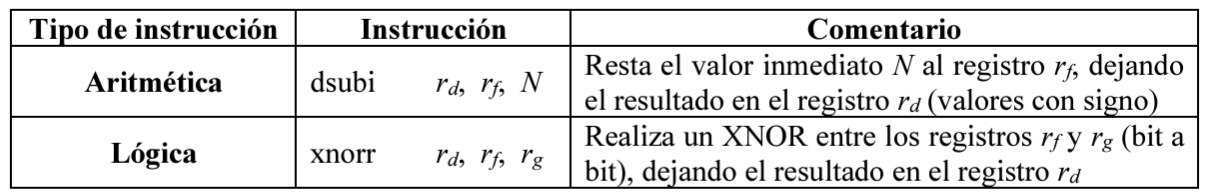
# 1. Introducción

Este trabajo práctico tiene como objetivo aplicar de manera integral los conceptos de diseño de procesadores y ensamblador adquiridos en la materia, utilizando el procesador específico de la cátedra: el TDA 1819. A través de la implementación de dos nuevas instrucciones, una aritmética (dsubi) y otra lógica (xnorr), se profundizará en la gestión y simulación de componentes de hardware.

La instrucción dsubi realiza una resta de un valor inmediato a un registro y almacena el resultado en otro registro, mientras que la instrucción xnorr ejecuta una operación lógica XNOR entre dos registros. La integración de estas instrucciones implica definir sus códigos de operación y detallar el comportamiento del procesador en cada etapa de ejecución.

Para validar la implementación, se desarrollará un programa en ensamblador que utilizará valores específicos (29607 y 6247, obtenidos del número de legajo y del DNI del autor) para realizar operaciones de resta y XNOR. Dependiendo de si el resultado de la resta es positivo o negativo, se aplicará la operación XNOR con un valor de referencia (10101110 o B5C9). Los resultados de estas operaciones se almacenarán en la memoria principal.

Además, se llevará a cabo una simulación del funcionamiento del procesador tanto en un entorno segmentado como no segmentado, para analizar los tiempos de ejecución de las instrucciones y el desempeño del procesador. Finalmente, se responderán preguntas analizando el flujo de las instrucciones en la arquitectura, detallando las etapas del ciclo de ejecución y los tiempos asociados.

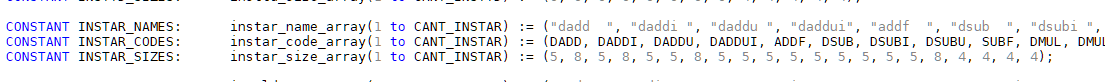


***Tabla 1*:** Instrucciones a implementar

# 2. Implementación de instrucciones

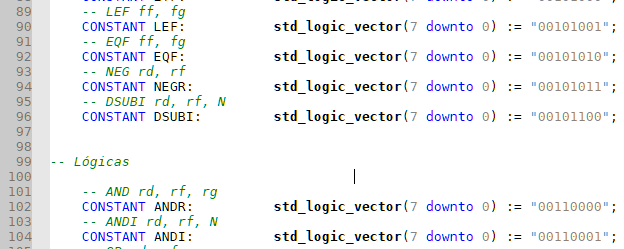
## 2.1 DSUBI

Cada instrucción en el procesador se activa mediante un código de operación específico. Para la implementación de **dsubi**, primero añadí su código en el archivo const\_ensamblador.vhd, incluyendo **"dsubi"** en la constante INSTAR\_NAME, **DSUBI** en INSTAR\_CODES, y finalmente su tamaño correspondiente en INSTAR\_SIZES, como se muestra en la Figura 1.



***Figura 1*:** Código de operación dsubi en const\_ensamblador.vhd

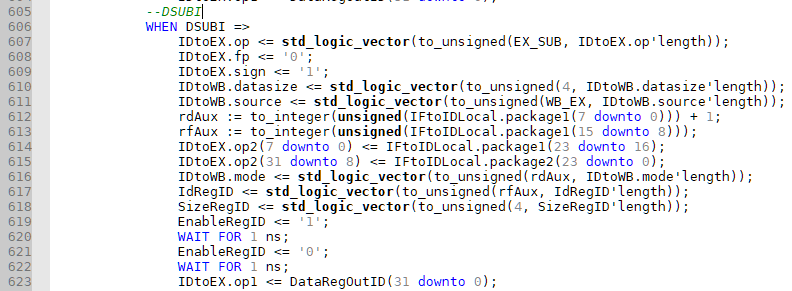
Luego, incorporé la instrucción en el repertorio modificando repert\_cpu.vhd. Dado que las operaciones aritméticas abarcan desde el código **"00011000"** hasta **"00101011"** y las operaciones lógicas empiezan en **"00110000"**, asigné el código **"00101100"** a **DSUBI**, como se muestra en la Figura 2.



***Figura 2*:** Incorporación de la instrucción dsubi en repert\_cpu.vhd

Después de asignar los códigos de operación, procedí a su implementación en el ciclo de instrucción. La primera etapa, que consiste en cargar el código de operación sin importar la instrucción, es común a todas las instrucciones y no requiere modificaciones. Sin embargo, en la etapa de decodificación, asigné una acción específica para los nuevos códigos de operación. Según el manual del procesador, en esta etapa se debe “acceder al registro IR, decodificar el código de operación de la instrucción y, si es necesario, obtener los operandos de la CPU para enviarlos a las unidades correspondientes para su ejecución” (Manual, p. 83).

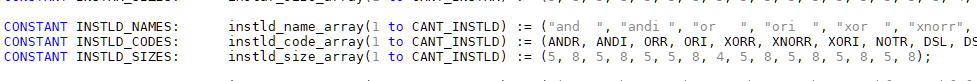
Es necesario adaptar la decodificación para incluir las nuevas instrucciones. Las etapas posteriores del ciclo de instrucción siguen el principio de "determinar la operación aritmético-lógica en función de la información de la decodificación", lo que facilita la ejecución y almacenamiento sin requerir grandes cambios, limitándose a enviar el código de operación correcto a la ALU. Para implementar la instrucción en decode.vhd, me basé en la instrucción **daddi** que utiliza los mismos operandos, reemplazando EX\_ADD con EX\_SUB en la etapa de ejecución, como se observa en la Figura 3.



***Figura 3*:** Incorporación de la instrucción dsubi en decode.vhd

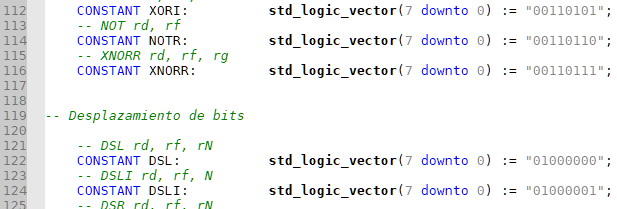
## 2.2 XNORR

Como se mencionó en la implementación de **DSUBI**, cada instrucción requiere un código de operación específico. Para **XNORR**, añadí su código en const\_ensamblador.vhd, incluyendo **"xnorr"** en INSTLD\_NAME, **XNORR** en INSTLD\_CODES, y su tamaño en INSTLD\_SIZES, como se muestra en la Figura 4.



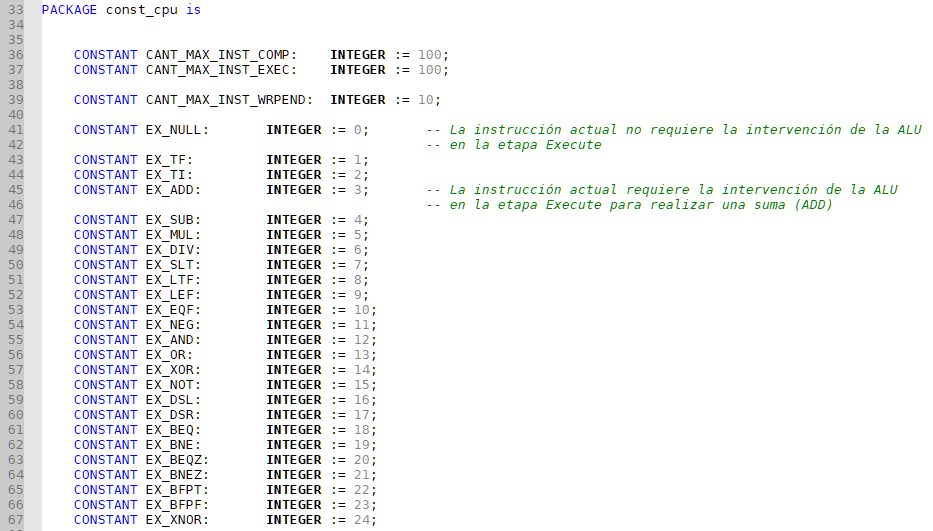
***Figura 4*:** Codigo de operación xnorr en const\_ensamblador.vhd

Posteriormente, incorporé la instrucción en el repertorio, modificando repert\_cpu.vhd. Las operaciones lógicas ocupan desde **"00110000"** hasta **"00110111"** y las operaciones de desplazamiento inician en **"01000000"**. Por lo tanto, asigné a **XNORR** el código **"00110111"**, como se ilustra en la Figura 5.



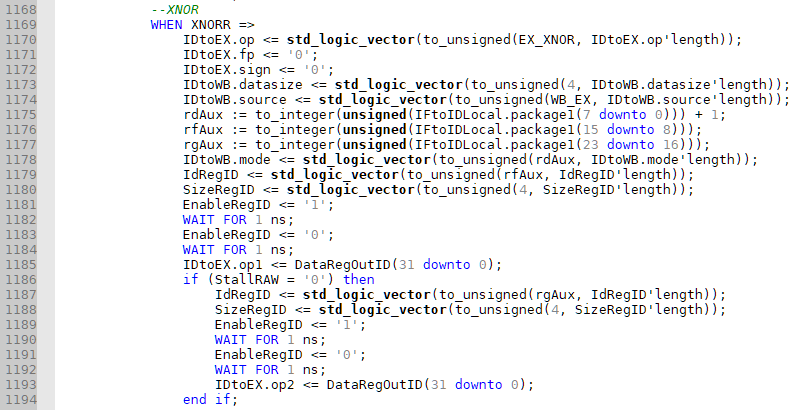
***Figura 5*:** Incorporación de instrucción xnorr en repert\_cpu.vhd

Para la implementación de esta instrucción, tomé como base la instrucción **xorr**, la cual utiliza registros similares. La modificación principal se realizó en la ALU, donde se añadió el código de operación correspondiente en const\_cpu.vhd para realizar la operación **XNORR**, asignándole el siguiente valor entero disponible, como se muestra en la Figura 6.



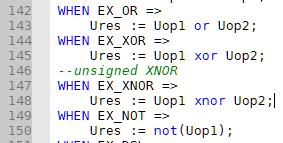
***Figura 6*:** Incorporación del código de operación dsubi en la ALU en const­\_cpu.vhd

También agregué la operación **XNORR** en decode.vhd, resultando en el código que se observa en la Figura 7.

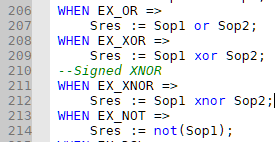


***Figura 7*:** Incorporación de instrucción xnorr en decode.vhd

En el archivo execute\_alu.vhd, se añadieron los códigos necesarios para realizar la operación con números con y sin signo, tal como se muestra en las Figuras 8 y 9.

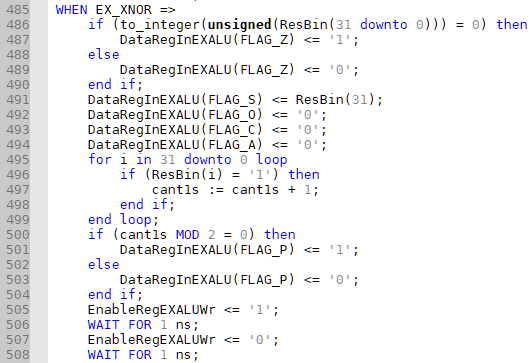


***Figura 8*:** Incorporación de xnorr en execute\_alu.vhd para unsigned

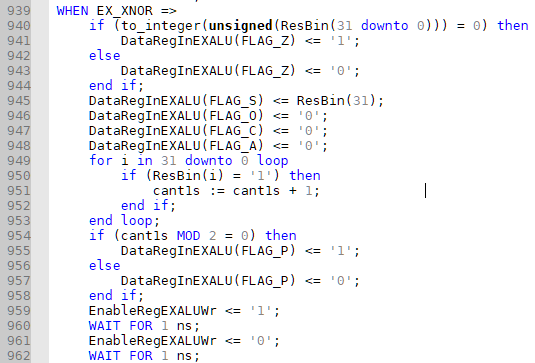


***Figura 9*:** Incorporación de xnorr en execute\_alu.vhd para signed

Finalmente, en el mismo archivo, incluí el cálculo de flags tanto para números con o sin signo, como se observa en las Figuras 10 y 11.



***Figura 10*:** Flags de xnorr en execute\_alu.vhd para unsigned



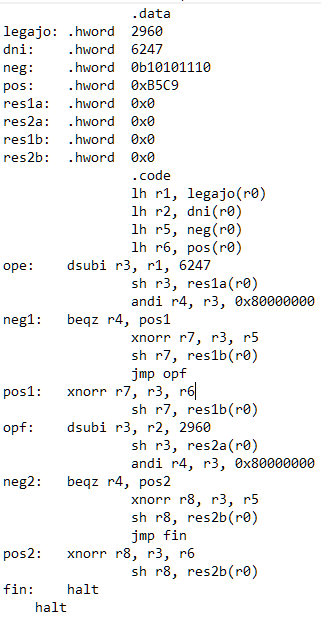
***Figura 11*:** Flags de xnorr en execute\_alu.vhd para signed

# 3. Desarrollo de programa de prueba

El programa de prueba en lenguaje ensamblador, test.asm (ubicado en TDA\_1819/src/Assembler), fue diseñado para validar las instrucciones dsubi y xnorr implementadas previamente. Este programa realiza una serie de operaciones aritméticas y lógicas sobre dos valores de 16 bits, obtenidos del número de legajo y los últimos cuatro dígitos del DNI, según los criterios especificados en la consigna. A continuación, se describe el desarrollo del programa paso a paso:

1. **Carga de valores:** El programa comienza cargando en registros específicos los valores de legajo, DNI, y dos constantes adicionales, neg (binario 10101110) y pos (hexadecimal B5C9). Estas constantes se emplearán en las operaciones lógicas dependiendo del signo de los resultados de las restas entre el legajo y el DNI.
2. **Primer cálculo de diferencia:** Utilizando la instrucción dsubi, el programa resta 6247 al valor del legajo y almacena el resultado en memoria. Luego, verifica el signo del resultado mediante una operación andi con la máscara 0x80000000 para determinar si el resultado es negativo o positivo.
3. **Operación lógica condicional:** Si el resultado de la resta es negativo, se ejecuta la instrucción xnorr entre el resultado y la constante neg (10101110). Si el resultado es positivo, la operación xnorr se realiza entre el resultado y la constante pos (B5C9). El resultado de esta operación lógica se guarda en la memoria principal.
4. **Segundo cálculo de diferencia:** El programa repite el cálculo de diferencia, esta vez restando 2960 al valor del DNI mediante dsubi. Al igual que en el paso anterior, verifica el signo del resultado y, en función de este, realiza una operación xnorr con las constantes neg o pos.
5. **Almacenamiento final y conclusión:** Todos los resultados de las operaciones aritméticas y lógicas se almacenan en la memoria principal para revisión, y el programa finaliza con la instrucción halt.

A continuación, se muestra el código test.asm en lenguaje ensamblador, que implementa las instrucciones dsubi y xnorr conforme a las especificaciones:



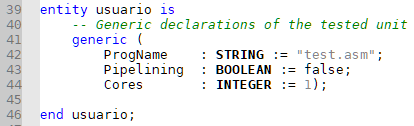
***Figura 12***: Código test.asm del programa de prueba en lenguaje ensamblador.

Esta implementación permite verificar la funcionalidad de las instrucciones desarrolladas, asegurando que el procesador TDA 1819 ejecute correctamente las operaciones aritméticas y lógicas requeridas. Además, al utilizar un programa en lenguaje ensamblador independiente, se asegura que el procesador maneje adecuadamente cada operación sin necesidad de ajustes adicionales en su arquitectura, con excepción del archivo usuario.vhd, donde se selecciona el código a ejecutar y la configuración de segmentación del pipeline.

# 4. Análisis de Resultados

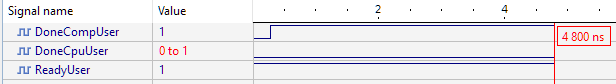
## ****4.1 Pipeline desactivado****

Se realizó una prueba con el pipeline desactivado, configurado en el archivo usuario.vhd, como se muestra en la figura 13.



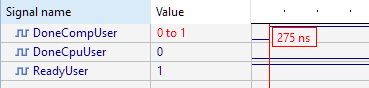
***Figura 13***: pipeline desactivado en usaurio.vhd

Con el pipeline desactivado todas las operaciones se ejecutan de manera secuencial, en la siguiente imagen se observan 3 señales que representan cuando el programa fue ensamblado cuando finaliza y cuando el sistema está listo para el usuario, por lo que con estas señales puedo simular el programa completo y veo que se ejecuta hasta los 4550ns como se observa en la figura 14



***Figura 14***: finalización de la ejecución del programa sin pipeline

Como se ve en la figura 15 el tiempo de ensamblaje es de 275 ns, por lo que el tiempo total será de 4800ns – 275ns = 4525ns



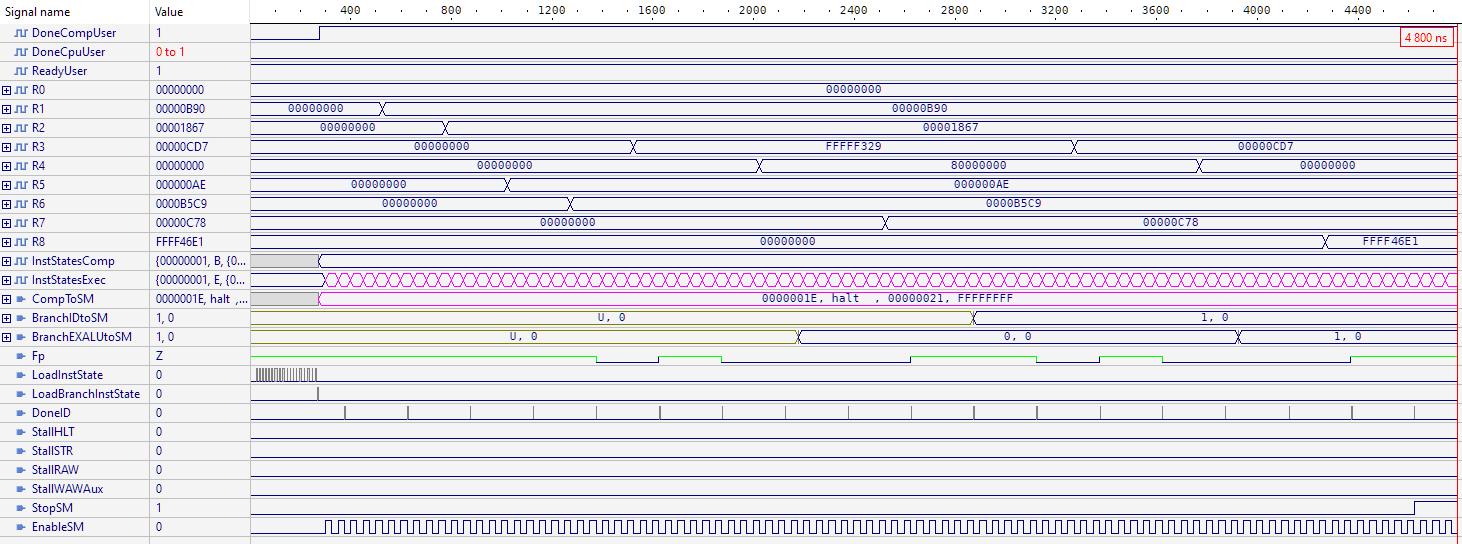
***Figura 15***: Tiempo de ensamblaje sin pipeline

Luego agregué al waveform las señales de los registros de la CPU y de la máquina de estados. Como se muestra en la figura 16 (hacer zoom para visualizarlo mejor), el registro R1 almacena los dígitos correspondientes al legajo en hexadecimal B90 (equivalente a 2960 en decimal), mientras que en R2 se cargan los dígitos correspondientes a mi documento en hexadecimal 1867 (equivalente a 6247 en decimal).

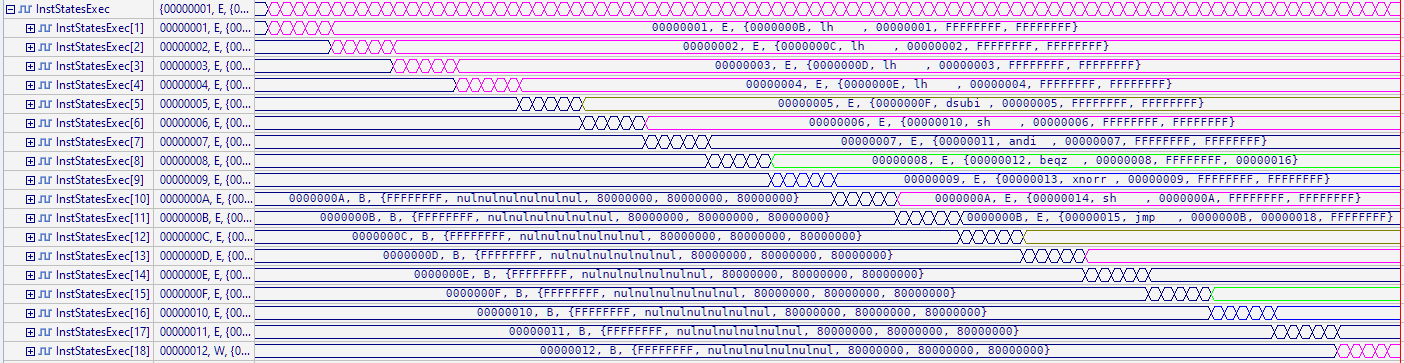
Posteriormente, se observa que al comienzo, R3 contiene el resultado de la operación legajo - DNI, que da FFFFF329 en hexadecimal. Esto ocurre porque el valor correcto en 16 bits es F329 (-3287 en decimal). Sin embargo, debido a que el procesador es de 32 bits, la extensión de signo completa los bits superiores con F al interpretarse como un número negativo, lo cual explica la salida en FFFFF329 en lugar de solo F329. Luego se ve que R3 toma el valor CD7 correspondiente a 3287 pero en hexadecimal, que es el resultado de hacer DNI – legajo.

Por otra parte en R7 al comienzo se hace (legajo – dni) xnor 10101110 que como se puede ver devuelve C78 (3192) lo cual es correcto, ya que F329 XNOR AE(10101110) da como resultado C78.

También se puede ver que R8 toma el valor FFFF46E1, correspondiente de hacer (dni – legajo) XNOR 0xB5C9, que en este caso pasa lo mismo que explique previamente.



***Figura 16***: Señales de STM y registros CPU

En cuanto a las instrucciones aritmético-lógicas que se implementaron y sus respectivos tiempos de ejecución es necesario revisar en detalle InstStatesExec el cual se puede ver al añadir la máquina de estados al waveform y se observa lo siguiente:  


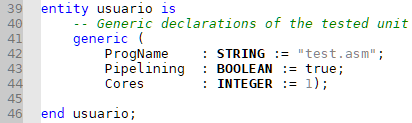
***Figura 17***: Señales de InstStatesExec sin pipeline

Aquí se puede ver las 4 instrucciones de interés (5, 9, 12 y 16) siendo sus tiempos y los momentos en los que pasan por la ALU los siguientes.

* **Instrucción 5**: Se ejecuta entre 1300 ns y 1550 ns, pasando por la ALU entre 1400 ns y 1450 ns.
* **Instrucción 9**: Se ejecuta entre 2300 ns y 2550 ns, pasando por la ALU entre 2400 ns y 2450 ns.
* **Instrucción 12**: Se ejecuta entre 3050 ns y 3300 ns, pasando por la ALU entre 3150 ns y 3200 ns.
* **Instrucción 16**: Se ejecuta entre 4050 ns y 4300 ns, pasando por la ALU entre 4150 ns y 4200 ns.

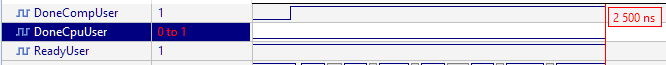
## ****4.2 Pipeline activado****

Con el pipeline activado lo que tengo que hacer es similar, salvo que pongo en usuario.vhd ‘Pipeling’ en true como se puede ver en la figura 18, por lo que todas las explicaciones previas sobre los resultados del programa y el tiempo de ejecución se mantienen y los resultados del programa no cambian, solo los tiempos de ejecución y es por eso que eso será lo único de lo que hablare en este apartado



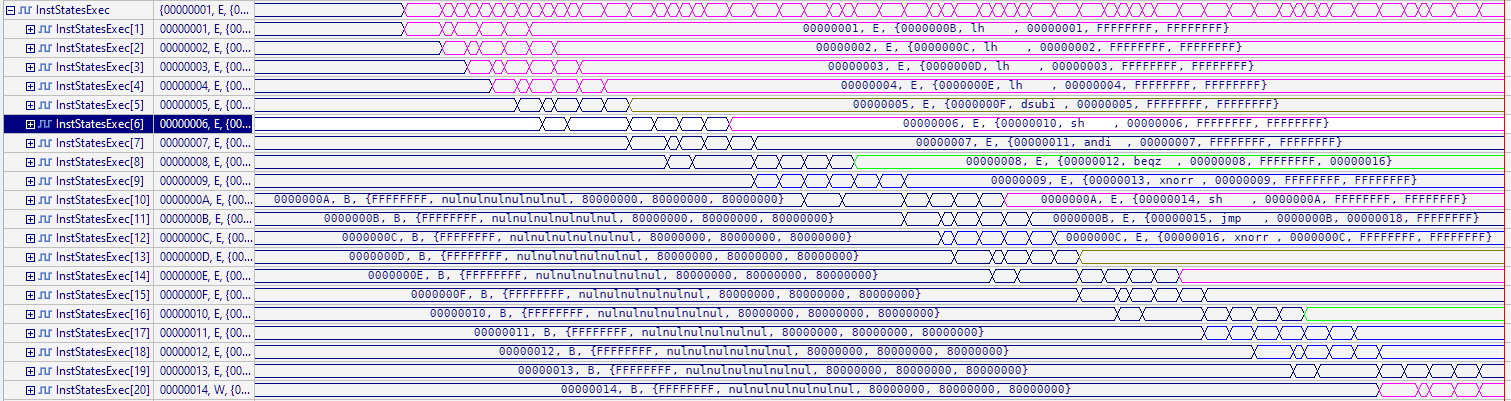
***Figura 18***: pipeline activado en usaurio.vhd

Como se puede ver en la figura 19, ahora el programa tarda 2500ns en completarse, si sacamos el tiempo de ensamblaje nos queda 2500ns – 275ns = 2225ns



***Figura 19***: finalización de la ejecución del programa con pipeline

Por ultimo las señales al agregar InstStatesExec cambian como se ve en la figura 20 ya que al tener el pipeline activado las instrucciones pueden ejecutarse en paralelo si se encuentran en distintas etapas del ciclo de ejecución y obtenemos los tiempos que menciono a continuación, los cuales son más eficientes. En este caso el xnorr la realiza la instrucción 17 que luego es cancelada por un fallo en el adelantamiento de instrucción tras el salto en el flujo de control del programa, por lo cual las instrucciones relevantes son la 5, 9, 13 y 18.



***Figura 20***: Señales de InstStatesExec con pipeline

A continuación, se detallan los tiempos de ejecución de cada instrucción relevante:

* **Instrucción 5:** Se ejecuta entre 527 ns y 750 ns, pasando por la ALU entre 600 ns y 650 ns.
* **Instrucción 9:** Se ejecuta entre 1000 ns y 1300 ns, pasando por la ALU entre 1100 ns y 1150 ns.
* **Instrucción 13:** Se ejecuta entre 1400 ns y 1650 ns, pasando por la ALU entre 1500 ns y 1550 ns.
* **Instrucción 18:** Se ejecuta entre 2000 ns y 2250 ns, pasando por la ALU entre 2100 ns y 2150 ns.

# 5. Conclusión

En este trabajo práctico se logró implementar y validar las instrucciones **dsubi** y **xnorr** en el procesador TDA 1819, lo que permitió profundizar en conceptos clave de diseño de procesadores y programación en ensamblador. A través del desarrollo y simulación de un programa de prueba, se verificó el correcto funcionamiento de las instrucciones, evaluando su impacto en un entorno segmentado y no segmentado.

El análisis de los resultados demostró que, con el pipeline desactivado, las instrucciones se ejecutan de manera secuencial, resultando en un mayor tiempo de ejecución. En contraste, al activar el pipeline, se optimizó el tiempo total gracias a la ejecución en paralelo de diferentes etapas del ciclo de instrucción, aunque surgieron desafíos como el manejo de saltos en el flujo de control.

La implementación y los tiempos observados permiten concluir que las mejoras introducidas mediante el uso del pipeline son significativas, logrando una ejecución más eficiente y confirmando la correcta funcionalidad de las instrucciones en el contexto del procesador TDA 1819.