Informe practica 10 Taller de Arquitectura

PROCESADOR TDA 1819

Joaquín Chanquía
NUMERO DE LEGAJO | 02887/7

1. Implemente las siguientes instrucciones propuestas para el procesador “TDA 1819":

Tipo de instrucción	Instrucción	Comentario
Aritmética	dsubi r_d, r_f, N	Resta el valor inmediato N al registro r_f, dejando el resultado en el registro r_d (valores con signo)
Lógica	xnorr r_d, r_f, r_g	Realiza un XNOR entre los registros r_f y r_g (bit a bit), dejando el resultado en el registro r_d

Dentro del código deje comentarios al lado de todas las líneas que agregue o modifique marcándolos como “--JCH “. Si se busca esto en los archivos de los que hablo a continuación pueden verse las líneas.

Los archivos vhd entregados están en un rar con dos carpetas adentro: Archivos_028877 y My_Designs. En Archivos se encuentran todos los archivos sueltos para poder acceder uno a uno o ubicarlos manualmente en las direcciones dadas. El archivo My_Designs esta diseñado para poder copiarlo sobre la carpeta My_Designs donde se guardan los workspaces de Aldec y que se modifiquen los archivos automáticamente dado que siguen la misma topología de carpetas.

Para poder implementar estas operaciones primero definí la operación junto a su código único en el archivo repert_cpu.vhd, ubicado en la carpeta Packages. Para la instrucción dsubi la ubique debajo de la operación dsub en la sección de operaciones aritméticas y le asigne el siguiente número de operación 2C ya que era el siguiente disponible a la última operación aritmética. Para la xnorr la ubique debajo de la operación xorr en la sección de operaciones lógicas. Y le asigne el número de operación 37 (hexadecimal) por la misma razón que al dsubi.

```
-- DSUB rd, rf, rg
CONSTANT DSUB:          std_logic_vector(7 downto 0) := "00011101";
-- DSUBI rd, rf, N
CONSTANT DSUBI:         std_logic_vector(7 downto 0) := "00101100";  --JCH Agregue esta linea para definir el codigo de operacion de DSUBI (2C)
-- DSUBU rd, rf, rg
CONSTANT DSUBU:         std_logic_vector(7 downto 0) := "00011110";
-- XOR rd, rf, rg
CONSTANT XORR:          std_logic_vector(7 downto 0) := "00110100";
-- XNORR rd, rf, rg
CONSTANT XNORR:         std_logic_vector(7 downto 0) := "00110111"; --JCH Agregue esta linea para definir el codigo de operacion de XNORR (37)
-- XORI rd, rf, N
CONSTANT XORI:          std_logic_vector(7 downto 0) := "00110101";
```

Lo siguiente que hice fue declarar el nombre y el tamaño asociado a cada código en el archivo const_ensamblador.vhd, los agregue en las mismas posiciones que en el repertorio y le asigne un tamaño de 8 bytes al dsubi por ser una operación con un inmediato y de 5 al xnorr por ser una operación lógica entre registros.

En este mismo archivo también modifique las variables CANT_INSTAR y CANT_INSTLD a las cuales le sume 1 al valor que tenían de antemano.

```
                        --JCH Agregue el nombre, codigo y tamanio de la operacion dsubi en las constantes aritmeticas
CONSTANT INSTAR_NAMES:  instar_name_array(1 to CANT_INSTAR) := ("dadd  ", "daddi ", "daddu ", "dadrui", "addf  ", "dsub  ", "dsubi ",
CONSTANT INSTAR_CODES:  instar_code_array(1 to CANT_INSTAR) := (DADD, DADDI, DADDU, DADDUI, ADDF, DSUB, DSUBI, DSUBU, SUBF, DMUL, DMU
CONSTANT INSTAR_SIZES:  instar_size_array(1 to CANT_INSTAR) := (5, 8, 5, 8, 5, 5, 8, 5, 5, 5, 5, 5, 5, 5, 5, 8, 4, 4, 4, 4);
                        --JCH Agregue el nombre, codigo y tamanio de la operacion xnorr en las constantes logicas
CONSTANT INSTLD_NAMES:  instld_name_array(1 to CANT_INSTLD) := ("and  ", "andi ", "or   ", "ori  ", "xor  ", "xnorr ", "xori ", "not
CONSTANT INSTLD_CODES:  instld_code_array(1 to CANT_INSTLD) := (ANDR, ANDI, ORR, ORI, XORR, XNORR, XORI, NOTR, DSL, DSLI, DSR, DSRI,
CONSTANT INSTLD_SIZES:  instld_size_array(1 to CANT_INSTLD) := (5, 8, 5, 8, 5, 5, 8, 4, 5, 8, 5, 8, 5, 8, 5, 8);
```

Luego debo declarar los pasos a realizar al momento de decodificarse la operación en el archivo decode.vhd en la ubicación Usuario\PC\2. CPU\2. Etapas\2. Decode. Para ello utilice como modelo la operación DADDI para el DSUBI, ya que ambos son una operación aritmética que utiliza un valor inmediato. Lo que le hice fue cambiar la operación de alu que se ejecuta de un EX_ADD a un EX_SUB. Para que realice una resta en lugar de una suma.

```
WHEN DSUBI =>  --JCH Aca se define el comportamiento de la operacion especificamente DSUB con Inmediato
    IDtoEX.op <= std_logic_vector(to_unsigned(EX_SUB, IDtoEX.op'length));
    IDtoEX.fp <= '0';
    IDtoEX.sign <= '1';
    IDtoWB.datasize <= std_logic_vector(to_unsigned(4, IDtoWB.datasize'length));
    IDtoWB.source <= std_logic_vector(to_unsigned(WB_EX, IDtoWB.source'length));
    rdAux := to_integer(unsigned(IFtoIDLocal.package1(7 downto 0))) + 1;
    rfAux := to_integer(unsigned(IFtoIDLocal.package1(15 downto 8)));
    IDtoEX.op2(7 downto 0) <= IFtoIDLocal.package1(23 downto 16);
    IDtoEX.op2(31 downto 8) <= IFtoIDLocal.package2(23 downto 0);
    IDtoWB.mode <= std_logic_vector(to_unsigned(rdAux, IDtoWB.mode'length));
    IdRegID <= std_logic_vector(to_unsigned(rfAux, IdRegID'length));
    SizeRegID <= std_logic_vector(to_unsigned(4, SizeRegID'length));
    EnableRegID <= '1';
    WAIT FOR 1 ns;
    EnableRegID <= '0';
    WAIT FOR 1 ns;
    IDtoEX.op1 <= DataRegOutID(31 downto 0);
```

Para la operación xnorr no fue tan sencillo porque la operación xnor no estaba declarada en el repertorio de ejecución de la alu. Primero al igual que en el dsubi utilice otra operación como modelo para el archivo decode.vhd, en este caso XORR ya que era una operación lógica entre registros. Y el cambio fue pasar del EX_XOR a un EX_XNOR que por el momento no existía en la cpu.

```
WHEN XNORR =>       --JCH Aca se define el comportamiento de la operacion especificamente XNOR con Registros
    IDtoEX.op <= std_logic_vector(to_unsigned(EX_XNOR, IDtoEX.op'length));
    IDtoEX.fp <= '0';
    IDtoEX.sign <= '0';
    IDtoWB.datasize <= std_logic_vector(to_unsigned(4, IDtoWB.datasize'length));
    IDtoWB.source <= std_logic_vector(to_unsigned(WB_EX, IDtoWB.source'length));
    rdAux := to_integer(unsigned(IFtoIDLocal.package1(7 downto 0))) + 1;
    rfAux := to_integer(unsigned(IFtoIDLocal.package1(15 downto 8)));
    rgAux := to_integer(unsigned(IFtoIDLocal.package1(23 downto 16)));
    IDtoWB.mode <= std_logic_vector(to_unsigned(rdAux, IDtoWB.mode'length));
    IdRegID <= std_logic_vector(to_unsigned(rfAux, IdRegID'length));
    SizeRegID <= std_logic_vector(to_unsigned(4, SizeRegID'length));
    EnableRegID <= '1';
    WAIT FOR 1 ns;
    EnableRegID <= '0';
    WAIT FOR 1 ns;
    IDtoEX.op1 <= DataRegOutID(31 downto 0);
    if (StallRAW = '0') then
        IdRegID <= std_logic_vector(to_unsigned(rgAux, IdRegID'length));
        SizeRegID <= std_logic_vector(to_unsigned(4, SizeRegID'length));
        EnableRegID <= '1';
        WAIT FOR 1 ns;
        EnableRegID <= '0';
        WAIT FOR 1 ns;
        IDtoEX.op2 <= DataRegOutID(31 downto 0);
    end if;
```

Para agregar esta operación primero fui al archivo const_cpu.vhd ubicado en la carpeta Packages y agregué la constante EX_XNOR debajo de EX_XOR. Luego de esto, para que funcione, tuve que asignarle el valor en orden después de esta (15) y correr todos los valores de las operaciones siguientes.

```
CONSTANT EX_XOR:        INTEGER := 14;
CONSTANT EX_XNOR:       INTEGER := 15;  --JCH Agregue esta linea para definir la constante de CPU EX_XNOR y corri todos los indices
CONSTANT EX_NOT:        INTEGER := 16;
CONSTANT EX_DSL:        INTEGER := 17;
CONSTANT EX_DSR:        INTEGER := 18;
CONSTANT EX_BEQ:        INTEGER := 19;
CONSTANT EX_BNE:        INTEGER := 20;
CONSTANT EX_BEQZ:       INTEGER := 21;
CONSTANT EX_BNEZ:       INTEGER := 22;
CONSTANT EX_BFPT:       INTEGER := 23;
CONSTANT EX_BFPF:       INTEGER := 24;
```

Por último, tuve que declarar el comportamiento de la operación EX_XNOR en el archivo execute_alu.vhd ubicado en Usuario\PC\2. CPU\2. Etapas\3. Execute\1. ALU. El comportamiento que declare fue Ures := Uop1 xnor Uop2; cuando la operación es EX_XNOR.

```
WHEN EX_XOR =>
    Ures := Uop1 xor Uop2;
WHEN EX_XNOR =>                 --JCH Agregue estas lineas
    Ures := Uop1 xnor Uop2;     --JCH Para definir la operacion EX_XNOR
WHEN EX_NOT =>
    Ures := not(Uop1);
```

2. Desarrolle un programa de prueba en el lenguaje Assembler tal que haga uso de las instrucciones previamente implementadas y pueda ser ejecutado en el TDA 1819.
Para ello, en primer lugar, deberá tener en cuenta estas consideraciones:

a. Se utilizarán valores con signo de 16 bits cada uno.

b. A partir de su número de legajo, establezca el máximo valor correspondiente al mismo que resulte representable en este sistema binario. Incluya el dígito verificador en caso de ser posible. Por ejemplo, si su legajo fuera 00615/3, entonces el valor máximo sería 6153. En cambio, si fuera 09365/7, sería 9365.

c. Aplique sobre su número de documento un mecanismo similar al descrito en el inciso anterior. Aquí el objetivo consiste, más precisamente, en seleccionar y preservar los cuatro o cinco dígitos menos significativos del susodicho documento según corresponda a fin de obtener el máximo valor expresable en este sistema binario. Por ejemplo, si su documento fuera 38283239, entonces el valor máximo sería 3239. Por su parte, si fuera 39431491, sería 31491.

d. En caso de que los valores obtenidos en los apartados b. y c. resulten iguales entre sí, réstele una unidad al número calculado en el punto b.

A continuación, se presentan las tareas que habrán de realizarse por el programa solicitado en Assembler:

e. Calcular la diferencia entre los dos valores determinados en los incisos anteriores. Luego, efectuar nuevamente la operación aritmética intercambiando las posiciones de dichos operandos.

f. Para cada cálculo llevado a cabo en la tarea previa, si el resultado hubiera sido negativo, realizar la operación lógica XNOR entre este último y el valor binario 10101110. En caso contrario, llevar a cabo esta misma operación utilizando como segundo operando el número hexadecimal B5C9.

g. Almacenar en la memoria principal de la computadora el resultado de cada operación aritmético/lógica efectuada en los puntos e. y f.

a) Al utilizarse valores con signo de 16 bits con signo el valor máximo posible para representar en el sistema es 32768.
b) Mi número de legajo es 02887/7. Por lo que el máximo número representable en este sistema es 28877.
c) Mi número de documento es 44356213. Utilizando los 4 dígitos menos significativos llego al máximo número representable en este sistema, 6213. Si tomara el 5to digito el numero resultaría muy grande, 56213.

COMENTARIO: El código ahora explicado es un código simplificado en el que, para la primera operación, sabiendo que el resultado es positivo se realiza el xnorr con el valor B5C9 en hexadecimal y para la segunda operación, sabiendo que el resultado es negativo se realiza la operación implementada con el valor 10101110 en binario (AE en hexadecimal). Mas adelante muestro una versión corregida del código que funciona comparando los resultados de las operaciones de resta.

Código en assembler:
Nombre del archivo: testbenchJCH1.asm

		.data		Posición en memoria:
A:		**.hword**	**28877**	1000
B:		**.hword**	**6213**	1002
C:		**.hword**	**0xB5C9**	1004
D:		**.hword**	**0xAE**	1006
E:		**.hword**	**0**	1008
F:		**.hword**	**0**	100A
G:		**.hword**	**0**	100C
H:		**.hword**	**0**	100E
Pos:	Tam:	**.code**		CodOp: Acción:
2000	6	**lh r1, A(r0)**		06 r1 ← A(1000) = 28877 (0x70CD)
2006	6	**lh r6, B(r0)**		06 r6 ← C(1004) = 6213 (0x1845)
200C	6	**lh r3, C(r0)**		06 r3 ← B(1002) = 0xB5C9
2012	6	**lh r8, D(r0)**		06 r8 ← D(1006) = 0xAE
2018	8	**dsubi r2, r1, 6213**		2C r2 = 28877 – 6213 = 22664(0x5888)
2020	8	**dsubi r7, r6, 28877**		2C r7 = 6213 – 28877= -22664(0xA778)
2028	5	**xnorr r4, r2, r3**		37 r4 = 0x5888 xnor 0xB5C9 = 0x12BE
202D	5	**xnorr r9, r7, r8**		37 r9 = 0xA778 xnor 0xAE = 0x5829
2032	6	**sh r2, E(r0)**		07 r2 → E(1008) = 0x5888
2038	6	**sh r7, G(r0)**		07 r7 → G(100C) = 0xA778
203E	6	**sh r4, F(r0)**		07 r4 → F(100A) = 0x12BE
2044	6	**sh r9, H(r0)**		07 r9 → H(100E) = 0x5829
204A	2	**halt**		81 Fin
204C				

En esta página intenté sintetizar la mayor cantidad de información sobre el código en assembler que escribí, primero en la sección .data puse la posición de memoria en la que se guarda cada una de las variables que definí, al ser half-words para que sean de 16 bits ocupan 2 bytes cada una.

Luego para cada instrucción puse la posición en la memoria de instrucciones que ocupa, el tamaño de la instrucción, el código de operación correspondiente a la misma y al final que operación se realiza y el registro afectado por la misma.

El tamaño de la instrucción lo conseguí mirando el archivo const_ensamblador.vhd y el código del archivo repert_cpu.vhd, ambos de los cuales hablé para el punto 1. También verifique los mismos mirando el programa en ejecución con la señal IP, la cual indica que posición de la memoria de instrucciones se está accediendo para ser buscada.

Este código debe ser ubicado en la carpeta Assembler.

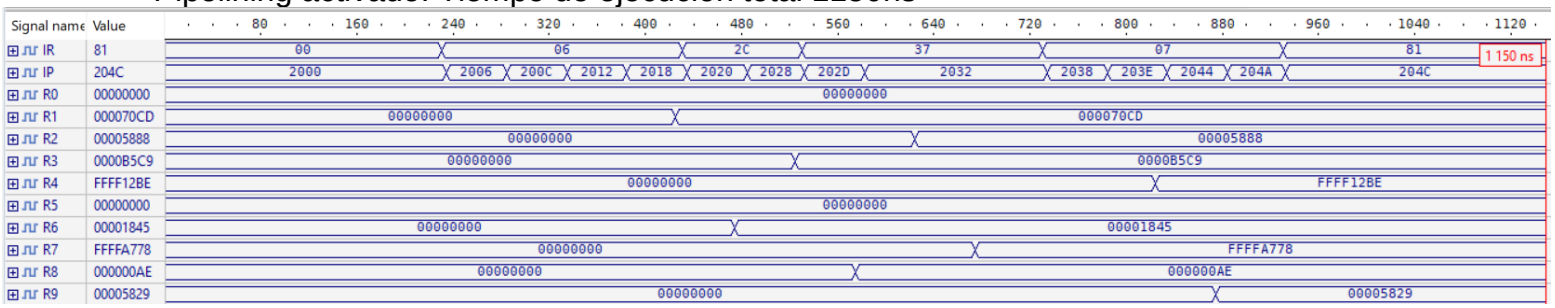
3. Una vez simulada la ejecución en el TDA 1819 del programa desarrollado en el ejercicio precedente, resuelva las consignas finales seguidamente definidas:
   a. Determine el tiempo de ejecución del programa con la segmentación del cauce del procesador habilitada y deshabilitada.
   b. Identifique las instrucciones del programa y, en particular, la etapa de sus respectivos ciclos de ejecución en las cuales se efectúan en la ALU del TDA 1819 cada una de las operaciones aritmético/lógicas especialmente solicitadas para dicho programa. Indique la duración y los instantes de inicio y fin de la etapa involucrada en cada caso (con la segmentación del cauce del procesador habilitada y deshabilitada).

Para ejecutar el código assembler escrito se utiliza el archivo “usuario.vhd”, ubicado en la carpeta Usuario. Para esto en la línea que dice progName se puede ingresar el nombre del archivo de la carpeta Assembler que será ejecutado. En mi caso este archivo se llama “testbenchJCH1.asm”. En este mismo archivo, debajo de la línea recién mencionada se encuentra la opción para activar o desactivar el pipelining (segmentación de cauce).

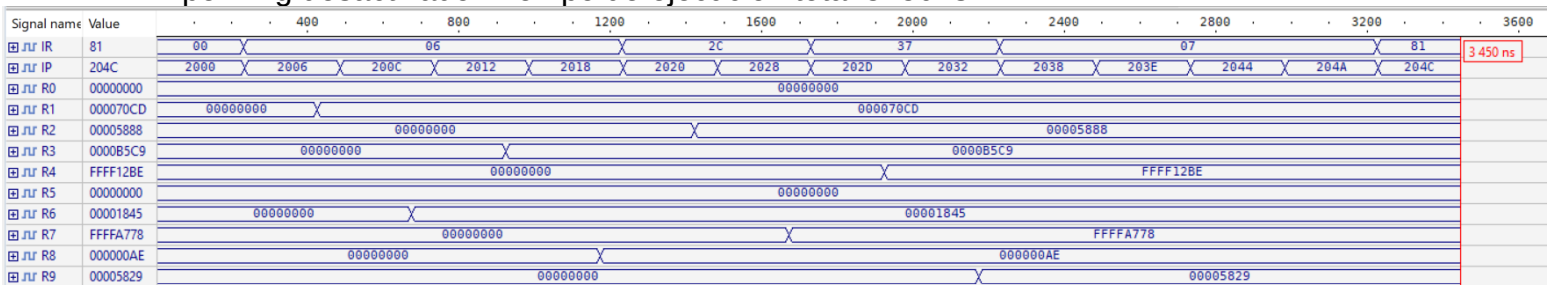
Ahora voy a mostrar los registros de la CPU y junto al tiempo de ejecución total del programa con la opción de pipelining activada o desactivada.

Para ver los registros agregue al waveform las señales ubicadas en:
/usuario/UUT/UUT2/UUT7

Pipelining activado: Tiempo de ejecución total 1150ns

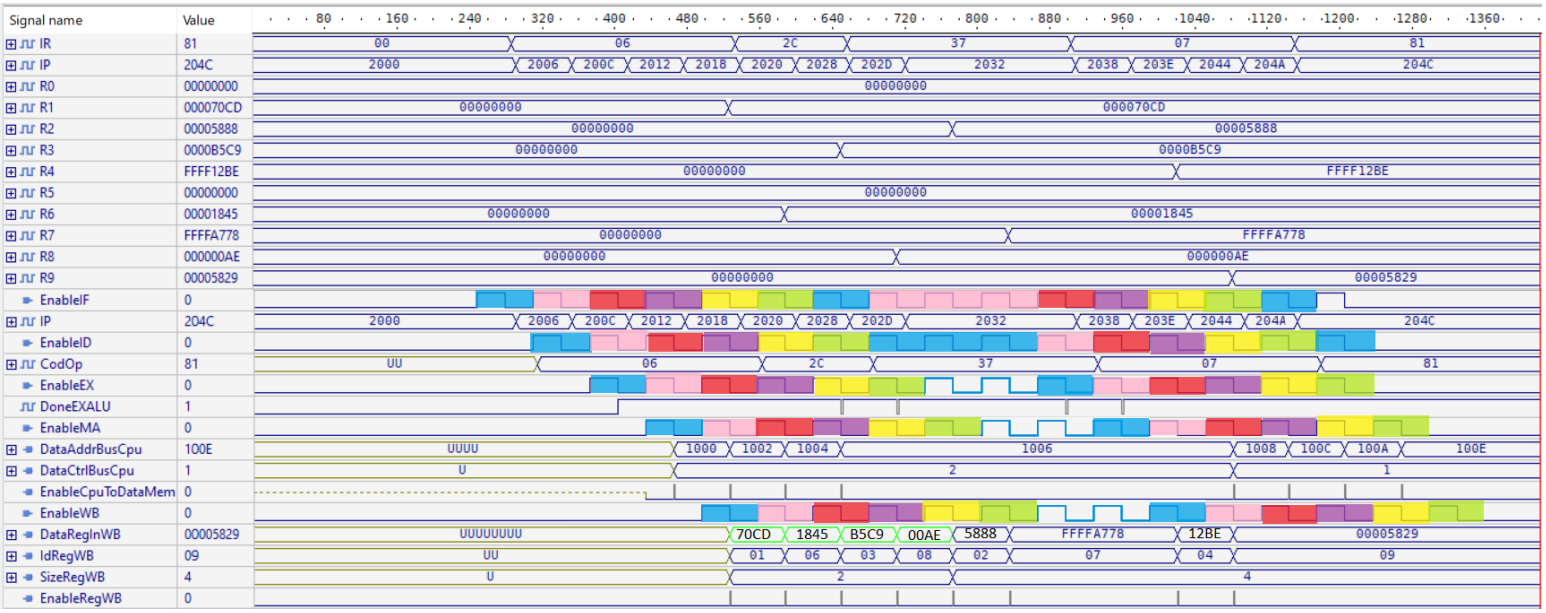


Pipelining desactivado: Tiempo de ejecución total 3450ns



Para mostrar las etapas por las que va pasando cada sentencia por el simulador primero les voy a asignar un color a cada una (en forma cíclica con 6 colores)

lh r1, A(r0)
lh r6, B(r0)
lh r3, C(r0)
lh r8, D(r0)
dsubi r2, r1, 6213
dsubi r7, r6, 28877
xnorr r4, r2, r3
xnorr r9, r7, r8
sh r2, E(r0)
sh r7, G(r0)
sh r4, F(r0)
sh r9, H(r0)
halt



Las señales que resalte son las que dicen si la etapa de cpu que indican esta en funcionamiento o no, se llaman EnableXX (siendo XX la abreviatura de la etapa), debajo de cada una de estas deje señales que muestran datos sobre la señal que esta pasando por esa etapa:

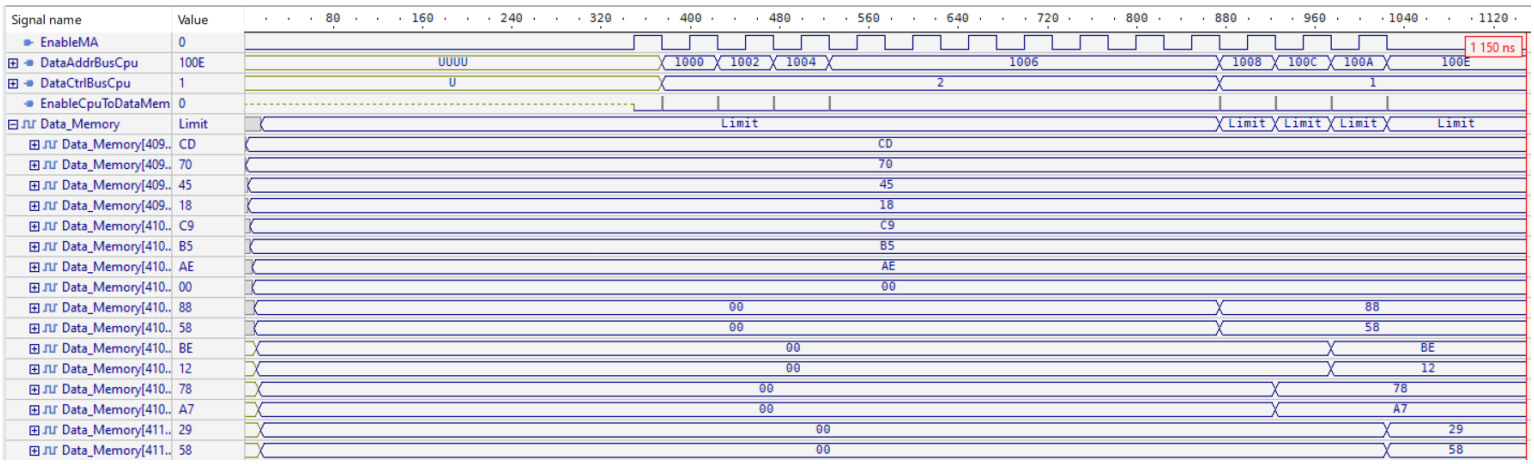
Etapa FETCH:
/usuario/UUT/UUT2/UUT2/UUT1/EnableIF
/usuario/UUT/UUT2/UUT7/IP
Esta señal indica cual es la siguiente línea de la memoria de instrucciones que debe ser leída por el fetch.
Una parte notable en esta señal es alrededor del tiempo 600 donde se nota que la operación de fetch no puede realizarse para la próxima línea porque la ejecución se encuentra interrumpida.

Etapa DECODE:
/usuario/UUT/UUT2/UUT2/UUT2/EnableID
/usuario/UUT/UUT2/UUT2/UUT2/CodOp
Esta señal indica que operación ha sido decodificada en esta etapa.
Primero se ven las operaciones de carga con un código 6, luego las dsubi con 2C como código de operación, seguidas del código 37 que corresponde al xnorr, luego los save a memoria con el código 7 y por último la operación halt con un valor de 81.

Etapa EXECUTE:
/usuario/UUT/UUT2/UUT2/UUT3/EnableEX
/usuario/UUT/UUT2/UUT2/UUT3/DoneEXALU
Esta señal indica cuando se ha terminado de ejecutar una operación en la alu
Esta aparece cuando por esta pasan las operaciones de resta y los xnor.

Etapa MEMORY ACCESS:
/usuario/UUT/UUT2/UUT2/UUT4/EnableMA
/usuario/UUT/UUT2/UUT2/UUT4/DataAddrBusCpu
Esta señal indica con que celda de memoria se va a trabajar en el acceso
Dada la estructura de mi código al cargarse en orden los registros que declare de la A a la H se ve que las celdas de memoria van de la 1000 hasta la 100E, dejando un espacio en el medio en el cual no cambian las celdas porque no se requieren accesos a memoria.
/usuario/UUT/UUT2/UUT2/UUT4/DataCtrlBusCpu
En esta señal se indica si el acceso es una lectura (con un 2) o una escritura (con un 1)
/usuario/UUT/UUT2/UUT2/UUT4/EnableCpuToDataMem
Esta señal indica cuando se debe realizar un acceso a memoria
Esta señal se activa cuando pasan las sentencias lh y sh.

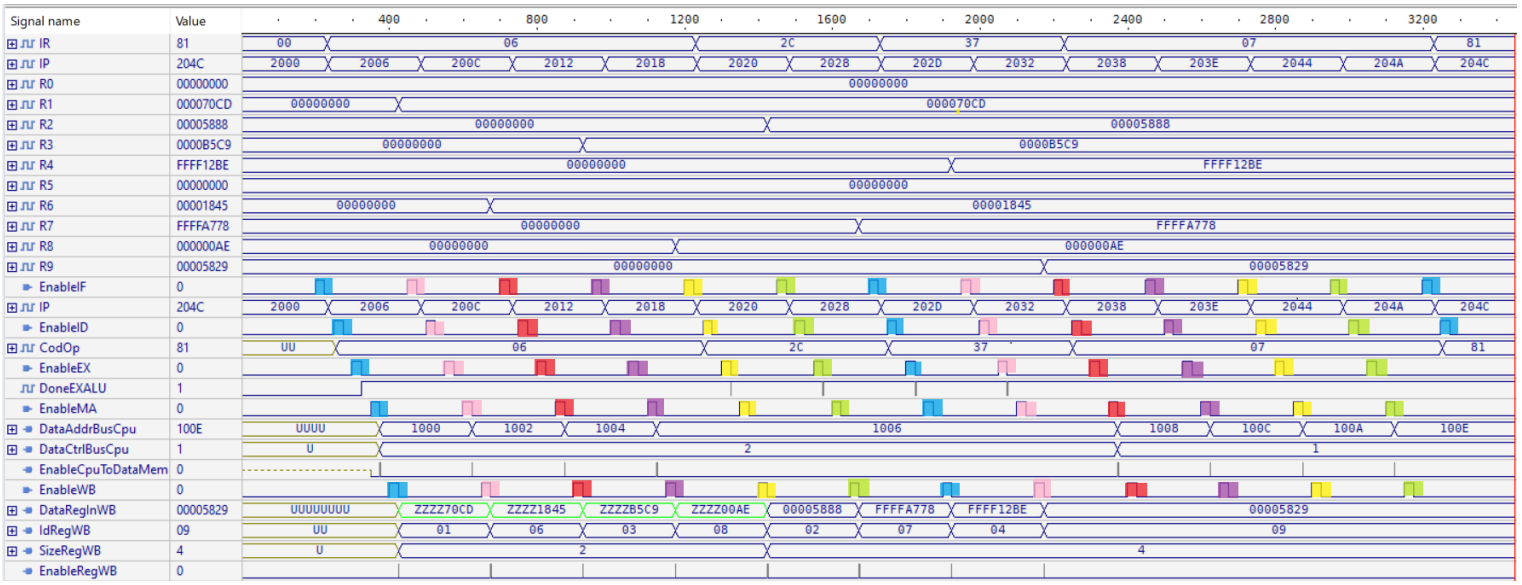
Para ver los accesos a memoria que se realizan se puede acceder a la misma mediante las señales ubicadas en:
/usuario/UUT/UUT4/Data_Memory



Aquí se puede ver otro waveform en el cual en la parte superior se ven las señales de la etapa de Memory Access que hable antes y abajo se encuentra la memoria de datos. En la cual pueden verse las escrituras que realiza el programa.

Etapa WRITE-BACK:
/usuario/UUT/UUT2/UUT2/UUT5/EnableWB
/usuario/UUT/UUT2/UUT2/UUT5/DataRegInWB
Esta señal indica que datos se deben ingresar al registro en el write-back
Gracias a esta señal se puede entender el motivo del atasco en la ejecución. Ya que se debe a que el resultado de la operación [dsubi r2, r1, 6213] no se encuentra escrito en el registro 2 para el momento en el que lo necesita la operación [xnorr r4, r2, r3] y al momento que este es escrito en el registro, la ejecución puede continuar.
/usuario/UUT/UUT2/UUT2/UUT5/IdRegWB
Esta señal indica en que registro se deben ingresar los datos
/usuario/UUT/UUT2/UUT2/UUT5/SizeRegWB
Esta señal indica que tamaño de datos se debe guardar en el registro
Para las primeras operaciones de lh, se utilizan 2 bytes para cargar 16 bits de memoria, pero luego de esto las operaciones entre registros son de 32 bits, es decir de 4 bytes.
/usuario/UUT/UUT2/UUT2/UUT5/EnableRegWB
Esta señal indica cuando debe realizarse una escritura en un registro
En esta señal se ve que todas las operaciones en esta prueba, excepto las de save a memoria utilizan el writeback para guardar datos en un registro (tambien se nota la sección de la ejecución en la que no se guardó nada por estar la misma parada)

Si se desactiva el pipelining el resultado es parecido exceptuando que las etapas no están activadas continuamente, sino que se van activando una a una y una operación no comienza hasta que la anterior ya termino completamente. Esto demuestra porque tarda mucho más tiempo en ejecutarse el programa cuando no tiene activado el pipelining.



Versión corregida del código:
Nombre del archivo: testbenchJCH2.asm

```
            .data
A:          .hword      28877
B:          .hword      6213
C:          .hword      0xB5C9
D:          .hword      0xAE
E:          .hword      0
F:          .hword      0
G:          .hword      0
H:          .hword      0
```

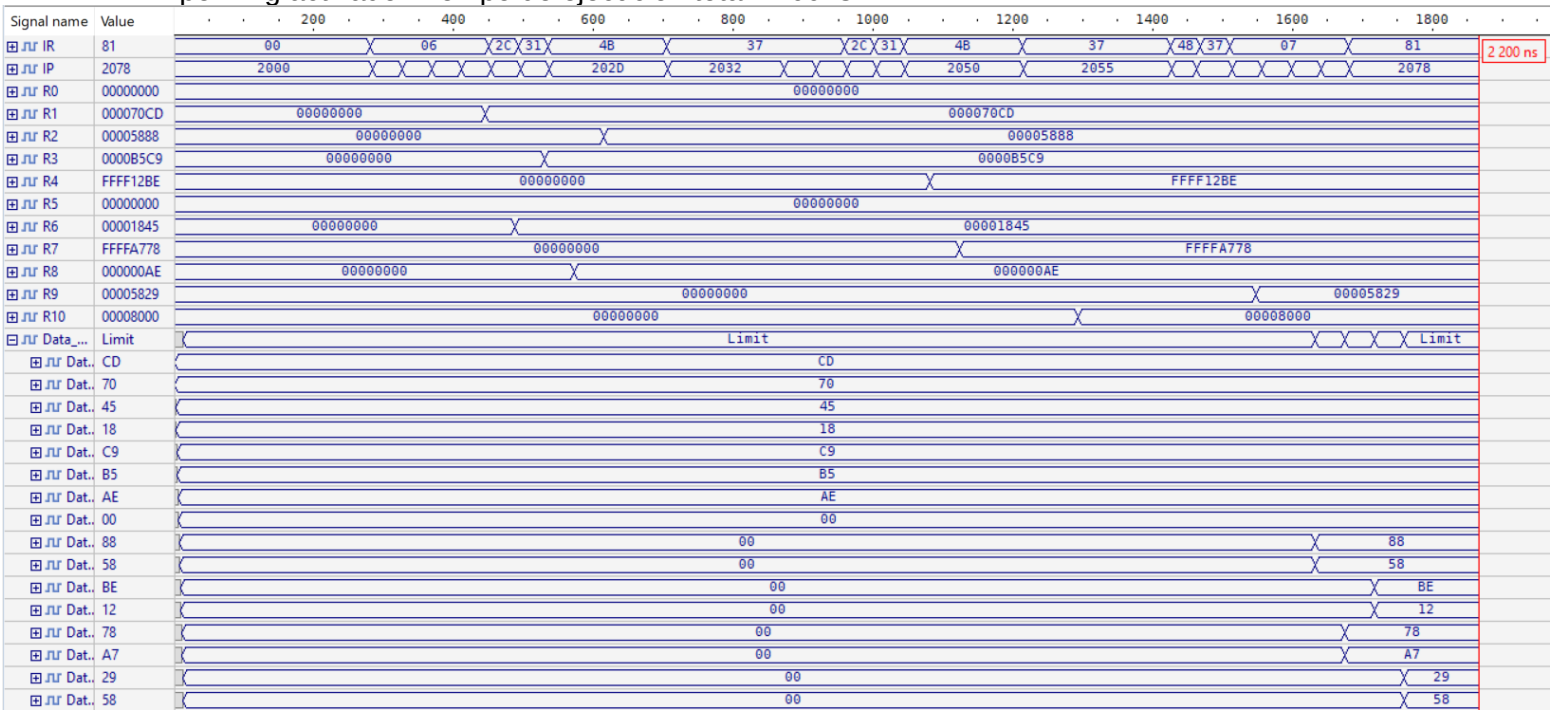
		CodOp:	Acción:
	.code		
	lh r1, A(r0)	06	r1 ← A(1000) = 28877 (0x70CD)
	lh r6, B(r0)	06	r6 ← C(1004) = 6213 (0x1845)
	lh r3, C(r0)	06	r3 ← B(1002) = 0xB5C9
	lh r8, D(r0)	06	r8 ← D(1006) = 0xAE
	dsubi r2, r1, 6213	2C	r2 = r1 - 6213
	andi r5, r2, 0x8000	31	r5 = r2 and 0x8000 (si el resultado es positivo r5=0)
	beqz r5, posit1	4C	salto a posit1 si r5=0 (si r2 es positivo)
	xnorr r4, r2, r8	37	r4 = r2 xnor 0xB5C9
	jmp num2	48	Salto incondicional a num2
posit1:	**xnorr r4, r2, r3**	37	r4 = r2 xnor 0xAE
num2:	**dsubi r7, r6, 28877**	2C	r7 = r6 - 28877
	andi r10, r7, 0x8000	31	r10 = r7 and 0x8000 (si el resultado es positivo r10=0)
	beqz r10, posit2	4C	salto a posit2 si r10=0 (si r7 es positivo)
	xnorr r9, r7, r8	37	r9 = r7 xnor 0xB5C9
	jmp fin	48	Salto incondicional a fin
posit2:	**xnorr r9, r7, r3**	37	r9 = r7 xnor 0xAE
fin:	**sh r2, E(r0)**	07	r2 → E(1008) = 0x5888
	sh r7, G(r0)	07	r7 → G(100C) = 0xA778
	sh r4, F(r0)	07	r4 → F(100A) = 0x12BE
	sh r9, H(r0)	07	r9 → H(100E) = 0x5829
	halt	81	Fin

Este código primero carga los valores en los registros al igual que lo hacia el código anterior, luego de esto realiza la primera resta (entre 28877 y 6213) y guarda el resultado en r2. Este valor si es positivo en la posición del bit más significativo tendrá un 0 y si es negativo tendrá un 1, para comprobar esto realizo un and con el valor 0x8000, el cual tiene un 1 en el bit menos significativo, por lo que si el numero contra el que se realiza la operación tiene un 0 ahí, el resultado será 0 (si es positivo) y si el numero tiene un 1 queda otro valor en el resultado. Por ello utilizando un salto puedo comprobar si e valor en el registro es 0 o no, y dependiendo de esto saber si el numero contra el que realice la operación era positivo o negativo. En mi caso realizo un salto en caso de que el registro sea 0 (es decir que sea positivo) y no salto en caso de que no lo sea.

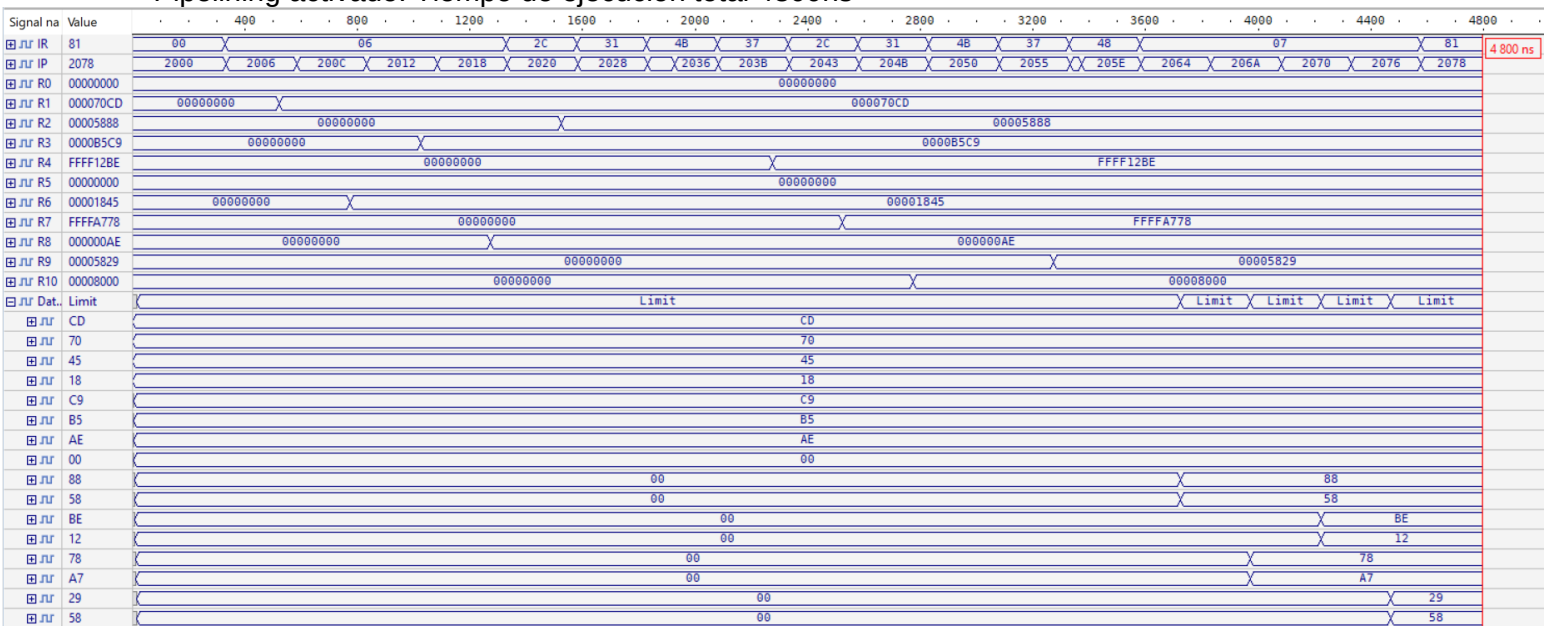
Para mostrar el correcto funcionamiento del código muestro una ejecución del programa, mostrando los registros y la memoria utilizada. Como puede observarse los resultados finales son iguales a los del primer programa que explique (deje los registros en la misma posición para que sea más sencillo de ver). Los únicos dos registros que ahora se usan que antes no eran usados son r5 y r10 para realizar la comprobación de signo antes explicada.

Para ejecutar este código la línea progName en usuario.vhd debe decir testbenchJCH2.asm, caso contrario ejecutara el primer código explicado.

Pipelining activado: Tiempo de ejecución total 2200ns

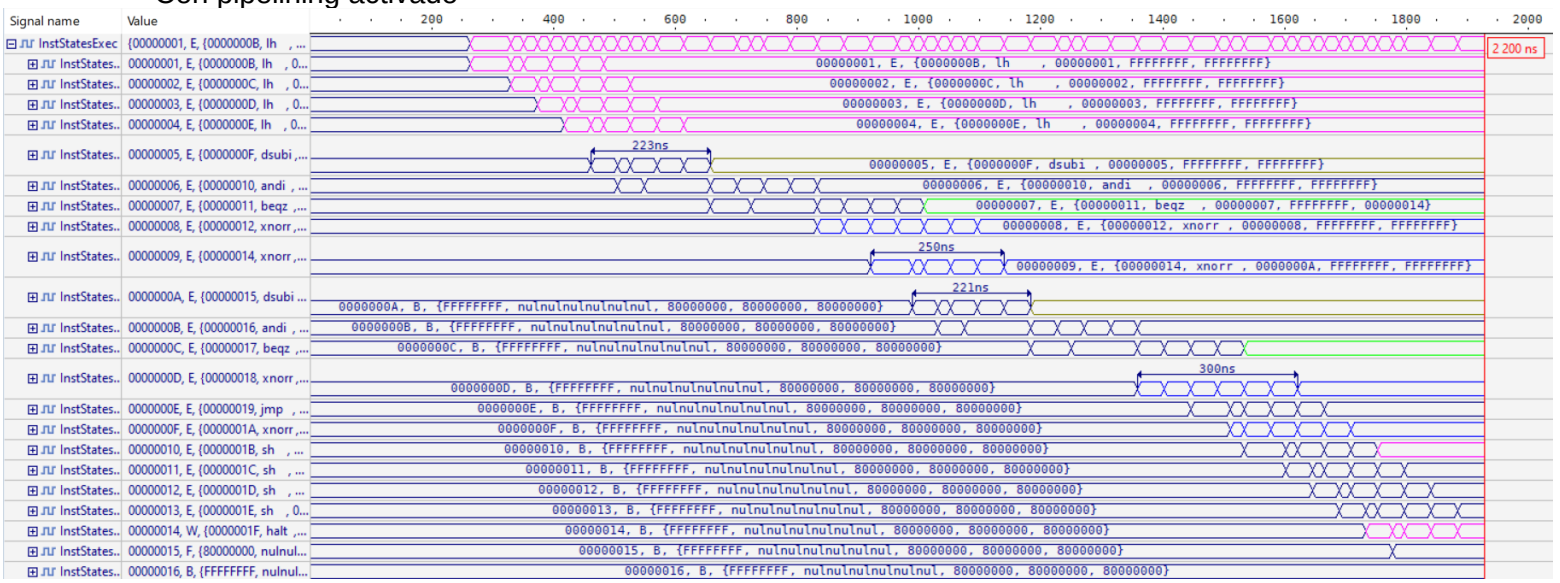


Pipelining activado: Tiempo de ejecución total 4800ns



Lo último que quedaría calcular es el tiempo que usan en la cpu cada una de las operaciones implementadas. Para ello utilizare otra señal ubicada en /usuario/UUT/UUT2/UUT1/UUT1/InstStatesExec la cual muestra todas las operaciones y en que etapa de ejecución se encuentran.

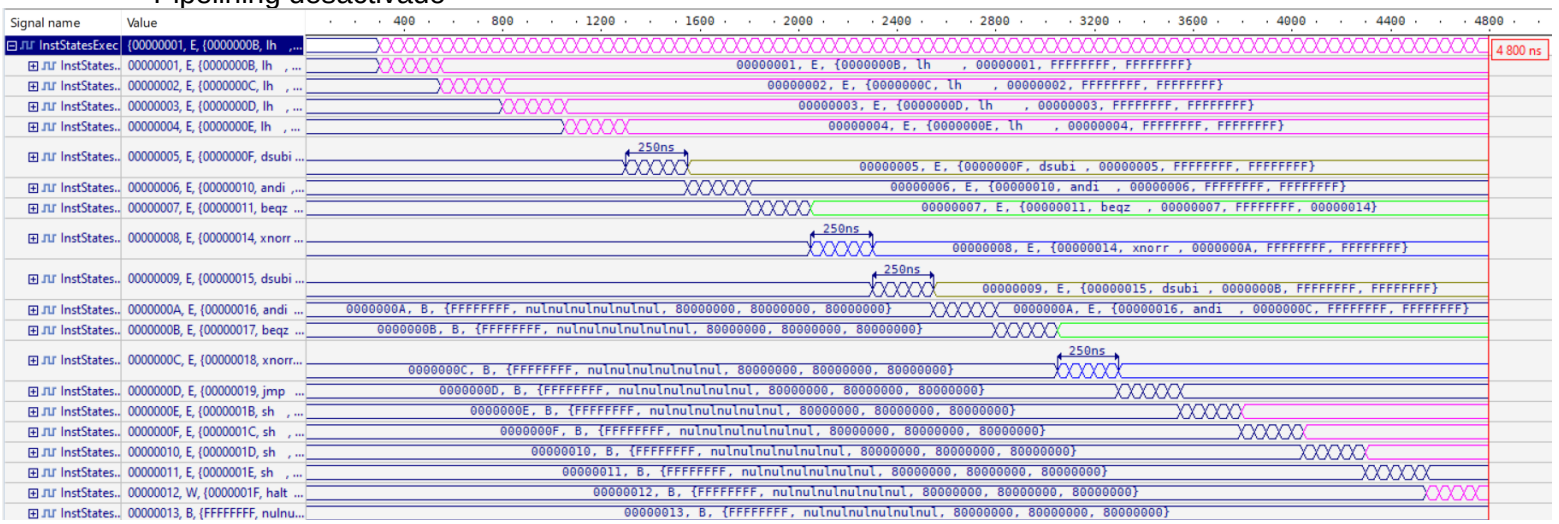
Con pipelining activado



Esos tiempos que remarque muestran las 4 veces que se ejecuta una de las operaciones que implemente en la arquitectura (los otros dos xnorr que aparecen no se ejecutan porque se realiza un salto que los interrumpe)

La etapa de execute para cada instrucción dura 50 ns, no lo marco en el grafico porque no llegan a verse los números y ensucia la imagen. Junto a los archivos entregados doy una copia de los waveform mostrados para este ultimo programa.

Pipelining desactivado



Con el pipeline desactivado dos grandes diferencias que pueden verse es que las operaciones duran todas lo mismo y que como los saltos son resueltos de antemano no aparecen operaciones xnor que no se ejecutan.

Para agregar las señales utilizadas en esta explicación se puede copiar y pegar este texto en un waveform. Dado el funcionamiento del simulador primero se debería agregar todas las señales del sistema recursivamente, borrarlas y luego pegar este texto. Dos señales que agregue que no están explicadas son la DoneID que indica todos los instantes en los que se ha terminado de realizar un decode y la StallRAW, la cual indica todos los momentos en los que la etapa de decode se encontró detenida por no contar con los datos necesarios.

/usuario/UUT/UUT2/UUT7/IR /usuario/UUT/UUT2/UUT7/IP /usuario/UUT/UUT2/UUT7/R0
/usuario/UUT/UUT2/UUT7/R1 /usuario/UUT/UUT2/UUT7/R2 /usuario/UUT/UUT2/UUT7/R3
/usuario/UUT/UUT2/UUT7/R4 /usuario/UUT/UUT2/UUT7/R5 /usuario/UUT/UUT2/UUT7/R6
/usuario/UUT/UUT2/UUT7/R7 /usuario/UUT/UUT2/UUT7/R8 /usuario/UUT/UUT2/UUT7/R9
/usuario/UUT/UUT2/UUT7/R10 /usuario/UUT/UUT2/UUT2/UUT1/EnableIF
/usuario/UUT/UUT2/UUT7/IP /usuario/UUT/UUT2/UUT2/UUT2/EnableID
/usuario/UUT/UUT2/UUT2/UUT2/CodOp /usuario/UUT/UUT2/UUT1/UUT1/DoneID
/usuario/UUT/UUT2/UUT1/UUT1/StallRAW /usuario/UUT/UUT2/UUT2/UUT3/EnableEX
/usuario/UUT/UUT2/UUT2/UUT3/DoneEXALU /usuario/UUT/UUT2/UUT2/UUT4/EnableMA
/usuario/UUT/UUT2/UUT2/UUT4/DataAddrBusCpu
/usuario/UUT/UUT2/UUT2/UUT4/DataCtrlBusCpu
/usuario/UUT/UUT2/UUT2/UUT4/EnableCpuToDataMem
/usuario/UUT/UUT2/UUT2/UUT5/EnableWB /usuario/UUT/UUT2/UUT2/UUT5/DataRegInWB
/usuario/UUT/UUT2/UUT2/UUT5/IdRegWB /usuario/UUT/UUT2/UUT2/UUT5/SizeRegWB
/usuario/UUT/UUT2/UUT2/UUT5/EnableRegWB /usuario/UUT/UUT4/Data_Memory
/usuario/UUT/UUT4/Inst_Memory /usuario/UUT/UUT4/DataAddrBusMem
/usuario/UUT/UUT2/UUT1/UUT1/InstStatesExec

La entrega del presente trabajo práctico involucra los siguientes requerimientos:
i. Un informe detallado que contenga toda la información que juzgue pertinente para la resolución de los problemas aquí planteados.
ii. Los archivos VHDL (extensión “.vhd”) del proyecto “TDA_1819” actualizados con todos los cambios que le resultaron necesarios para implementar las instrucciones propuestas en el ejercicio 1. No se requiere enviar el espacio de trabajo o workspace completo, alcanza con especificar en el informe los nombres de los ficheros modificados junto con las rutas en las cuales deberán ser ubicados dentro del proyecto original para la correcta simulación del procesador.
iii. El nuevo programa desarrollado en el lenguaje Assembler tal como es solicitado en el ejercicio 2 (extensión “.asm”). Se recomienda asimismo en este caso incluir en el informe la ruta del mencionado programa en el proyecto “TDA_1819” para evitar cualquier inconveniente que pudiera presentarse durante el ensamblaje y/o la ejecución del mismo.

Fecha límite de entrega por mensajería de IDEAS: viernes 24 de noviembre de 2023 a las 11:59:59 hs. (mediodía)