# Introducción

Para esta práctica se utilizará un procesador diseñado por la catedra, llamado “TDA 1819” que podría decirse que es un procesador hibrido entre un “MSX86” y “MIPS64”. El procesador en cuestión tiene una arquitectura de tipo RISC, en donde cada instrucción se ejecutará exactamente en un único ciclo de reloj. Cuenta con un bus de datos de datos de 32 bits y con 2 unidades independientes para operaciones aritmético-lógicas, una ALU (Arithmetic Logic Unit) para operandos enteros y una FPU (Floating-Point Unit) para operandos en punto flotante. Tenemos 16 registros de uso general para números enteros y 16 registros de uso general para números de punto flotante. Por ultima, en cuanto a la segmentación contamos con cinco etapas, en primera instancia una etapa de búsqueda, luego una etapa de decodificación, en tercer lugar, una etapa de ejecución, luego tenemos una etapa de acceso a memoria y para finalizar una etapa de almacenamiento en registro. Con el procesador descripto anteriormente haremos algunas pruebas y añadiremos nuevas funcionalidades a su reportorio de instrucciones, adicionándole el uso de “Waveforms” para la medición de tiempos, con segmentación de cauce habilitada o deshabilitada.

# Nuevo repertorio de instrucciones

Para las pruebas que se llevaran a cabo más adelante, se hará de uso de 2 instrucciones que no están en el repertorio original del procesador. Para ello deberemos modificar algunos archivos del “Workspace” que se nos fue brindado para acceder a todo el procesador “TDA 1819”.

Tenemos una carpeta “Package” donde tenemos algunos paquetes, dentro de ella podemos encontrar un archivo llamado “repert\_cpu.vhd”, allí tenemos el repertorio de todas las instrucciones junto con el código de operación asignado a cada una de las mismas a fin de que el ensamblador pueda codificarla de manera tal que resulte inteligible para la unidad de control de la CPU, la cual se encargará de interpretarlo y, a partir de él, determinar las acciones a realizar a continuación para que el procesador pueda ejecutar dicha instrucción.

Para nuestro caso, se añadirán las siguientes instrucciones:Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Tenemos una instrucción de tipo aritmética y otra de tipo lógica, por lo tanto, debemos buscar en el archivo anteriormente enunciado, y en la documentación, aquellos códigos de operación que corresponden a operaciones aritméticas y a operaciones lógicas. Podemos ver que las operaciones aritméticas están entre “00011000” y “00101011”. En cambio, las operaciones lógicas están entre “00110000” y “00110110”. Podemos ver que, tanto para aritméticas como para lógicas, tenemos lugar disponible para incluir las nuevas operaciones como veremos a continuación.

Texto, Tabla

Descripción generada automáticamenteTexto

Descripción generada automáticamente con confianza media

Podemos observar que la instrucción “DSUBI” fue añadida con el código de operación “00101100” que es el numero siguiente al de la operación “NEGR”. Podemos poner el siguiente ya que el primer código del siguiente tipo de operación (en este caso las operaciones lógicas), comienza recién en el valor “00110000”. Lo mismo nos pasa para el caso de la operación lógica “XNORR” que cuenta con el código de operación “00110111”.

Ahora que ya tenemos las instrucciones, debemos añadirle la lógica correspondiente, esto lo hacemos en el archivo “decode.vhd”, que corresponde a la etapa de decodificación de la segmentación. Para esta instrucción no requerimos modificar la ALU debido a que ya hay una implementación de resta. Para el caso de la instrucción XNOR, deberemos modificar tanto la ALU para que en la etapa de “execute” se ejecute “EX\_XNOR” que no se encuentra implementado.

Texto, Carta

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Como se ve en la línea 1202 y como se había dicho antes, debemos implementar en la ALU “EX\_XNOR” que representa hacer una xnor entre dos operandos.

Texto

Descripción generada automáticamente

Esto lo debemos hacer tanto para números enteros como para números flotantes.

Una vez que ya actualizamos todo podemos proceder a las pruebas.

# Programa de pruebas

Para la prueba pertinente de las instrucciones implementadas y descriptas anteriormente, se utilizarán 2 datos:

1. Numero de legajo: se utilizarán los dígitos que quepan dentro del máximo valor correspondiente al mismo que resulte representable en el sistema donde interactuamos, como tenemos binarios con signo, el número máximo representable se corresponde con el número , por lo tanto, como nuestro código de legajo de interés es 02497/5, utilizaremos los dígitos 24975, que esta dentro de los valores de representación.
2. DNI: Se procede como con el numero de legajo, pero partimos desde los números menos significativos, por lo tanto, como nuestro DNI de interés es 44051761, escogemos el valor 1761.

Ahora que ya definimos los operandos, debemos incluirlos dentro del código assembler que probara las instrucciones previamente pedidas. Para ello utilizamos el código que se encuentra en “TDA\_1819/src/Assembler/test.asm”. Texto

Descripción generada automáticamente

Comenzamos definiendo el .data donde guardaremos en memoria tanto, los operandos, como los resultados de las operaciones. Podemos ver legajo y dni con los valores explicados anteriormente. Luego tenemos 2 valores “neg” con el valor “10101110”, este valor lo utilizaremos para el caso en que la resta entre el legajo y el dni, o viceversa, de negativo, y lo que haremos será aplicarle una XNOR con el valor “neg”. El otro valor es “pos” con el valor “B5C9”, que al igual que neg lo utilizaremos como operando del XNOR, pero cuando la resta sea positiva. Por último, tenemos los 4 resultados, las dos restas utilizando la instrucción implementada “DSUBI” y los 2 resultados de aplicar la compuerta “XNOR” también implementada.

Texto

Descripción generada automáticamente con confianza media

Comenzamos copiando en los registros los operandos y resultados de interés. Luego tendremos 2 etiquetas, “op1” y “op2”, en donde la primera se refiere a la primera operación, y la siguiente a la ultima operación. Dicho de otra forma, “op1” es “Legajo – DNI” y “op2” es “DNI – Legajo”. Vemos que en ambos hacemos lo mismo, comenzamos utilizando la instrucción “DSUBI” para realizar una resta inmediata entre “Legajo – DNI” o entre “DNI – Legajo”, luego con una operación “sh” guardamos el valor en memoria. Por último, con la operación “slti” comparamos la resta hecha con el valor 0, para saber si el resultado es menor a 0, en caso de que sea menor a 0, guardara en el registro 4, un 1, que nosotros podremos aprovechar con la operación “beqz”, ya que, si r4 es igual a 0, salta a la instrucción etiquetada “pos1” o “pos2” según corresponda. Podemos notar que tanto “posX” o “negX” realizan lo mismo, solo cambia el operando de la compuerta XNOR dependiendo de si la resta es positiva o negativa.

Una vez explicado el programa, algunas cosas que hay que tener en cuenta:

* Almacenamos números de 16 bits.
* Al igual que en MSX, tenemos el formato “Little Endian” para almacenar los datos de un byte en memoria.
* El programa contiene 2 halt, ya que, por alguna razón, si ponemos solo 1 nos dice que falta halt. A su vez, probamos con poner un “nop” en la instrucción etiquetada “fin” pero tampoco nos resultó. Creemos que es un error del compilador de Active-HDL debido a que no pasaba en simuladores de procesadores anteriores.

Para poder compilar y ejecutar nuestro programa de assembler, en el archivo que se encuentra en “TDA\_1819/src/Usuario/usuario.vhd” deberemos aclarar como se llama el archivo con extensión .asm que utilizaremos, y lo deberemos agregar a la carpeta que se encuentra en “TDA\_1819/src/Assembler”. Luego tenemos una opción para activar o desactivar la segmentación de Cauce. Podemos verlo mas claro en el siguiente código:

Texto

Descripción generada automáticamente

Una vez simulado el código, podemos acceder a la memoria y ver si se guardan los valores correctamente, esto lo hacemos desde aquí:Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Cuando hacemos un click en UUT4, nos aparecerán señales tales como:

Tabla

Descripción generada automáticamente con confianza media

Allí, encontramos Data\_Memory que contiene la memoria de nuestra PC, podemos observar que los primeros 2 valores, son 618F (Recordando que es Little Endian) que se corresponde en decimal con el numero 24.975, que es el numero de legajo declarado en .data. Podemos entonces concluir que funciona correctamente.Tabla

Descripción generada automáticamente

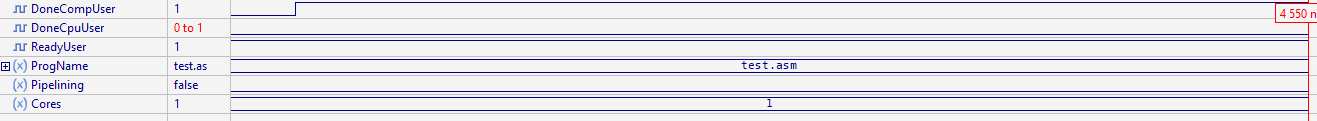
Ahora podemos comparar los tiempos del programa tanto, con pipeline activado o desactivado, y tenemos que:

* Con pipeline **ACTIVO**:



Tenemos que con cauce activo el programa tarda 2450 ns. Por lo tanto con cauce desactivado se esperaría un tiempo mayor.

* Con pipeline **DESACTIVADO:**



Como era de esperarse, con el cauce desactivado tenemos un tiempo de ejecución de 4550 ns, siendo este mayor a cuando tenemos el cauce activo.

# Guía para los archivos

Aquí se podrá encontrar todos aquellos archivos modificados que serán adjuntados junto a este informe, para que pueda ser analizado.

* El archivo “const\_ensamblador.vhd” va en la dirección: “TDA\_1819/src/Packages/”
* El archivo “repert\_cpu.vhd” va en la dirección: “TDA\_1819/src/Packages/”
* El archivo “decode.vhd” va en la dirección: “TDA\_1819/src/Usuario/PC/2. Etapas/2. Decode/”
* El archivo “execute\_alu.vhd” va en la dirección: “TDA\_1819\src\Usuario\PC\2. CPU\2. Etapas\3. Execute\1. ALU/”
* El archivo “const\_cpu.vhd” va en la dirección: “TDA\_1819\src\Packages”
* El archivo “usuario.vhd” va en la dirección: “TDA\_1819\src\Usuario”

# Anexo

Existe un error en el programa que no pude solucionar, y es que, por alguna razón, el código assembler no salta cuando debería saltar, en la última operación, correspondiente al xnor, el resultado que da es EF64 y es el resultado de hacer (DNI - LEGAJO) XNOR B5C9 = EF64, pero no es correcto ya que debería usar el otro código provisto por la catedra.