Proyecto 1 – 2025 – Gestión de riesgos y excepciones en MIPS

Langa Ibáñez, Ana - 839811
Pastor Peralta, Fernando - 897113

lunes, 31 de marzo de 2025

Resumen:

Este proyecto tiene como objetivo aprender a trabajar con VHDL, un lenguaje de descripción de hardware en un entorno de simulación profesional. En este caso, se va a trabajar con un procesador MIPS de 32 bits segmentado en 5 etapas. Para el desarrollo de este proyecto, se han adquirido conocimientos previos proporcionados por la asignatura 'AOC II'.

La primera parte del proyecto, consiste en modificar el procesador MIPS proporcionado por el profesorado para soportar las dos instrucciones añadidas en la práctica 3, además de una nueva instrucción adicional. Esto implica modificar tanto la unidad de control como la ruta de datos.

En segundo lugar, se deben gestionar los riesgos de datos que puedan ocurrir durante la ejecución del procesador.

Como tercer paso, es necesario gestionar los riesgos de control y establecer los contadores que nos ayudarán a monitorear el rendimiento del procesador.

Tras completar los pasos anteriores, se deben verificar los manejos de excepciones.

Por último, se deben realizar pruebas para verificar cada uno de los pasos y el diseño general. Estas pruebas se llevarán a cabo tanto con los test proporcionados como con pruebas realizadas por nosotros mismos.

Nota:

1. report "Simulation time : " & time'IMAGE(now) & ". Data written: " & integer'image(to_integer(unsigned(Din))) & ", in ADDR = " & integer'image(to_integer(unsigned(ADDR))); se ha comentado en el archivo RAM_128_32_P1_2025.vhd

2. report "Simulation time : " & time'IMAGE(now) & ". Data written: " & integer'image(to_integer(signed(BusW))) & ", in Reg = " & integer'image(to_integer(unsigned(RW))); se ha comentado en el archivo BReg.vhd

3. Las RAM de los test que hemos realizado no tienen las 4 primeras palabras reservadas ya que no usamos IRQ en ninguno de los programas.

Verificación de los requisitos funcionales:

○ RF1 JAL y RET

Descripción: Incluir las instrucciones JAL y RET ya trabajadas en prácticas en el MIPS del proyecto.

¿Cómo?

Para implementar la funcionalidad de estas dos nuevas instrucciones, hemos realizado cambios tanto en la ruta de datos como en la unidad de control.

Para la instrucción JAL, en la ruta de datos, hemos extendido la señal PC4 y la señal del jal hasta la etapa WB. De esta manera conseguimos almacenar en el registro correspondiente PC4. Por otro lado, el RET no necesita modificaciones en la ruta de datos ya que el salto se produce en la etapa ID.

En la unidad de control hemos tenido que añadir señales para el JAL:

- Encendemos la señal jal para indicar que es una instrucción JAL.
- Mantenemos la señal RegDst a 0 ya que queremos almacenar la dirección de retorno en rt.
- Encendemos la señal RegWrite para escribir en el banco de registros.
- Mantenemos la señal MemtoReg a 0, ya que queremos guardar el PC4_WB en el banco de registros

Verificación: Comprobar su correcto funcionamiento en un banco de pruebas.

-Nombre RAM de instrucciones: NUESTRO TEST PARA LAS INSTRUCCIONES JAL Y RET

-Nombre RAM de datos: DATOS TEST DEL JAL Y RET

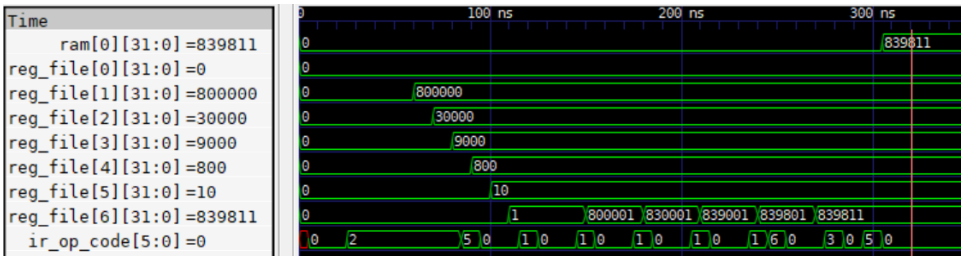
-Código ensamblador:

```
0-lw r1, 48(r6)   # @12 → 48 en bytes ->08C10030  --Cargo el 800000
1-lw r2, 52(r6)   # @13 → 52 en bytes ->08C20034  --Cargo el 30000
2-lw r3, 56(r6)   # @14 → 56 en bytes ->08C30038  -- 9000
3-lw r4, 60(r6)   # @15 → 60 en bytes ->08C4003C  -- 800
4-lw r5, 64(r6)   # @16 → 64 en bytes ->08C50040  -- 1
5-lw r6, 68(r6)   # @17 → 68 en bytes ->08C60044
6-jal r10, 4                  # 140A0004
7-nop                         # 00000000
8-sw r6,0(r0)                 # 0C060000
9-nop                         # 00000000
10-fin:jal fin                # 18000000
11-nop                        # 00000000
12-sumasbr:add r6, r6, r1# 04263000
13-nop                        # 00000000
14-nop                        # 00000000
15-add r6, r6, r2             # 04463000
16-nop                        # 00000000
17-nop                        # 00000000
18-add r6, r6, r3             # 04663000
19-nop                        # 00000000
20-nop                        # 00000000
21-add r6, r6, r4             # 04863000
22-nop                        # 00000000
23-nop                        # 00000000
24-add r6, r6, r5             # 04a63000
25-ret r10                    # 19400000
26-nop                        # 00000000
```

Este es el código de la práctica 3. Para su prueba utilizamos otra memoria de datos que está incluida en el fichero → RAM_128_32_P1_2025.vhd

Podemos observar que el valor final de la ram[0] es 839811

-Captura de GTKWave:



○ RF2 LW_INC

Descripción: Incluimos la instrucción LW_INC al MIPS del proyecto.

```
lw inc rt, imm(rs) (Atomic read-increment), which performs the following actions:
                    rt <= Mem(rs + SignExt(imm))
                    Mem(rs + SignExt(imm)) <= Mem(rs + SignExt(imm)) +1
                    PC <= PC + 4
```

¿**Cómo**?

En la UC hemos puesto esto:

```
WHEN  FI_opcode  =>  f_inc <= '1'; ALUSrc <= '1'; MemRead <= '1'; MemtoReg <= '1'; RegWrite <= '1';
```

- ALUSrc: La ALU utiliza un valor inmediato extendido como segundo operando
- MemRead: Se realiza una lectura desde la memoria de datos
- MemtoReg: El valor leído desde la memoria se escribe en el banco de registros
- RegWrite: Se habilita la escritura en un registro

Verificación:

-Nombre RAM de instrucciones: NUESTRO TEST LW-INC

-Nombre RAM de datos: DATOS ORIGINALES

-Código ensamblador:

```
LW_INC r0, 8(r1) ---> 40200008
NOP
LW_INC r1, 4(r4) ---> 40810004
NOP
NOP
ADD r2, r0, r1 ---> 04011000
NOP
NOP
SW r2, 16(r4) ---> 0C820010
BEQ r0, r0, -1   --> 1000FFFF
```

Obtenemos el resultado esperado:

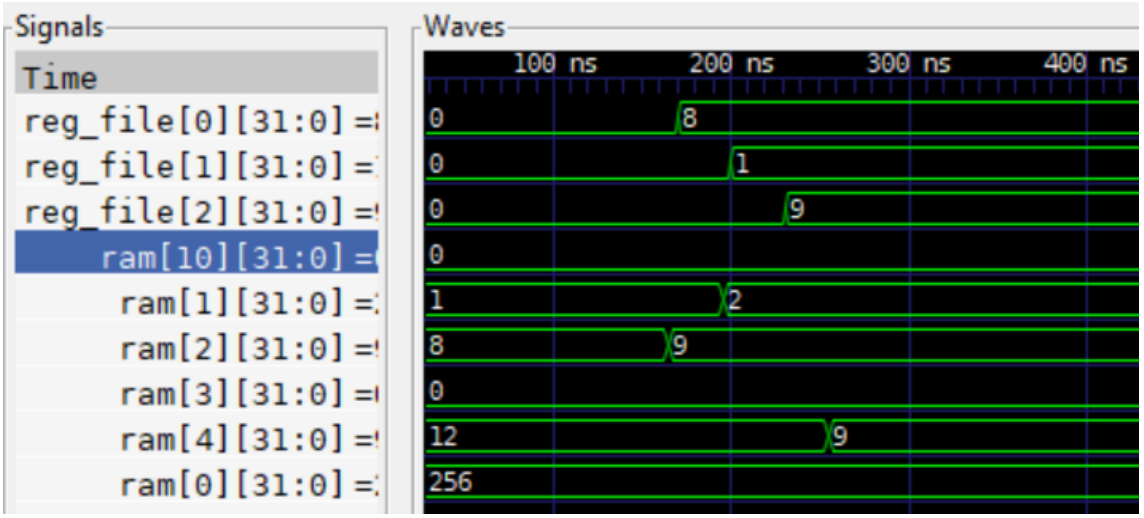
r0 = 8 — Mem[2]=8+1=9

r1 = 1 — Mem[1]=1+1=2

r2 = r2+r1=8+1=9

Mem[4] = 9

-Captura de GTKWave:



○ RF3 Anticipación de operandos

Descripción: El procesador es capaz de anticipar los operandos para las instrucciones LW, SW y ARIT a distancia 1 (siempre que el dato a anticipar no venga de un lw) y 2.

¿Cómo?

La anticipación de operandos se realiza en la etapa EX. Sirve para anticipar operandos que están en la etapa WB o en MEM y de esta manera evitar riesgos de datos y reducir ciclos de espera.

Las señales *Corto_A_Mem, Corto_B_Mem, Corto_A_WB, Corto_B_WB* sirven para determinar si se debe hacer una anticipación de operandos desde la etapa MEM o desde la etapa WB. Los A son para el registro rs y los B para el registro rt.

Para todas estas señales hay que tener en cuenta tres condiciones clave:

1. Se verifica si el registro fuente (Rs o Rt) de la instrucción en la etapa EX coincide con el registro de destino (RW_MEM o RW_WB) de una instrucción en una etapa previa (MEM o WB).

   Si hay coincidencia, significa que el dato que necesita la instrucción en **EX** está siendo producido por una instrucción previa que todavía no ha escrito en el banco de registros.

2. Se comprueba si la instrucción en MEM o WB realmente tiene activada la escritura en registros.

   Si la instrucción previa no escribe en un registro, entonces su resultado no es relevante para la anticipación de datos.

3. Se verifica que la instrucción en la etapa previa sea válida.

   Si la instrucción no es válida, no queremos hacer anticipación de datos de datos inválidos.

También tenemos dos señales de control de MUX que controlan el dato correcto que debe tener la ALU en la etapa EX:

- MUX_ctrl_A <= "01" when (Corto_A_Mem = '1') else
  "10" when (Corto_A_WB = '1') else
  "00";

  Controla el dato que se enviará como operando A (rs)

- MUX_ctrl_B <= "01" when (Corto_B_Mem = '1') else

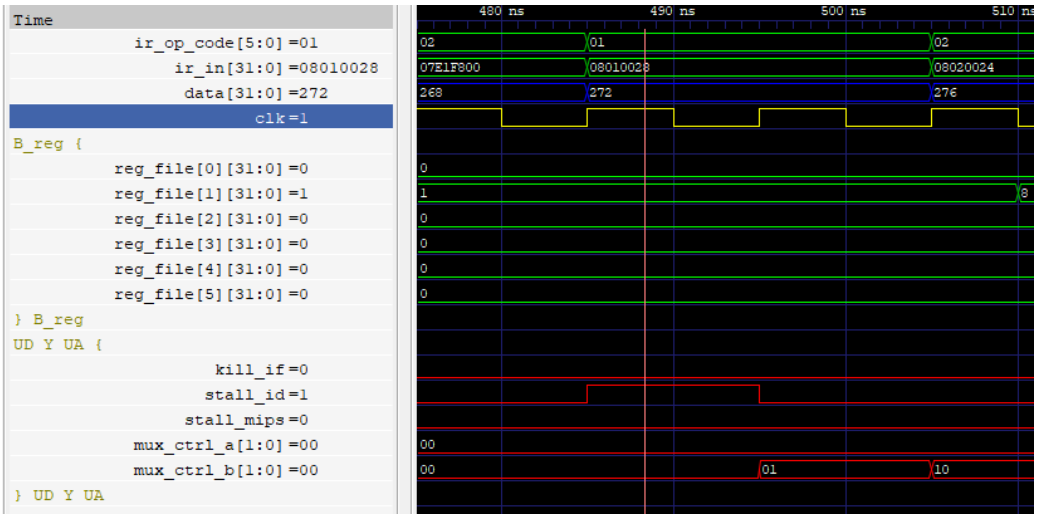
  Controla el dato que se enviará como operando B (rt)

"10" when (Corto_B_WB = '1') else

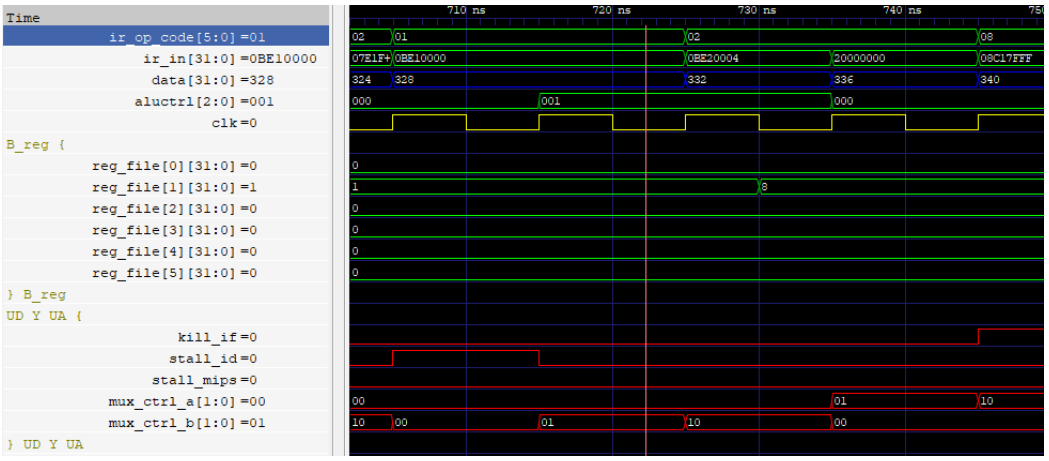
"00";

Verificación:

- ***En el banco de pruebas test_IRQ se realizan las siguientes anticipaciones:***
  - De LW R1, 8(R0) a ADD r31, R1, R31: anticipación de Rs distancia 2 (tras detención de un ciclo del ADD):
    Al haber dependencia en la siguiente instrucción (por el registro r1), detenemos el pipeline durante 1 ciclo (guardando el estado de todo lo que implica a las siguientes instrucciones), ld_uso_rs = 1 => stall_ID = 1.
    Tras la detención de ID, nuestra unidad de anticipación ya será capaz de anticipar el resultado del LW a nuestra instrucción ADD, Corto_A_Mem = 1 => MUX_ctrl_A = 01.



  - De SUB r31, R31,R1 a LW R1, 0(R31): anticipación de Rs distancia 1:
    SUB es productor en el registro r31, y LW consume en rs el registro r31. Aquí actuará nuestra UA, para darle el valor correcto al LW, ya que aún no se ha escrito en r31 el valor de la resta y actualmente si accediesemos al banco de registros, nos encontraríamos con el valor anterior de r31 (un valor erróneo que coger).
    Nuestra UA detectará que hay mismos registros usados en la instrucción a ejecutar y Mem, y que se va a escribir en el BR, entonces le dará a el resultado de la resta a LW en rs mediante las señales: Corto_A_Mem = 1 => MUX_ctrl_A = 01.



- ***Test de ADD R3, R1, R2 a SW R3, 80(R0):***

  -Nombre RAM de instrucciones: NUESTRO TEST PRODUCTOR CONSUMIDOR-SW;

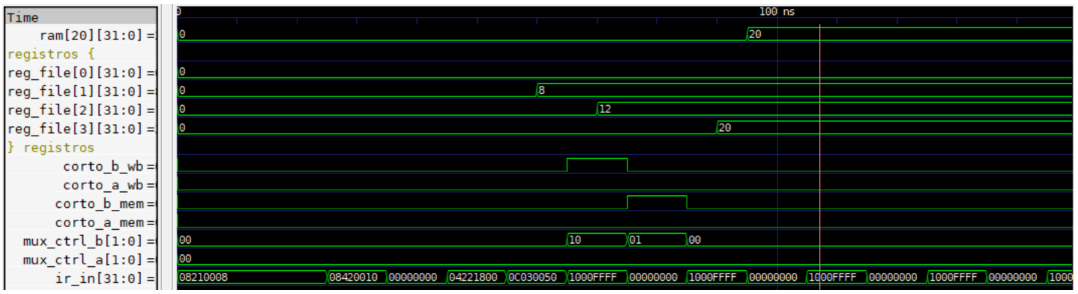
  -Nombre RAM de datos: DATOS ORIGINALES

  -Código ensamblador:

LW r1, 8(r1) ;	→ r1 = 8	→ X"08210008"
LW r2, 16(r2) ;	→ r2 = 12	→ X"08420010"
NOP ;		→ X"00000000"
ADD r3, r1, r2;	→ r3 = r1 + r2 = 20	→ X"04221800"
SW r3, 80(r0) ;	→ ram[20] = 20;	→ X"0C030050"
BEQ r0, r0, -1		→ X"1000FFFF"

El ADD es productor en r3, por otro lado, SW es consumidor en r3 por Rt: Corto_B_Mem = 1 => MUX_ctrl_B = 01, (el Corto_B_WB anterior es causado por el LW y ADD).

-Captura de GTKWave:



- ***Nuestro test de la UA:***

  -**Nombre RAM de instrucciones:** NUESTRO TEST DE LA UA

  -Nombre RAM de datos: DATOS TEST DE UA Y UD

  -Código ensamblador:

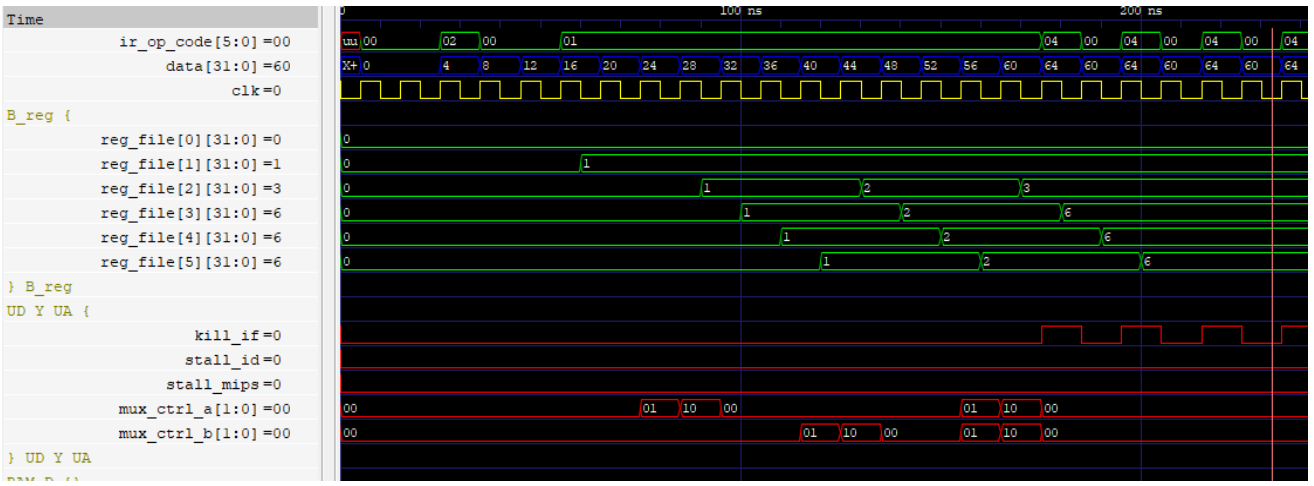
```
;--------PARA COMPROBAR LA UA--------
lw r1, 0(r0)
NOP
NOP

;CONSUMIDORES EN rs (A):
add r2, r1, r0 ; r2 = 1 PRODUCTOR EN R2
add r3, r2, r0 ; r3 = 1 1º CONSUMIDOR DE R2,  Corto_A_Mem = 1 => MUX_ctrl_A = 01
add r4, r2, r0 ; r4 = 1 2º CONSUMIRDOR EN R2, Corto_A_WB = 1  => MUX_ctrl_A = 10
add r5, r2, r0 ; r5 = 1 LA UA NO ACTUA                        MUX_ctrl_A = 00

;CONSUMIDORES EN rt (B):
add r2, r1, r1 ; r2 = 2 PRODUCTOR EN R2
add r3, rO, r2 ; r3 = 2 1º CONSUMIDOR DE R2,  Corto_B_Mem = 1 => MUX_ctrl_B = 01
add r4, r0, r2 ; r4 = 2 2º CONSUMIRDOR EN R2, Corto_B_WB = 1  => MUX_ctrl_B = 10
add r5, r0, r2 ; r5 = 2 LA UA NO ACTUA                        MUX_ctrl_B = 00

;CONSUMIDORES EN rs (A) Y rt (B):
add r2, r1, r2 ; r2 = 3 PRODUCTOR EN R2
add r3, r2, r2 ; r3 = 6 1º CONSUMIDOR DE R2,  Corto_A_Mem = 1 && Corto_B_Mem = 1 => MUX_ctrl_A = 01 && MUX_ctrl_B = 01
add r4, r2, r2 ; r4 = 6 2º CONSUMIRDOR EN R2, Corto_A_WB = 1  && Corto_B_WB = 1  => MUX_ctrl_A = 10 && MUX_ctrl_B = 10
add r5, r2, r2 ; r5 = 6 LA UA NO ACTUA                                            MUX_ctrl_A = 00 && MUX_ctrl_B = 00
beq r0, r0, -1 ; fin
```

-Captura de GTKWave:



○ RF4 Riesgos de datos

Descripción: El procesador es capaz de detener la ejecución para evitar los riesgos causados por las dependencias en instrucciones lw-uso (distancia 1), beq o WRO (distancias 1 o 2).

¿Cómo?

La unidad de detección de riesgos se encuentra en la etapa de decodificación. Cuando se encuentre un riesgo de datos, se debe activar *stall_ID,* siempre que la instrucción en ID sea válida. De esta manera conseguimos parar la etapa ID y la IF hasta que el riesgo desaparece. Para llegar hasta la activación de esta señal se comprueban varias fases anteriores:

- Identificación del uso de registros: El procesador indica si la instrucción en ID lee los registros Rs o Rt mirando el op_code.

```
rs_read <= '1' when ((IR_op_code = ARIT_opcode) or (IR_op_code = LW_opcode) or (IR_op_code = SW_opcode) or
                     (IR_op_code = BEQ_opcode) or (IR_op_code = RET_opcode) or (IR_op_code = FI_opcode)) else '0';
rt_read <= '1' when ((IR_op_code = ARIT_opcode) or (IR_op_code = SW_opcode) or (IR_op_code = BEQ_opcode) or
                     (IR_op_code = FI_opcode)) else '0';
```

- Detección de dependencias de datos: El procesador detecta si hay dependencia con instrucciones en EX o MEM.

```
dep_rs_EX <= '1' when ((valid_I_EX = '1') and (valid_I_ID = '1') and (Reg_Rs_ID = RW_EX) and (RegWrite_EX = '1') and
                (rs_read = '1'))   else '0';
dep_rs_Mem<= '1' when ((valid_I_MEM = '1') and (valid_I_ID = '1') and (Reg_Rs_ID = RW_Mem) and (RegWrite_Mem = '1')
                    and (rs_read = '1'))       else '0';
dep_rt_EX<= '1' when ((valid_I_EX = '1') and (valid_I_ID = '1') and (Reg_Rt_ID = RW_EX) and (RegWrite_EX = '1') and
(rt_read = '1'))            else '0';
dep_rt_Mem<= '1' when ((valid_I_MEM = '1') and (valid_I_ID = '1') and (Reg_Rt_ID = RW_Mem) and (RegWrite_Mem = '1')
and (rt_read = '1'))       else '0';
```

- Confirmar si las dependencias son riesgos de datos: Se utilizan para detectar si las dependencias de datos son riesgos de datos en la instrucción correspondiente.

```
ld_uso_rs <= '1' when (MemRead_EX = '1' and (Reg_Rs_ID = RW_EX) and (rs_read = '1')) else '0';
ld_uso_rt <= '1' when (MemRead_EX = '1' and (Reg_Rt_ID = RW_EX) and (rt_read = '1')) else '0';
BEQ_rs  <= '1' when ((IR_op_code = BEQ_opcode) and ((dep_rs_EX = '1') or (dep_rs_Mem = '1'))) else '0';
BEQ_rt  <= '1' when ((IR_op_code = BEQ_opcode) and ((dep_rt_EX = '1') or (dep_rt_Mem = '1'))) else '0';
RET_rs  <= '1' when ((IR_op_code = RET_opcode) and ((dep_rs_EX = '1') or (dep_rs_Mem = '1'))) else '0';
JAL_uso_rs      <= '0';
JAL_uso_rt <= '0';
```

- Activación de stall_ID: Si alguna de las señales del punto anterior se activa, riesgo_datos_ID se activa y stall_ID también.

```
riesgo_datos_ID <= BEQ_rt OR BEQ_rs OR ld_uso_rs OR ld_uso_rt OR RET_rs OR JAL_uso_rs OR JAL_uso_rt;
stall_ID <= riesgo_datos_ID;
```

También podemos encontrar la señal para parar todo el procesador *stall_MIPS*

```
stall_MIPS_internal <= not(IO_MEM_ready)and valid_I_MEM;
stall_MIPS <= stall_MIPS_internal;
```

Esta señal se activa cuando la memoria no está preparada y la instrucción de la etapa MEM es válida.

<u>Verificación:</u> Verificamos saltos de beq, jal y rat

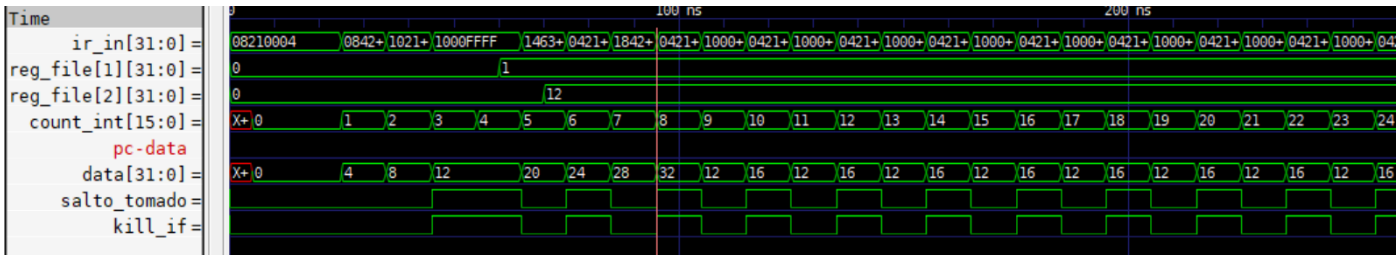
-Nombre RAM de instrucciones: NUESTRO TEST DE SALTOS

-Nombre RAM de datos: DATOS ORIGINALES

-Código ensamblador:

```
LW r1, 4(r1)      ; r1 = 1
LW r2, 16(r2)     ; r2 =12
BEQ r1, r1, 2     ; Salta JAL @20
BEQ r0, r0,-1     ; Salta a @12
ADD r1, r1, r1    ; r1 = r1 + r1
JAL r3, 1         ; Salta a RET @28
ADD r1, r1, r1    ; r1 = r1 + r1
RET r4            ; Salta a @12
ADD r1, r1, r1    ; r1 = r1 + r1
```

-Captura de GTKWave:



- ***Nuestro test de la UD (PARTE 1):***

  -Nombre RAM de instrucciones: NUESTRO TEST DE LA UD

  -Nombre RAM de datos: DATOS TEST DE UA Y UD

  -Código ensamblador:

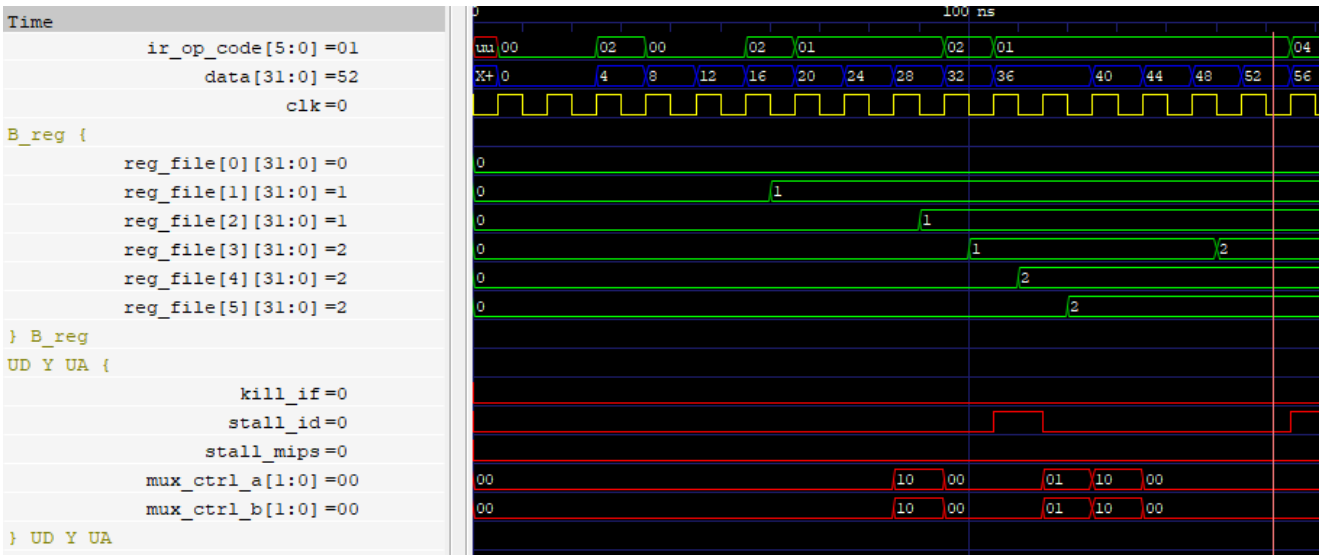
```
;--------PARA COMPROBAR LA UD--------
lw r1, 0(r0)
NOP
NOP
;LW-USO

;PRODUCTOR CONSUMIDOR A DISTANCIA 2 Y 3
lw r2, 0(r0)   ; r2 = 1
add r3, r0, r1 ; r3 = 1 NO PRODUCE DEPENDENCIAS CON EL LOAD
add r4, r2, r2 ; r4 = 2 RESUELTO POR LA UA: Corto_A_WB = 1  && Corto_B_WB = 1  => MUX_ctrl_A = 10 && MUX_ctrl_B = 10
add r5, r2, r2 ; r5 = 2 OPERANDOS YA DISPONIBLES DESDE FLANCO DESCENDENTE EN EL BR, NO PRODUCE RIESGOS

;PRODUCTOR CONSUMIDOR A DISTANCIA 1, 2 Y 3
lw r2, 0(r0)   ; r2 = 1 AL HABER DEPENDECIA EN LA SIGUIENTE INSTRUCCIÓN DETENEMOS PIPELINE DURANTE 1 CICLO
               ;(GUARDANDO EL ESTADO DE TODO), Id_uso_rs = 1 OR Id_uso_rt = 1 => stall_ID = 1
add r3, r2, r2 ; r3 = 2 RESUELTO POR LA UA: Corto_A_Mem = 1  && Corto_B_Mem = 1  => MUX_ctrl_A = 01 && MUX_ctrl_B = 01
add r4, r2, r2 ; r4 = 2 RESUELTO POR LA UA: Corto_A_WB = 1  && Corto_B_WB = 1  => MUX_ctrl_A = 10 && MUX_ctrl_B = 10
add r5, r2, r2 ; r5 = 2 SIN RIESGOS
```

(Este programa hace más cosas, se verá la segunda parte del programa más adelante)

-Captura de GTKWave:



○ RF5 Riesgos de control

Descripción: El procesador es capaz de eliminar los riesgos de control causados por las instrucciones de salto: BEQ, JAL, RET y RTE.

¿Cómo?

Según el diseño de nuestro MIPS, antes de saber si vamos a hacer un salto en una instrucción de salto, empezamos con la ejecución de la siguiente instrucción (ya que la segmentación nos lo permite).

En caso de que estemos en un BEQ y no tomemos un salto, el mecanismo es muy útil, ya que simplemente se seguiría ejecutando la instrucción y no perderemos tiempo en empezarla, sin embargo, si hacemos un salto, habríamos comenzado a ejecutar una instrucción que no queremos ejecutar. Ahí es donde entra nuestra unidad de detección de riesgos (UD), que mediante la señal Kill_IF, puede matar a la instrucción que se estaba ejecutando y que no queremos que se ejecute.

Para que se active Kill_IF se debe cumplir:

1. La señal salto_tomado está activa:

   ```
   salto_tomado <= ((Z AND Branch_ID) or RTE_ID or jal_ID or ret_ID);
   ```

   El código muestra que la señal solo se activa si beq salta “Z AND Branch_ID” y siempre que estemos ejecutando una instrucción JAL, RET o una RTE.

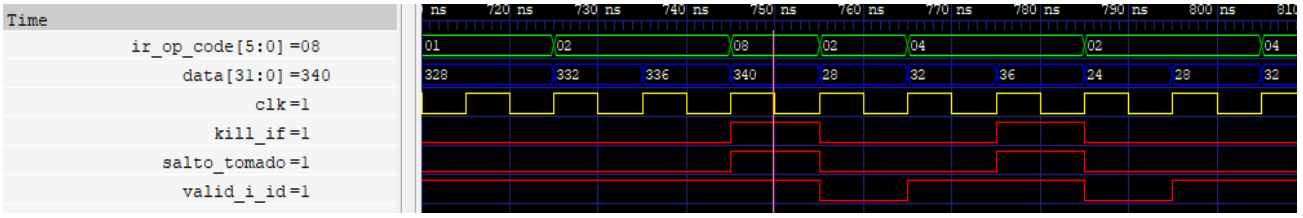
   Como se ve, esta señal nos indica si hay salto o no en la etapa ID, por ello se permite que se empiece a ejecutar una instrucción antes del salto

2. valid_I_ID = 1:

   Es decir, si la instrucción es válida en ID.

Verificación:

- ***En el banco de pruebas test_IRQ se realizan varios saltos tomados en los que sólo se ejecutan las instrucciones adecuadas:***
  Como podemos observar en la imágen, cuando hay una instrucción RTE, salto tomado se pone a uno y mata la instrucción que había en la etapa IF. Se va a la instrucción indicada en el salto.



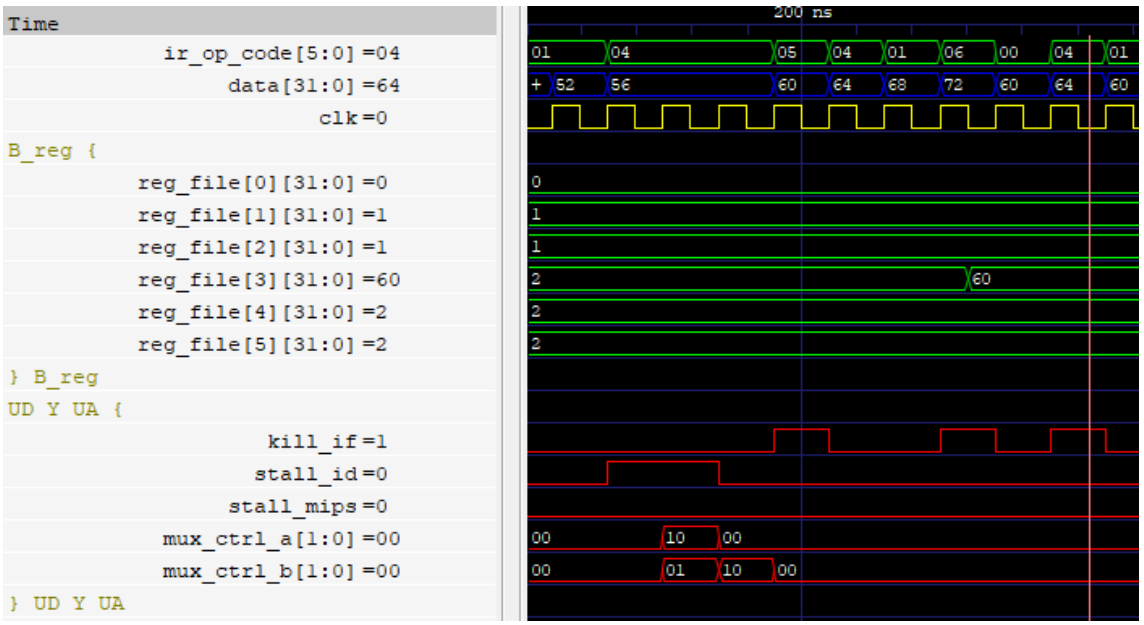
- ***Nuestro test de la UD (PARTE 2):***
  -Nombre RAM de instrucciones: NUESTRO TEST DE LA UD
  -Nombre RAM de datos: DATOS TEST DE UA Y UD
  -Código ensamblador:

```
;--------PARA COMPROBAR LA UD--------
;BEQ Y RET CONSUMIDORES
;BEQ CONSUMIDOR
add r2, r1, r0 ; r2 = 1 PRODUCTOR EN R2
add r3, r1, r1 ; r3 = 2 PRODUCTOR EN R3
beq r2, r3,  0 ; CONSUMIMOS DE AMBOS PRODUCTORES, AL SER DOS, EL PIPELINE TENDRÁ DEPENDENCIAS EN Mem Y WB
          ;EN LOS 2 CICLOS SIGUIENTES, DESPLAZAMOS EL PIPELINE (AL IGUAL QUE ANTES) 2 CICLOS.
          ; (1º CICLO:(BEQ_rs = 1(por Mem) BEQ_rt = 1(por EX)), 2º CICLO:(BEQ_rt = 1(por Mem))) => stall_ID = 1 (DURANTE 2 CICLOS)
          ; SI SOLO FUESE CONSUMIDOR DE 1, EL PIPELINE SE DESPLAZARÍA SOLO 1 CICLO.

;RET CONSUMIDOR
jal r3, 1     ; tenemos que ir a una subrutina para probar el ret
beq r0, r0, -1 ; fin
;subrutina del jal
add r2, r1, r0 ; r2 = 1 PRODUCTOR EN R2
ret r3        ; CONSUMIMOS EL PRODUCTOR ANTERIOR, PASARÁ LO MISMO QUE ANTES.
          ;(1º CICLO:(RET_rs = 1(por Mem)), 2º CICLO:(RET_rs = 1(por Mem))) => stall_ID = 1 (DURANTE 2 CICLOS)
```

(Esta es la segunda parte del programa que continúa mencionado antes, por lo que habrá movimientos de señales previos a la ejecución de este código)

-Captura de GTKWave:



○ RF6 cambio a modo excepción

Descripción: Si el procesador está en modo usuario y recibe IRQ, ABORT o UNDEF pasa al modo correspondiente y ejecuta la rutina que indica la tabla de vectores de excepción. Si está en modo excepción ignora las señales hasta volver a modo usuario.

¿Cómo?

Cuando el procesador detecta una excepción, debe guardar la dirección de la instrucción en curso para poder retornar correctamente después de manejar la excepción. Una excepción se

procesa cuando se recibe una de las señales IRQ, Data_abort o Undef, las excepciones están habilitadas, *Mips_status(1)=0'* y el procesador no está parado *'stall_MIPS'.*

```
Exception_accepted_internal <= '1' when (((IRQ = '1') or ((Data_abort = '1')and (valid_I_MEM = '1')) or (UNDEF = '1')) AND
                                        (MIPS_status(1)='0') AND (stall_MIPS = '0')) else '0';
Exception_accepted <= Exception_accepted_internal;
```

Cuando se procesa una excepción las instrucciones de MEM y WB continúan y el resto se matan. Para retornar se debe elegir la siguiente instrucción válida. Para ello tenemos sus direcciones almacenadas en *PC_exception_EX* y *PC_exception_ID*, y sus bits de validez en *valid_I_EX* y *valid_I_ID.* Si no hay válidas se elige el valor del PC. *Exception_LR* almacena la dirección a la que hay que retornar tras una excepción.

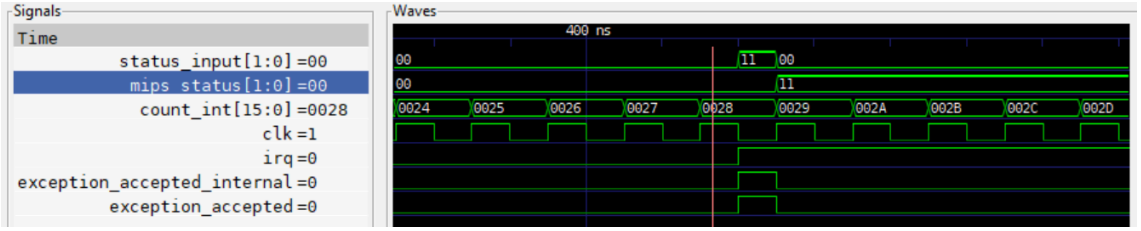
En el PC se pueden cargar:

```
Return_I <= PC_exception_EX when ((valid_I_EX = '1')AND(RTE_EX = '0')) else
            PC_exception_ID when (valid_I_ID = '1') else
            PC_out;
```

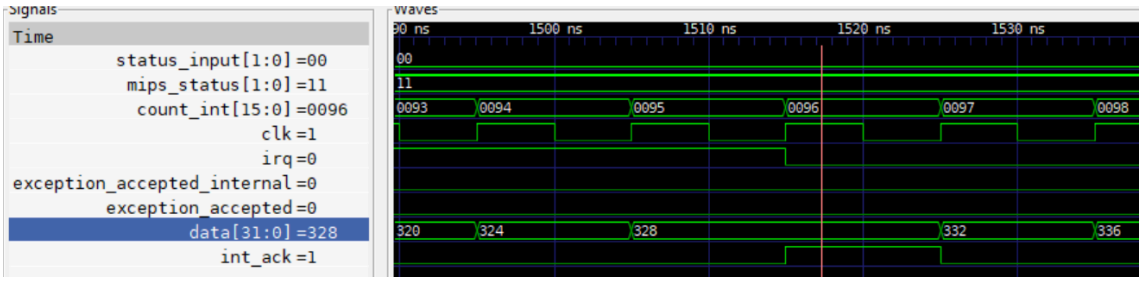
Añadimos *RTE_EX=0* ya que el retorno se hace en el ID por lo tanto, la instrucción ya se ha ejecutado por completo y ha perdido la información.

<u>Verificación</u>:

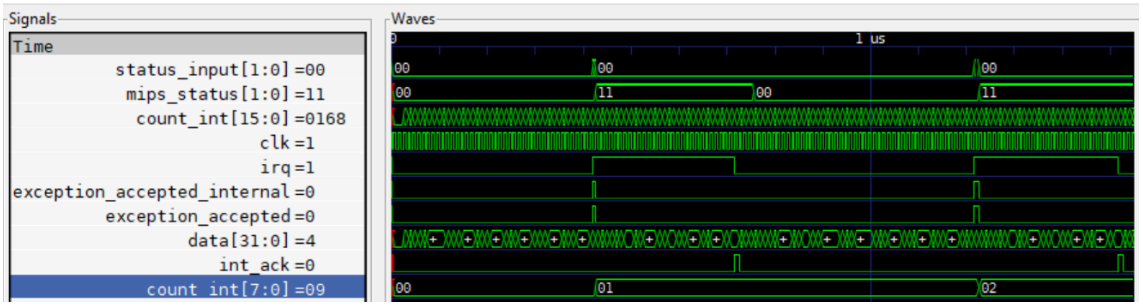
- ***IRQ: banco de pruebas test_IRQ en el que se realizan diversas IRQs. Todas ellas se atienden si y sólo si el procesador está en modo usuario. Se ejecuta una RTI que contabiliza el número de IRQs. Se verifica su funcionamiento correcto.***
  Como podemos observar en la imagen, la IRQ se atiende ya que el procesador está en modo usuario.



  En esta imagen podemos observar que int_ack se pone a 1 tal y como se indica en el código del test_IRQ.



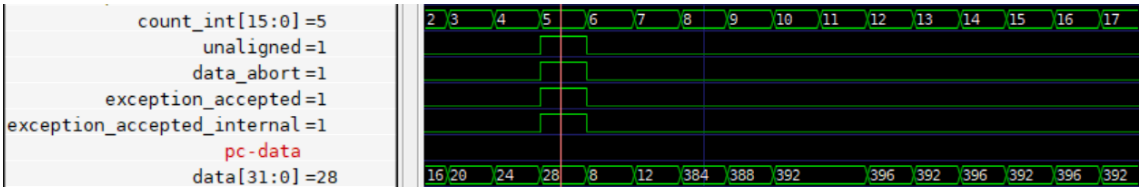
  Por último observamos que cada vez que la IRQ se acepta, el contador de las excepciones aumenta



- ***Data_Abort: en la memoria de instrucciones se incluyen dos bancos de pruebas en el que se realizan un acceso no alineado, y un acceso a una dirección fuera de rango. En ambos casos la ejecución salta a la rutina Data_Abort (bucle infinito).***

  Acceso no alineado:

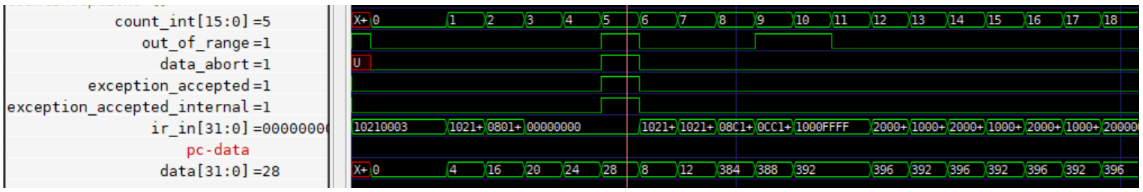
  Comprobamos que se realiza un data_abort por un acceso de memoria desalineado y salta a x"00000008". Después se hace un bucle infinito



  Dirección fuera de rango:

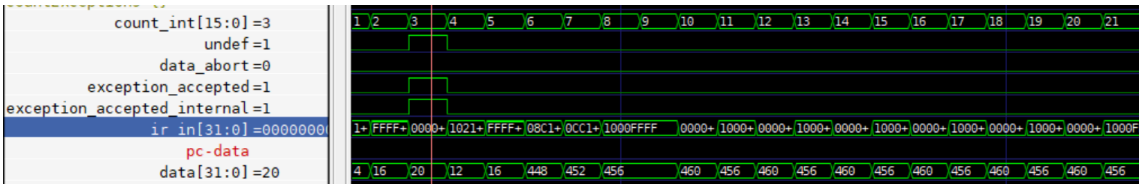
  Comprobamos que se realiza un data_abort por un acceso a memoria desalineado y salta a x"00000008". Después se hace un bucle infinito.

  El out_of_range se activa cuando una dirección se está fuera de rango, sin embargo, data_abort solo se activa si el out_of_range ocurre en un acceso a memoria (WE or RE), por ello hay un segundo impulso de out_of_range que no activa data_abort.



- ***Undef: en la memoria de instrucciones se incluye un banco de pruebas en el que se ejecuta una instrucción con un código de instrucción desconocido. La ejecución salta a la rutina UNDEF (bucle infinito).***

  Comprobamos que se realiza un undef y salta a x"0000000C" (12). Después se hace un bucle infinito)



○ RF7 retorno a modo usuario RTE

Descripción: Al terminar la excepción se vuelve a la ejecución original con la instrucción RTE

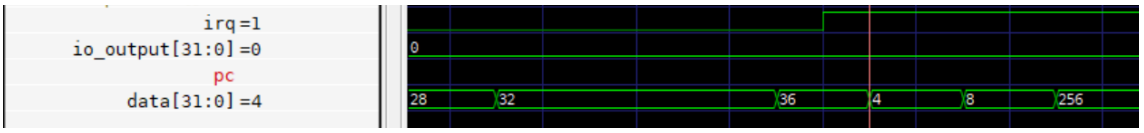
¿Cómo?

Al finalizar la excepción, se utiliza la instrucción RTE para retornar a la ejecución normal. Esta instrucción, realiza un salto a la instrucción anteriormente almacenada en *Exception_LR*. Además, se encarga de actualizar el status y activa de nuevo las interrupciones. El status pasa a ser '00'. Para el PC, se añade al mux *PC_in* una nueva entrada para coger el valor de *Exception_LR_output* cuando es una instrucción RTE_ID.

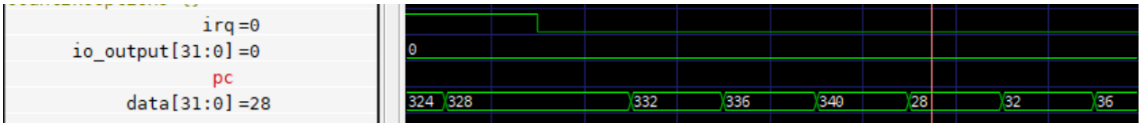
```
PC_in <= x"00000008"          when (Exception_accepted = '1') and (Data_abort = '1') else
         x"0000000C"          when (Exception_accepted = '1') and (UNDEF = '1') else
         x"00000004"          when (Exception_accepted = '1') and (IRQ = '1') else
         Exception_LR_output  when RTE_ID = '1' else
         BusA                 when RET_ID = '1' else
         Dirsalto_ID          when salto_tomado = '1' else
         PC4; -- PC+4
```

Verificación: Verificamos las IRQs. El código funciona aunque se interrumpa repetidas veces. Para el resto de excepciones se asume que es un fallo irresoluble y se debe resetear el sistema, pero la gestión es idéntica que para las IRQs. A continuación, probamos que cuando se crea una IRQ se salta y luego, una vez terminada, se vuelve correctamente.

Cuando ocurre una IRQ se salta a x"00000004":



Cuando acaba la IRQ se vuelve a la ejecución:



○ RF8 contadores

Descripción: los contadores nos dan información de la ejecución del código

¿Cómo?

A continuación, se van a explicar los seis contadores que podemos encontrar:

1. **inc_cycles <= '1';**

   Este es el contador de ciclos. Sirve para contar la cantidad de ciclos que se han ejecutado durante el programa. El valor de este contador siempre es '1' ya que el reloj nunca para.

2. **inc_I <= valid_I_WB;**

   Este es el contador de las instrucciones ejecutadas. Cuando una instrucción llega a la etapa WB (Write-back) ya ha pasado por todas las etapas anteriores, lo que significa que esa instrucción ya ha sido ejecutada. Por esta razón se utiliza '*valid_I_WB'.*

3. **inc_data_stalls <= stall_ID and not(stall_MIPS);**

   Este es el contador que se utiliza para la cantidad de detenciones por riesgo de datos, es decir, cuando se ha tenido que parar la ejecución debido a dependencias de datos que no se pueden solucionar mediante la anticipación de operandos. Además, añadimos el 'not(stall_MIPS)' porque puede ocurrir que stall_ID esté activo mientras stall_MIPS esté también activo y como los inc_Mem_stalls los contamos a parte, no añadimos este caso como riesgo de datos.

4. **inc_control_stalls <= kill_IF;**

   Este es el contador para la cantidad de detenciones por riesgos de control. La señal *'kill_if'* indica que el procesador ha matado la instrucción de la etapa fetch ya que está manejando una instrucción de salto y no sabe si la va a tomar o no hasta que la procese. Anticipa salto no tomado y por ello se carga la siguiente instrucción.

5. **inc_Exceptions <= Exception_accepted;**

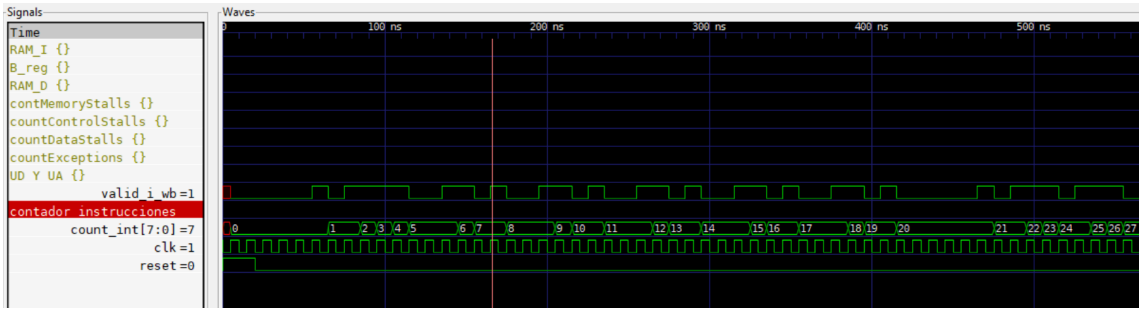
   Este es el contador para la cantidad de excepciones que han sido aceptadas. Para ello se mira la señal *'Exception_accepted'* que indica si se ha aceptado o no.

6. **inc_Mem_stalls <= stall_MIPS;**

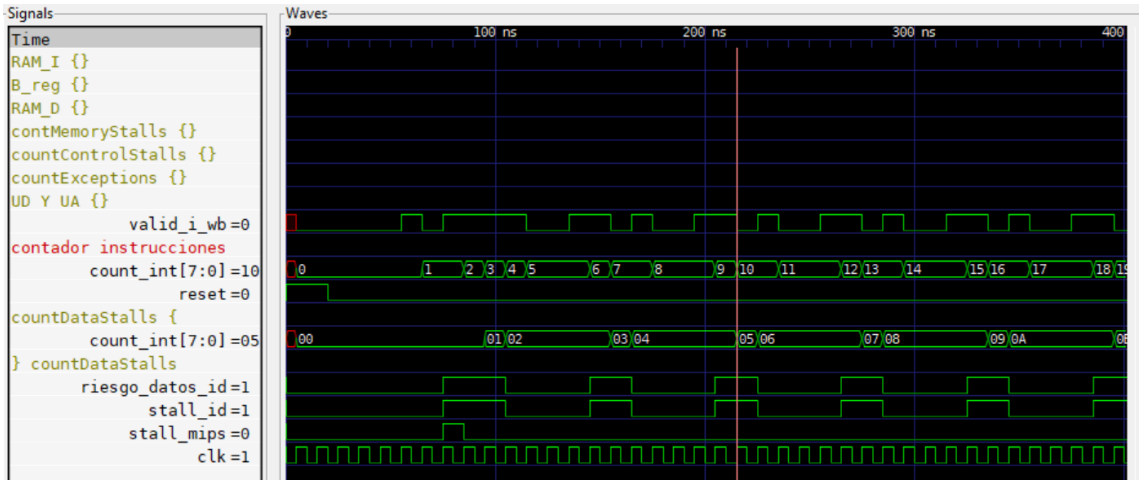
   Este es el contador para la cantidad de las detenciones por acceso a memoria cada vez que se pide una operación a memoria, y esta no está "ready".
   stall_MIPS_internal <= not(IO_MEM_ready)and valid_I_MEM;
   stall_MIPS <= stall_MIPS_internal;

Verificación: En el banco de pruebas test_IRQ se ha comprobado que:

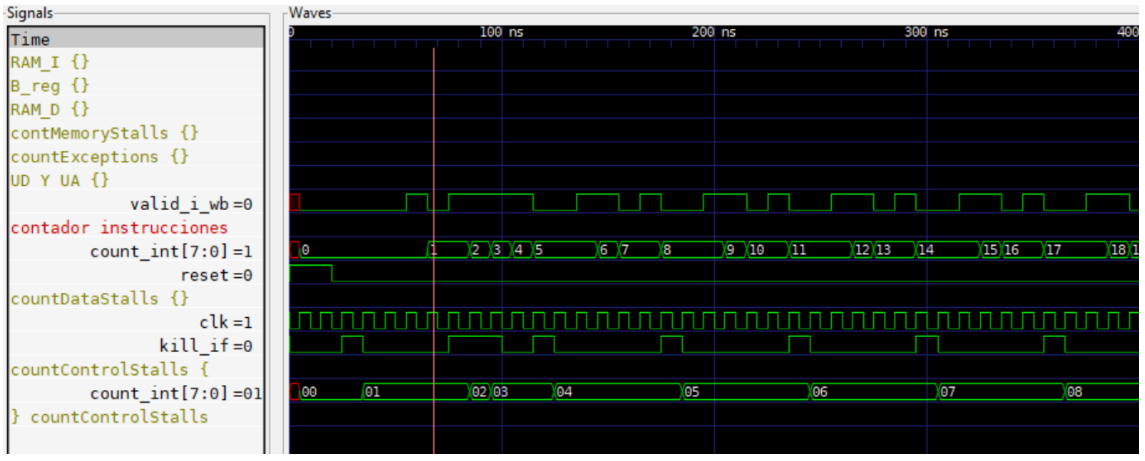
- Por cada instrucción válida que hace su etapa WB se incrementa el contador de instrucciones Ins. Si la instrucción no es válida no se incrementa.



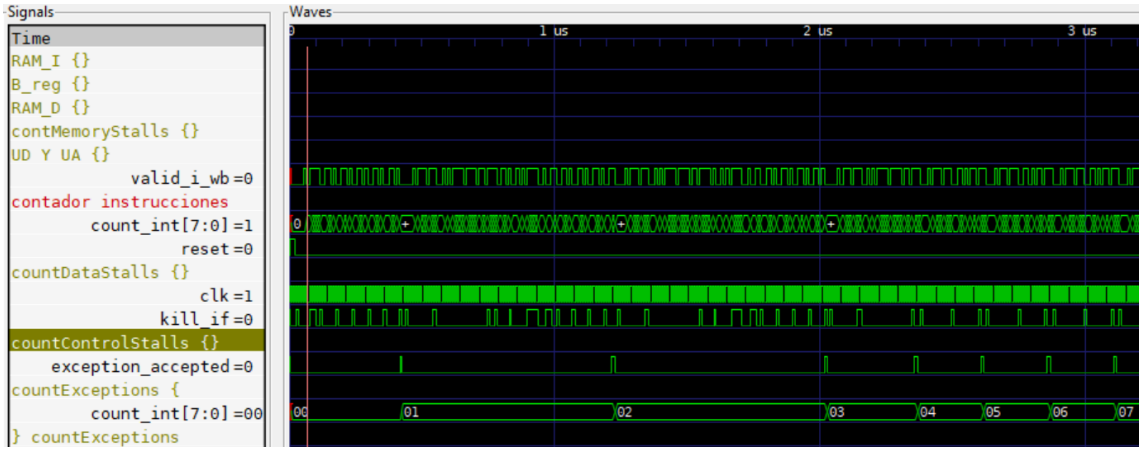
- Data_stalls contabilizan bien los ciclos debidos a los riesgos de datos incrementando el número indicado en los test mencionados en el RF5.



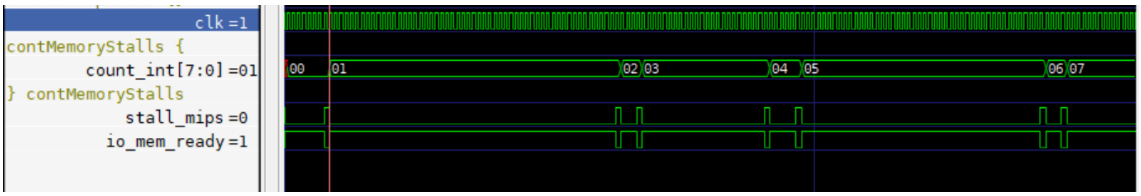
- Control_stalls contabilizan bien los ciclos debidos a los riesgos de control, incrementando un ciclo en cada salto tomado (ver pruebas en el RF6).



- Exception_accepted contabiliza las excepciones aceptadas. Su cuenta coincide con la cuenta que realiza el propio código.



- Memory_stalls: cada vez que se pide una operación a memoria, y esta no está “ready” se incrementa este contador.



- IMPORTANTE: las paradas deben incrementar sólo un contador. Si hay hay varios motivos, se cuenta el de mayor jerarquía : memory stall> data_stall> control_stall. Ya que el otro se mantendrá y se contabilizará cuando toque.

Cuantificación de horas dedicadas:

	Ana	Fernando
Estudio del MIPS, VHDL, entorno, instalación	2	2
Adición de las nuevas instrucciones	2	2
Gestión de excepciones	2	2
Depuración, verificación y programas de prueba	20	20
Memoria	6	6
TOTAL	32	32

El trabajo ha sido realizado por los 2 miembros del equipo a la vez, por lo tanto, hemos realizado las mismas horas de trabajo. Adaptarnos al entorno nos ha costado un poco, ya que en las prácticas anteriores no era necesario tener en cuenta tantas señales como durante este proyecto. Añadir las nuevas instrucciones ha sido rápido ya que partíamos del trabajo realizado durante la práctica 3. La depuración ha sido la parte más costosa, ya no solo por la verificación del correcto funcionamiento sino también por el planteamiento de nuestros propios test. La memoria ha sido un trabajo largo ya que hemos querido explicar correctamente todos los pasos.

Conclusiones y Autoevaluación:

	CONCLUSIONES	AUTOEVALUACIÓN:
Ana	Este proyecto me ha ayudado a asentar los conocimientos adquiridos en la asignatura. Creo que trabajar con una persona que no conocía también ha sido una buena experiencia para ver cómo trabajan otras personas. En general creo que hemos hecho un buen trabajo y hemos tenido una buena organización. Hemos hecho un buen equipo.	9
Fernando	Por mi parte, el proyecto me ha ayudado mucho a entender los conceptos de UA y UD, además de ayudarme a afianzar más la arquitectura global del MIPS.	9

Agradecimientos:

Un gran agradecimiento a Juan José Muñoz Lahoz (902677) y Marcos San Julián Fuertes (899849), por ayudarnos a encontrar problemas y apoyarnos durante la realización del proyecto.