Arquitectura y organización de computadores 2

Proyecto 1
Gestión de riesgos y excepciones en MIPS

Dorian Boleslaw Wozniak - 817570@unizar.es

Adrian Arribas Mateo - 795593@unizar.es

\

Lunes, 26 de marzo de 2023

Índice

1. Resumen

En este proyecto, se ha modificado un procesador MIPS de 32 bits segmentado en 5 etapas, para incluir nuevas funcionalidades.

Se ha implementado el tratamiento de excepciones para interrupciones externas, así como para accesos a memoria desalineados y lectura de instrucciones no existentes en la arquitectura del procesador. Además se han implementado dos nuevas instrucciones, RTE y WRO, que permiten regresar a la última instrucción anterior a la interrupción y escribir datos para el exterior del procesador, respectivamente.

También se han debido hacer los cambios necesarios para gestionar los riesgos de datos ya sea mediante eliminación de aquellos que sea posible utilizando la anticipación de operandos, así como deteniendo la ejecución de las etapas IF e ID para aquellos no eliminables como los generados por LW (1 ciclo al no poder anticipar en etapa EX), BEQ o WRO (hasta 2 ciclos, al procesarse ambos en la etapa ID, donde no hay anticipación).

La unidad de control además resuelve los riesgos de control considerando los saltos no tomados. En caso de realizar el salto, elimina la instrucción en etapa IF. RTE es un caso de salto incondicional.

Finalmente, se gestionará un conjunto de contadores que monitorizan el rendimiento y comportamiento del procesador a lo largo de la ejecución.

La implementación de las funcionalidades se ha verificado diseñando una serie de pruebas que cubran una serie de casos significativos, que junto con los tests de prueba dados junto al guión, corroboran el correcto funcionamiento de las nuevas adiciones.

También se incluyen las conclusiones individuales del proyecto de cada integrante del equipo así como un control de dedicación para cada uno.

2. Verificación de los requisitos funcionales

2.1. RF1 cambio a modo excepción

Descripción

El procesador admite una serie de excepciones: reset, interrupciones externas (IRQ), acceso a datos desalineados (Data_Abort), y lectura de una instrucción no definida en la arquitectura del procesador (UNDEF). Cada una dispone de una entrada en un vector de excepciones en las primeras palabras de la memoria de instrucciones. Al producirse una de las excepciones, se salta automáticamente a su dirección predeterminada, donde se detalla la acción a tomar, generalmente un salto al código de tratamiento de dicha excepción.

Para ello, se requiere disponer de dos modos de ejecución: usuario y excepción. Al producirse una excepción, se modifica el registro de estado, que indicará el modo de ejecución actual así como si se admiten nuevas excepciones. El registro de estado tiene dos bits para indicar cada una de estas propiedades, pero en este proyecto tendrán el mismo valor ambas.

```
-- Selector de estado
-- Desactiva IRQ si se procesa excepción
status_input <=
    "11" WHEN (Exception_accepted = '1') ELSE
    "00";
```

Una excepción se considera aceptada siempre que no se inhiban las instrucciones y alguna de las excepciones esté activa:

```
Exception_accepted <= (NOT MIPS_status(1)) AND (IRQ OR Data_Abort OR Undef);
```

Así mismo, si se acepta una instrucción, se debe modificar el registro de estado. También se debe actualizar cuando se regresa de una excepción con RTE, que se explicará más adelante:

```
update_status <= Exception_accepted OR (RTE_ID AND Valid_I_ID);
```

Por otro lado, es necesario saltar a la posición adecuada del vector de excepciones si se recibe una excepción.

```
-- Dirección a cargar en PC
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
    ...
    PC4; -- Avanzar a siguiente instrucción
```

Se quiere atender las excepciones lo más rápido posible. En el caso de una detención, al no ejecutarse la instrucción con un riesgo de datos parada en ID, no es necesario esperar y se puede sobreescribir la instrucción en IF con la de la excepción:

```
-- Carga en PC salvo si hay parada y no hay excepción que vaya a sustituir
-- a la instrucción que va a esperar
load_PC <= NOT (Parar_ID AND NOT Exception_accepted);
```

Adicionalmente, debe almacenar la dirección de retorno para cuando acabe el tratamiento de la excepción con la última instrucción válida en la *pipeline*:

```
-- Selector instrucción de retorno de excepción
Return_I <=
    PC_exception_EX WHEN (valid_I_EX = '1') ELSE
    PC_exception_ID WHEN (valid_I_ID = '1') ELSE  -- Si no hay instr. en EX
    PC_out;                                       -- Si no hay instr. en ID
```

Las instrucciones en las etapas MEM y WB se ejecuratán de forma normál. La última instrucción válida dependerá de la situación:

- Normalmente, la última instrucción válida se encontrará en EX.
- En caso de detención por riesgo de datos, si se envía una señal de parada, la etapa EX del siguiente ciclo no contendrá datos válidos. Por ello, se debe tomar la instrucción en ID.
- En el caso de producirse un salto, la instrucción que estaba en la etapa IF no debe considerarse cuando pase a ID, así como la instrucción de salto que estaba en ID, cuando pase a la etapa EX, se debe desestimar al no realizar ninguna acción. Por tanto, la única instrucción válida en la *pipeline* es la primera instrucción tras el salto que es leída en IF.

El procesador contiene la lógica para determinar los casos anteriores:

```
-- Señal de anulación de instrucción en IF
valid_I_IF <= NOT(kill_IF);

-- Si no hay parada o salto, marca siguiente EX como válido
valid_I_EX_in <= valid_I_ID AND NOT (parar_ID OR salto_tomado);
```

Verificación

La verificación de que se producen los saltos correctos se ha realizado con una serie de tests, tanto con los proporcionados como con los de diseño propio.

En primer lugar se comentarán los saltos a excepciones Data_Abort (2 pruebas) y UNDEF (1 prueba).

Las dos pruebas de Data_Abort prueban accesos a memoria con un desplazamiento no múltiplo de 4 (siendo que el tamaño de las direcciones de memoria y de la palabra son de 32 bits). Ambas son similares: se prueban los casos `LW R1, 3(R0)` y `LW R1, 32767(R0)`.

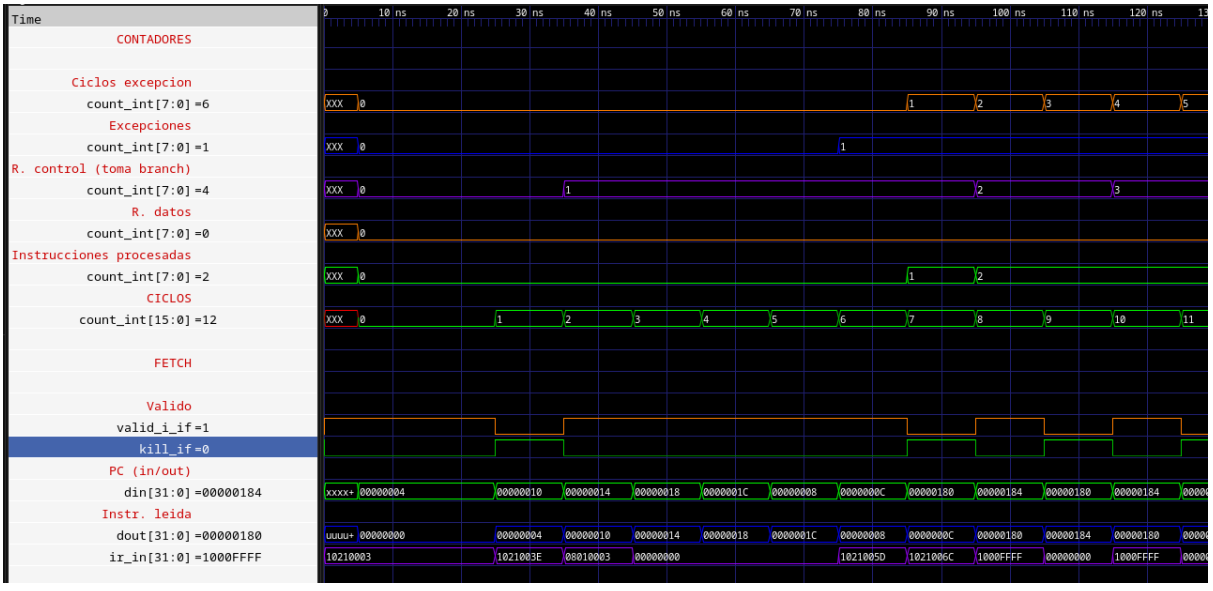

Figura 1: Fragmento del cronograma de ejecución del banco de pruebas de Data_Abort.

Como se puede apreciar en la Figura 1, al pasar la instrucción 0x08010003 (`LW R1, 3(R0)`) a la fase MEM, se genera una excepción. La siguiente instrucción cargada en IF es la 0x00000008, la posición de Data_Abort en el vector de excepciones. Las siguientes instrucciones son un salto incondicional a la posición 0x180 en memoria, que contiene un bucle infinito. El funcionamiento en el segundo banco para Data_Abort es equivalente.

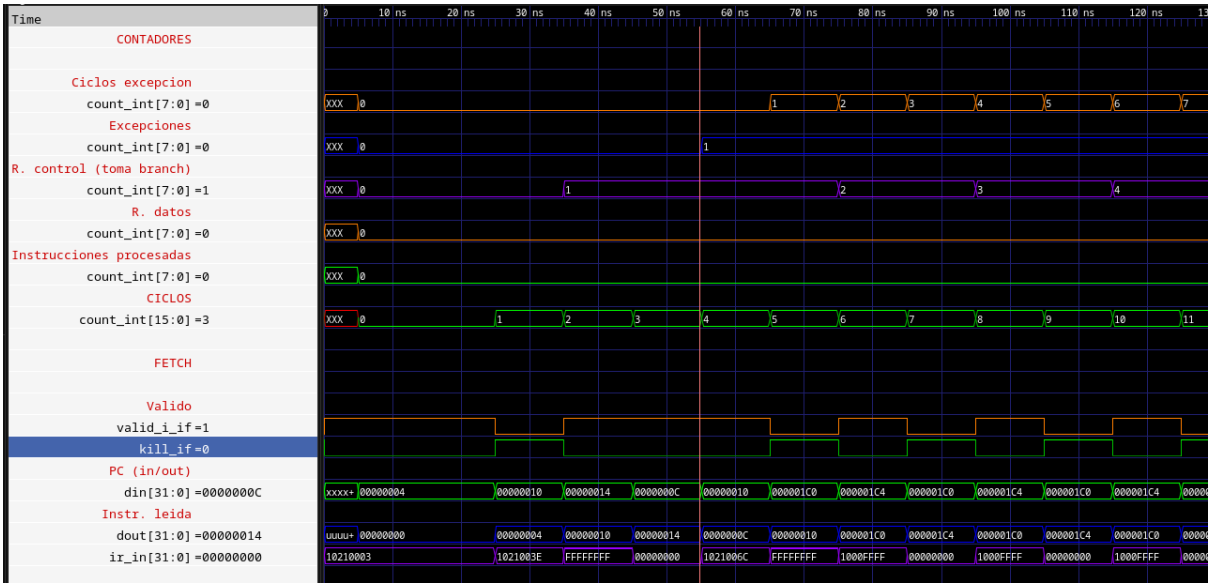

Figura 2: Fragmento del cronograma de ejecución del banco de UNDEF.

Para el banco de pruebas UNDEF, se intenta procesar la instrucción 0xFFFFFFFF. No hay una instrucción con código de operación 111111, por lo que en la fase de ejecución se produce una excepción UNDEF en la fase ID. El funcionamiento es similar, y se puede apreciar en la figura 5: se salta a la posición 0xC de memoria, donde se encuentra la entrada del vector de excepciones para UNDEF, y a continuación a 0x1C0, que contiene otro bucle infinito.

En cuanto a las interrupciones, se dispone de dos tests para probarlos. Uno de los tests es el proporcionado. Dicho test, en su bucle principal, muestra por el registro de salida la dirección de inicio de la pila, y a continuación calcula potencias de dos en un registro. Cada vez que se produce una interrupción, se muestra por el registro de salida el valor de una variable, y posteriormente se incrementa.

Un detalle a destacar es un problema no relacionado con el funcionamiento del hardware, sino con el banco de pruebas proporcionado. Si se interrumpe lo más pronto posible tras el reset, no se habrá cargado la dirección de inicio de la pila, y al apilar los registros utilizados,

se sobreescribirá el inicio de la memoria de datos, y el programa acabará no funcionando adecuadamente.

Adicionalmente, se ha diseñado un test propio más complejo basado en el código del banco de pruebas IRQ proporcionado. Este test, por un lado, calcula los 15 primeros números de la sucesión de Fibonacci, acabando por el 610 y mostrando cada número por el registro de salida. Por otro lado, las interrupciones calculan y escriben por el registro de estado la secuencia 0x49 0x52 0x51 0x21, que en ASCII se codifica como “IRQ!”. El test queda descrito con más detalle en el Anexo.

También se realizan una serie de interrupciones en una serie de puntos específicos del programa para probar el tratamiento de la instrucción de retorno. En este caso:

- En la primera iteración, se produce un riesgo de datos entre `lw r5, 4(r0)` y `beq r5, r6, end`. La IRQ se produce en el ciclo en el que se detendría la instrucción BEQ. En vez de producir la detención, se produce la interrupción almacenando LW como la instrucción de retorno al ser válido el EX de la instrucción anterior. El comportamiento se puede observar en la figura 3.
- En la segunda iteración, se interrumpe en el segundo ciclo de una parada de `wro r2` generada por `lw r2, 18(r0)`. En este caso, ya se ha producido la primera parada y la instrucción EX no es válida. Por tanto, se debe saltar al WRO que se encuentra en ID al regresar. Se puede apreciar en la figura 4.
- Entre la quinta y sexta iteración, se interrumpe cuando se lee la instrucción BEQ tras el salto incondicional al inicio del bucle. Al recomenzar la ejecución después del riesgo de control generado por el salto incondicional, la única instrucción válida es IF (pues ya se ha procesado el salto y no hace nada después de ID), por lo que se almacena dicha instrucción. Al producirse una interrupción justo después de salir de otra, se produce el mismo caso. Se puede apreciar en las figuras 5 y 6.
- También hay otros casos similares a los descritos anteriormente con otras combinaciones de instrucciones.

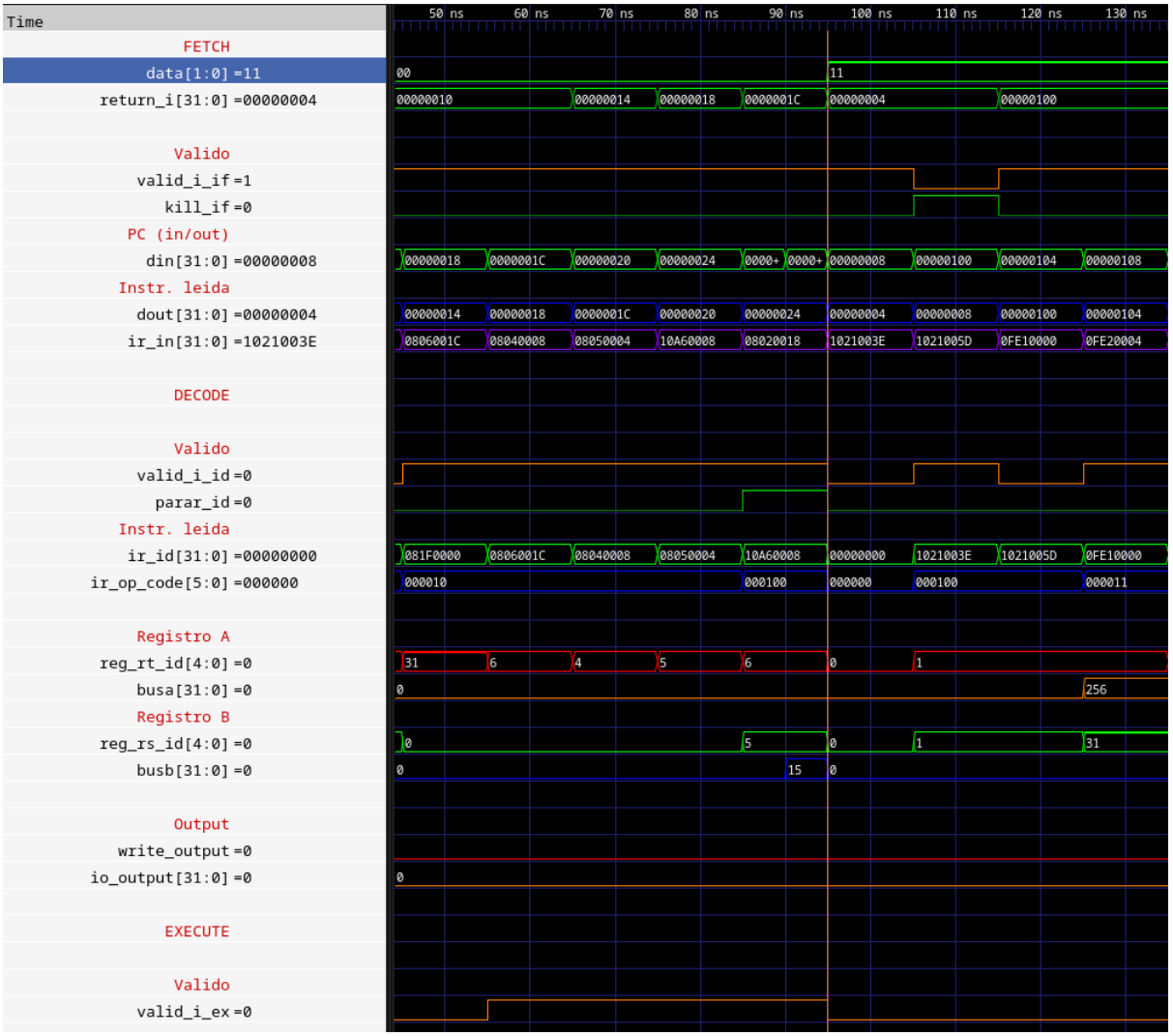

Figura 3: Fragmento del cronograma del test 4. Nótese que la dirección almacenada antes de la interrupción (línea roja vertical) es 0x1C, correspondiente a `lw r5, 4(r0)`. Tras la línea roja, se anulan las instrucciones en ID y EX, y se carga la dirección 0x04, es decir, el vector de excepciones para IRQ.

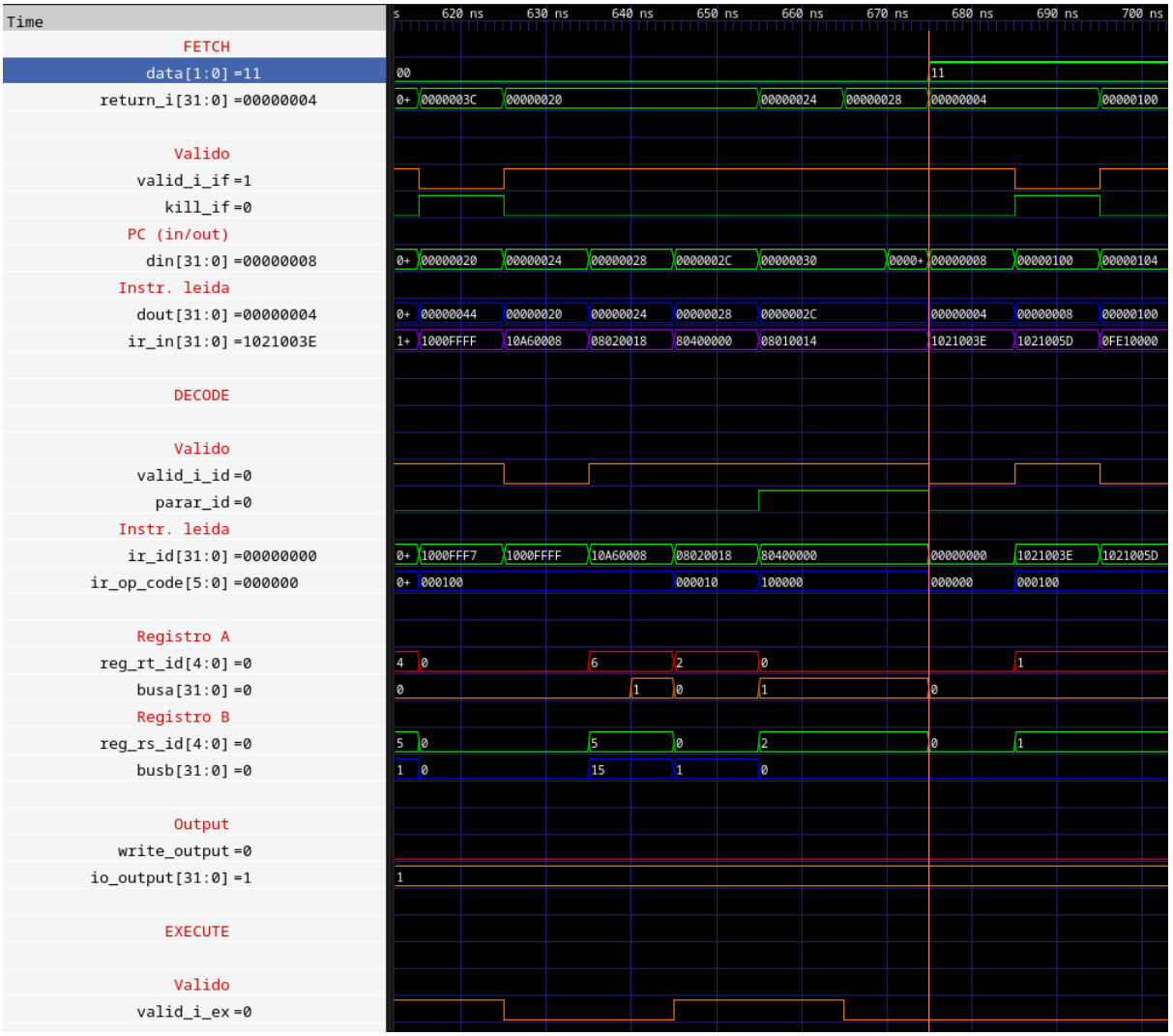

Figura 4: Fragmento del cronograma del test 4. La dirección de retorno almacenada pertenece al `wro r2`, que ha sido detenido ya un ciclo. Se puede observar que la instrucción EX no es válida. Data es el registro de estado: cuando se produce la excepción, pasa a 11.

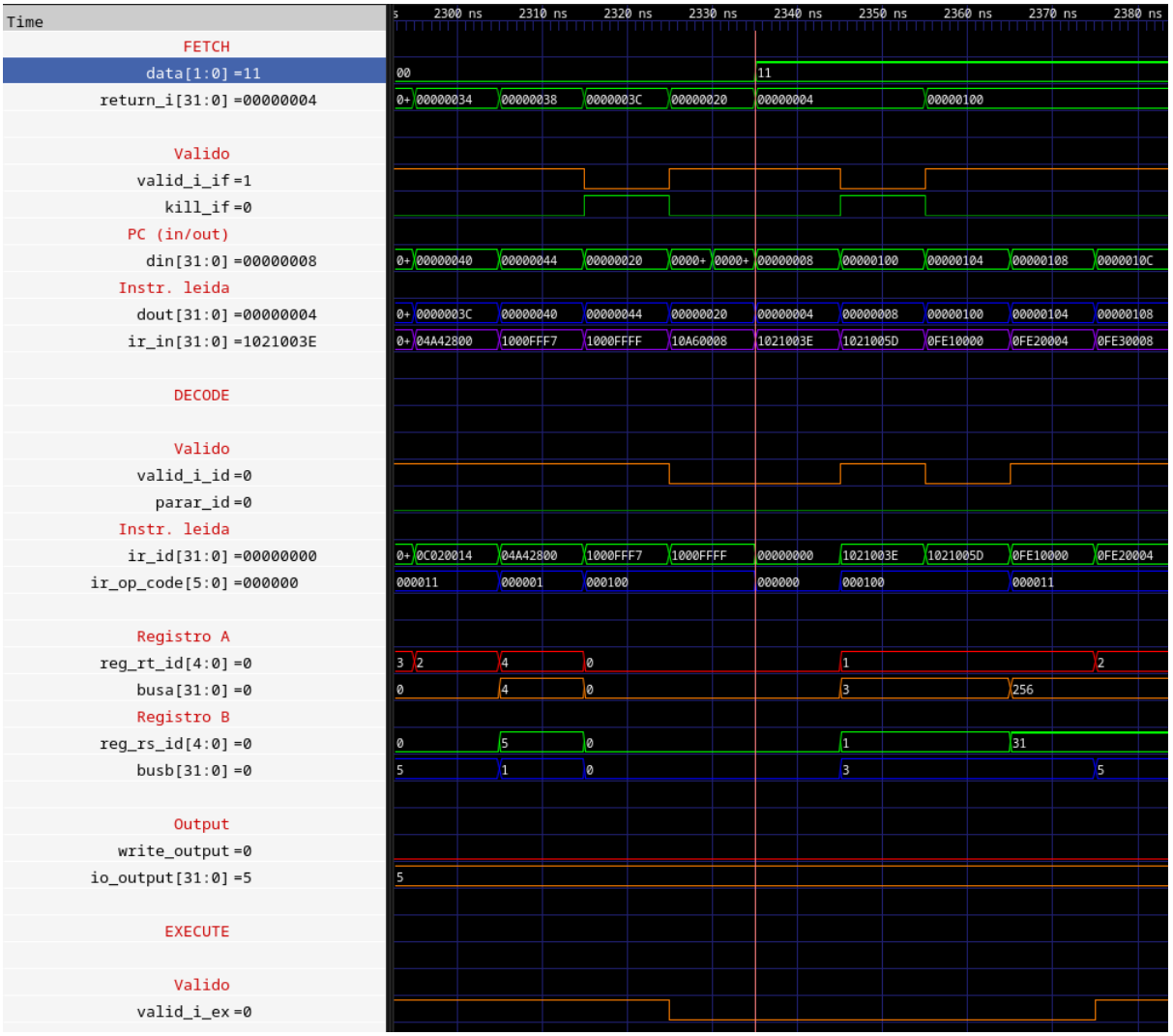

Figura 5: Fragmento del cronograma del test 4. La dirección de retorno almacenada pertenece al `beq r5, r6, end`, cargado en IF tras un salto. Las instrucciones ID y EX no son válidas antes de la interrupción.

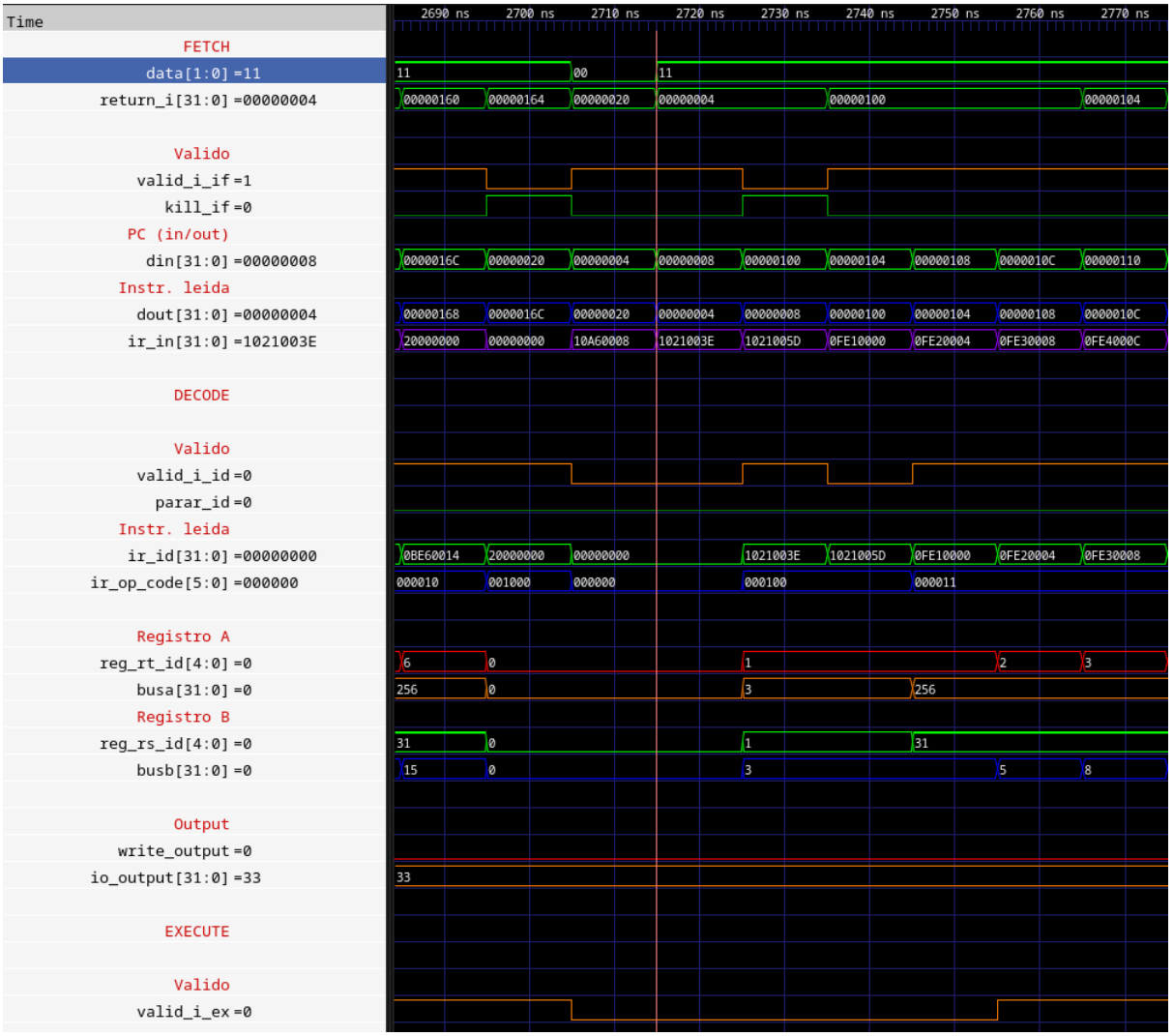

Figura 6: Fragmento del cronograma del test 4. En este caso, se produce una interrupción inmediatamente posterior a un RTE (se explica más adelante.). Al ser la última instrucción antes de la primera RTE el BEQ descrito anteriormente, se vuelve a escoger IF como instrucción de regreso al ser la única válida.

2.2. RF2 retorno a modo usuario RTE

Descripción

Junto con las interrupciones, es necesario una operación para regresar al modo usuario y retomar el procesamiento de las instrucciones de la sección principal del código. Como se ha descrito anteriormente, al ocurrir una excepción, se almacena la dirección de retorno, es decir, la úlitma instrucción válida, en el registro LR.

La instrucción RTE es un salto incondicional que carga la dirección de LR en el PC.

```
-- Dirección a cargar en PC
PC_in <=
    ...
    Exception_LR_output WHEN salto_tomado = '1' AND RTE_ID = '1' ELSE -- RTE
    Dirsalto_ID WHEN salto_tomado = '1' AND RTE_ID = '0' ELSE -- BEQ con salto
    PC4; -- Avanzar a siguiente instrucción
```

Hay que destacar que las excepciones Data_Abort y UNDEF no se consideran recuperables, y por tanto no se realizan pruebas. Por otro lado, este diseño no trata el caso de utilizar RTE en modo usuario, y puede provocar un comportamiento indefinido. Se puede incorporar una protección contra este tipo de casos añadiendo una condición extra al intentar cargar la dirección de LR tal que si el registro de estado está en modo usuario, no haga ningún salto e ignore la instrucción.

Verificación

Se utilizan las pruebas anteriores para determinar que se produce correctamente un regreso adecuado del modo usuario para cada caso. Se incluyen los cronogramas de cada caso descrito en el punto 2.1 en las figuras 7 y 8. La figura 6 ya muestra el regreso de modo excepción para el caso donde se almacena una instrucción en IF.

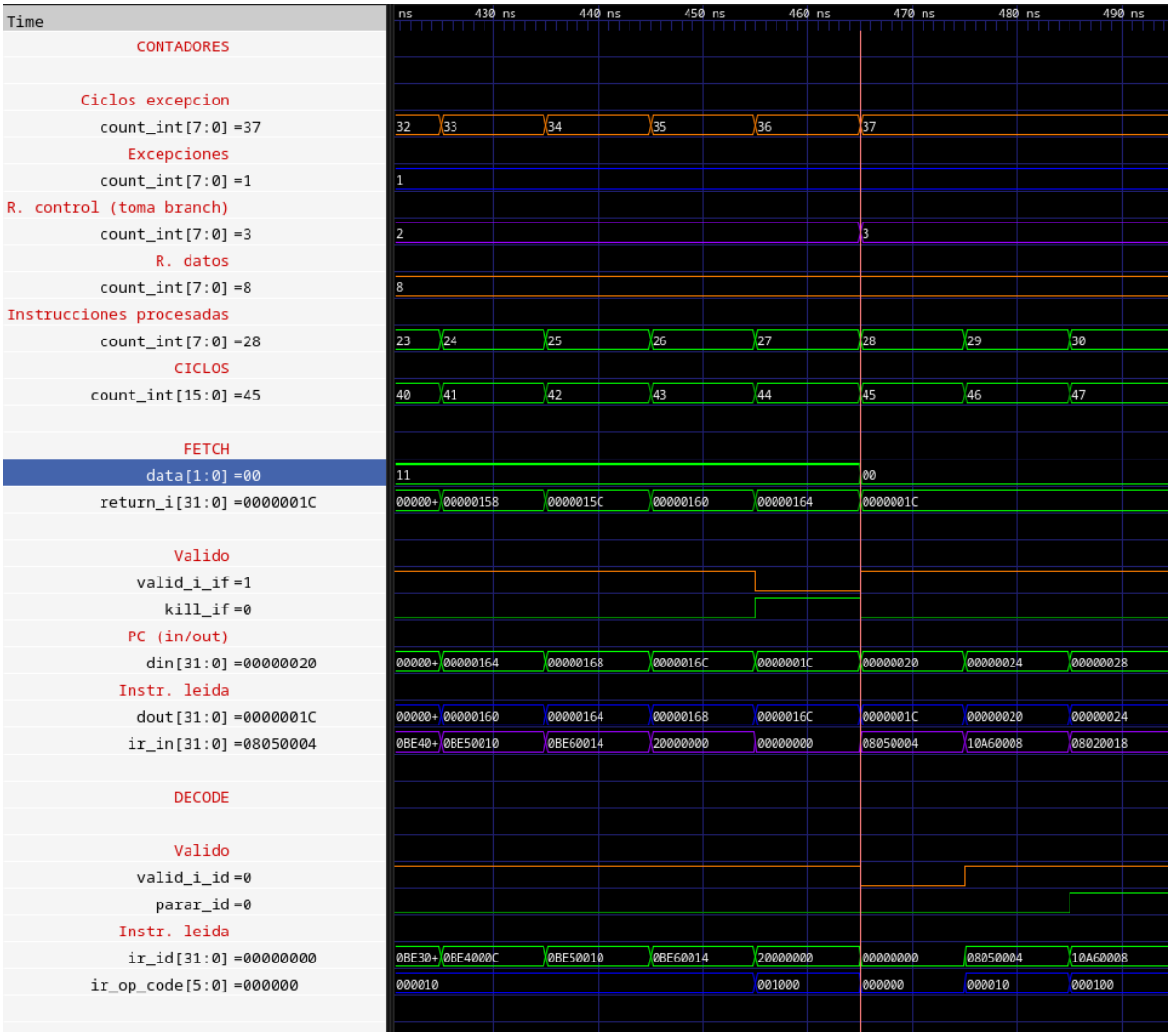

Figura 7: Fragmento del cronograma del test 4. Se puede apreciar que se vuelve a la instrucción LW almacenada en LR antes del salto por excepción. Cabe destacar como cambia el registro de estado de 11 a 00.

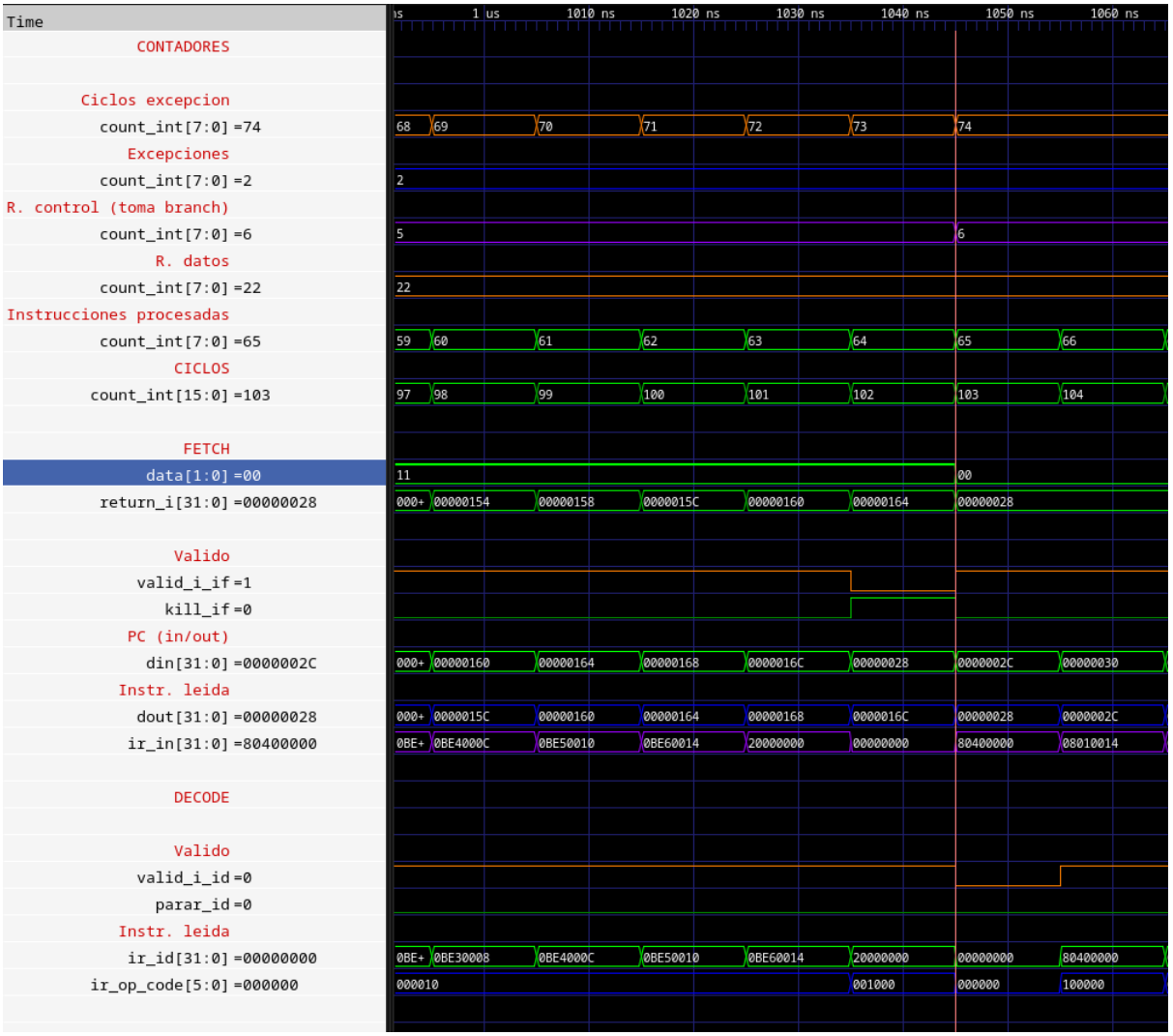

Figura 8 Fragmento del cronograma del test 4. Se regresa a la instrucción WRO almacenada durante el salto por la interrupción.

2.3. RF3 WRO

Descripción

WRO es una nueva instrucción añadida al repertorio que permite escribir un dato en el registro de salida del MIPS para comunicarse con el exterior.

El dato a cargar se toma del bus A en la etapa ID, por lo que no se puede utilizar la anticipación de operandos. Por tanto, hay que asegurarse de que no se escriben datos erróneos cuando la *pipeline* está parada y, por tanto, no contiene los datos necesarios.

```
Write_output <= NOT Parar_ID AND write_output_UC AND valid_I_ID;
```

Verificación

El test 4 utiliza de forma extensa las instrucciones WRO para mostrar datos en pantalla. Se puede observar en el cronograma como va cambiando el valor de la salida, y cuando está detenido no escribe.

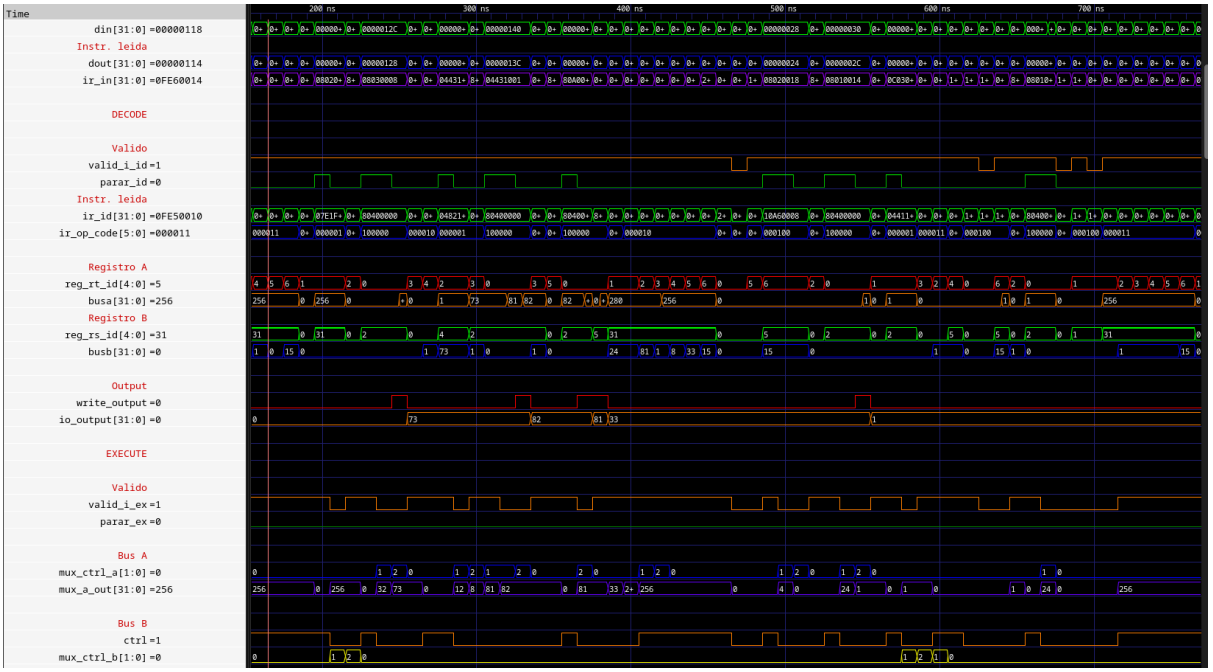

Figura 9: Fragmento del cronograma del test 4. Se puede observar como se escribe una serie de valores en WRO que, en codificación ASCII, forman la palabra "IRQ!".

2.4. RF4 Anticipación de operandos

Descripción

El uso de anticipación de operandos de las etapas MEM y WB a la etapa EX pueden prevenir la mayoría de dependencias *read-after-write*, donde un productor genera operandos requeridos por un consumidor.

Para detectar si es necesario activar un cortocircuito, se utilizan las señales RW y RegWrite de la etapa potencialmente anticipable. Si RegWrite, que determina si se va a escribir un dato en el banco de registros, está activo y alguno de los dos operandos es el mismo que el operando de escritura, se activa el cortocircuito.

```
Corto_A_Mem <=
    '1' WHEN ((Reg_Rs_EX = RW_MEM) AND (RegWrite_MEM = '1')
         AND (valid_I_MEM = '1')) ELSE
    '0';

Corto_A_WB <=
    '1' WHEN ((Reg_Rs_EX = RW_WB) AND (RegWrite_WB = '1')
         AND (valid_I_WB = '1')) ELSE
    '0';

-- Ibidem para cortos del bus B
```

En caso de poder anticipar el mismo dato desde MEM o desde WB, se elige el más nuevo (MEM).

```
   MUX_ctrl_A <=
       "10" WHEN (Corto_A_Mem = '0' AND Corto_A_WB = '1') ELSE
       "01" WHEN (Corto_A_Mem = '1') ELSE
       "00";
-- Ibidem para cortos del bus B
```

Se puede anticipar si el productor de datos es una instrucción aritmética o una carga de datos en etapa de WB, y el consumidor no requiere . Los otros casos se describen en el siguiente apartado.

Verificación

Además de los casos de anticipación del código del test de IRQ dado, se ha creado un test adicional que prueba combinaciones de anticipaciones entre los operandos rs y rt de las instrucciones arit, lw y sw. El programa se divide en dos partes: anticipaciones donde el productor es una instrucción aritmética y donde el productor es una carga de datos.

El objetivo del programa es obtener un resultado en el banco de registros y en la memoria de datos que no sería correcto si no se anticipan los operandos. En la figura 9, se puede observar en la zona inferior las anticipaciones que toma la unidad de anticipación.

El código de la prueba comentado con las anticipaciones que se realizan en cada etapa se encuentra en el Anexo.

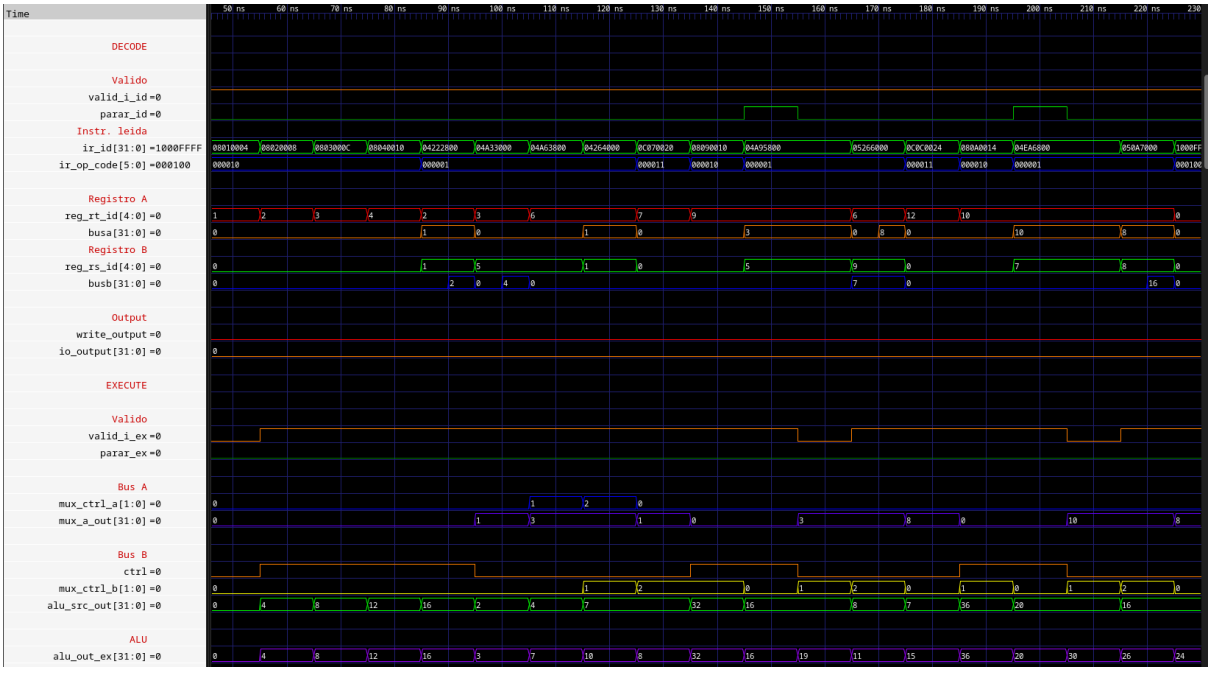

Figura 10: Fragmento del cronograma del test 2.

2.5. RF5 Riesgos de datos

Descripción

Hay algunas dependencias *read-after-write* que no se pueden anticipar. Estas son lecturas inmediatamente después de un LW, así como las instrucciones BEQ y WRO cuando consumen datos.

En primer lugar, la unidad de control detecta posibles dependencias de datos. Para ello, compara las fases ID, EX y MEM. Si alguna de las etapas escribe en el banco de registros un dato que requiere la instrucción en ID, se marca como dependencia. Hay que tener en cuenta que NOP y RTE no utilizan operandos y por tanto nunca generan dependencias, y WRO solo genera dependencias para rs.

```
dep_rs_EX <=
   '1' WHEN ((valid_I_EX = '1') AND (Reg_Rs_ID = RW_EX)
        AND (RegWrite_EX = '1') AND (IR_op_code /= NOP)
        AND (IR_op_code /= RTE_opcode)) ELSE
   '0';

dep_rs_Mem <=
   '1' WHEN((valid_I_MEM = '1') AND (Reg_Rs_ID = RW_MEM)
        AND (RegWrite_Mem = '1') AND (IR_op_code /= NOP)
       AND (IR_op_code /= RTE_opcode)) ELSE
   '0';
-- Ibidém para rt, pero además tiene la clausula AND (IR_op_code /=
WRO_opcode)
```

Una vez detectados, se subdividen las dependencias según si son generadas por LW, BEQ o WRO. Si alguno de estos se detecta, se produce un riesgo de datos.

```
ld_uso_rs <=
   '1' WHEN dep_rs_EX = '1' AND MemRead_EX = '1' ELSE
   '0';

BEQ_rs <=
   '1' WHEN (dep_rs_EX = '1' OR dep_rs_Mem = '1') AND IR_op_code = BEQ ELSE
   '0';

WRO_rs <=
    '1' WHEN (dep_rs_EX = '1' OR dep_rs_Mem = '1')
```

```
        AND IR_op_code = WRO_opcode ELSE
    '0';

riesgo_datos_ID <= BEQ_rt OR BEQ_rs OR ld_uso_rs OR ld_uso_rt OR WRO_rs;

- Ibidém para rt
```

Verificación

Para verificar la corrección de las detenciones por riesgos de datos, por un lado se utilizan los casos de la prueba 2 donde se generan paradas por requerir los ADD el dato de LW. EN la figura 10 se puede ver como se generan paradas cuando los consumidores se encuentran a 1 ciclo del LW, pero si se puede anticipar si estan a distancia 2.

Además de las pruebas del test de IRQ otorgado, se ha creado un programa de prueba. el test 3, que prueba una serie de situaciones donde BEQ y WRO requieren datos que no tienen aún. El código y las paradas que se realizan se pueden ver en el anexo.

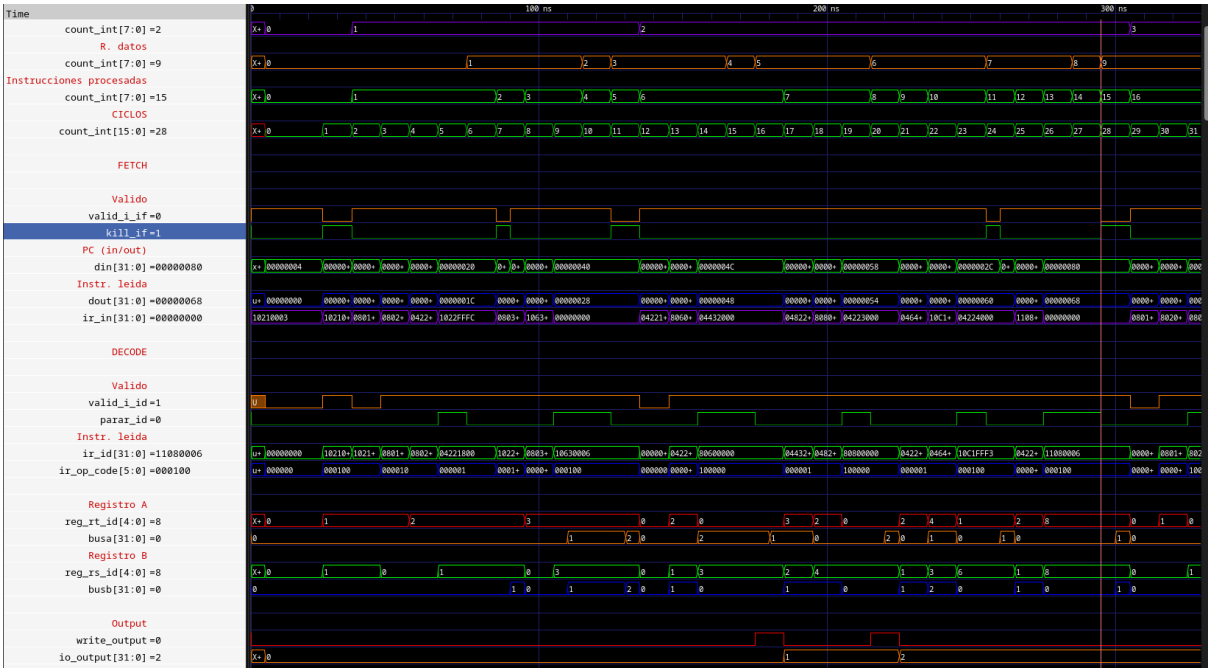

Figura 11: Fragmento del cronograma del test 3. Se puede apreciar las paradas en las instrucciones del tipo 000100 (BEQ) y 100000 (WRO) generadas.

2.6. RF6 Riesgos de control

Descripción

Los riesgos de control son generados por saltos tomados. Por defecto, el procesador asume que el salto no se ha tomado. Sin embargo, si se toma el salto es necesario eliminar la instrucción actual en IF, pues no se debe ejecutar, y sustituir por la instrucción leída tras el salto.

```
Kill_IF <= NOT (Parar_ID_internal) AND valid_I_ID AND salto_tomado;
```

Verificación

Casi todas las pruebas requieren el funcionamiento correcto de los riesgos de control. A continuación, se muestran dos ejemplos de un salto no tomado y de un salto tomado del test 4.

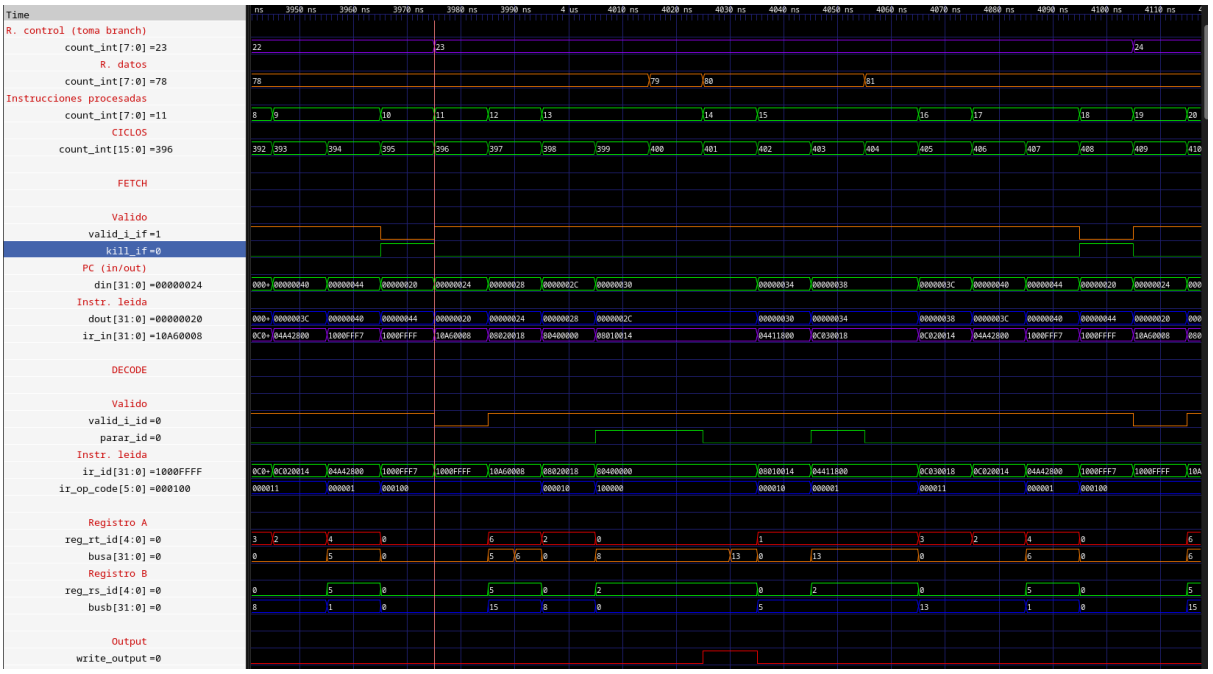

Figura 12: Fragmento del cronograma del test 4. Se puede observar como se genera un riesgo de control cuando se produce un salto por la instrucción 0x1000FFF7, que salta al inicio del bucle, y se elimina la instrucción 0x1000FFFF, que es el bucle incondicional que se ejecuta cuando acaba el programa. La instrucción 0x10A60008, al no cumplirse la igualdad, no genera riesgo de datos.

2.7. RF7 Contadores

Descripción

Los contadores son útiles para medir el rendimiento de los programas, así como el comportamiento y eventos que ocurren en la *pipeline*.

El proyecto cuenta con una serie de contadores:

- El contador de ciclos siempre incrementa por cada flanco.
- El contador de instrucciones cuenta el número de instrucciones que llegan a WB (todas tienen que pasar por ahí) y son válidas, o si se toma un salto (tal y como está diseñado, los saltos se consideran instrucciones inválidas a partir de EX).
- Los contadores de paradas de datos y control cuentan cuando se producen paradas de la fase ID y cuando se elimina la fase IF. Si se producen varias detenciones seguidas, se cuentan cada una.
- El contador de excepciones aumenta cuando se acepta una excepción y pasa a modo de excepción. Se cuentan los ciclos que pasan hasta que el registro de estado vuelve a modo usuario.

```
inc_cycles <= '1'; -- Incrementa siempre

inc_I <= '1' WHEN valid_I_WB = '1' OR salto_tomado = '1' ELSE
   '0';
inc_data_stalls <= '1' WHEN Parar_ID = '1' AND parar_EX = '0' ELSE
   '0';
inc_control_stalls <= '1' WHEN Kill_IF = '1' AND Parar_ID = '0' ELSE
   '0';
inc_Exceptions <= '1' WHEN Exception_accepted = '1' ELSE
   '0';
inc_Exception_cycles <= '1' WHEN MIPS_status(0) = '1' ELSE
   '0';
```

Verificación

Se ha utilizado el test 4 para verificar que los contadores se actualizan adecuadamente. A continuación, se presentan algunos ejemplos.

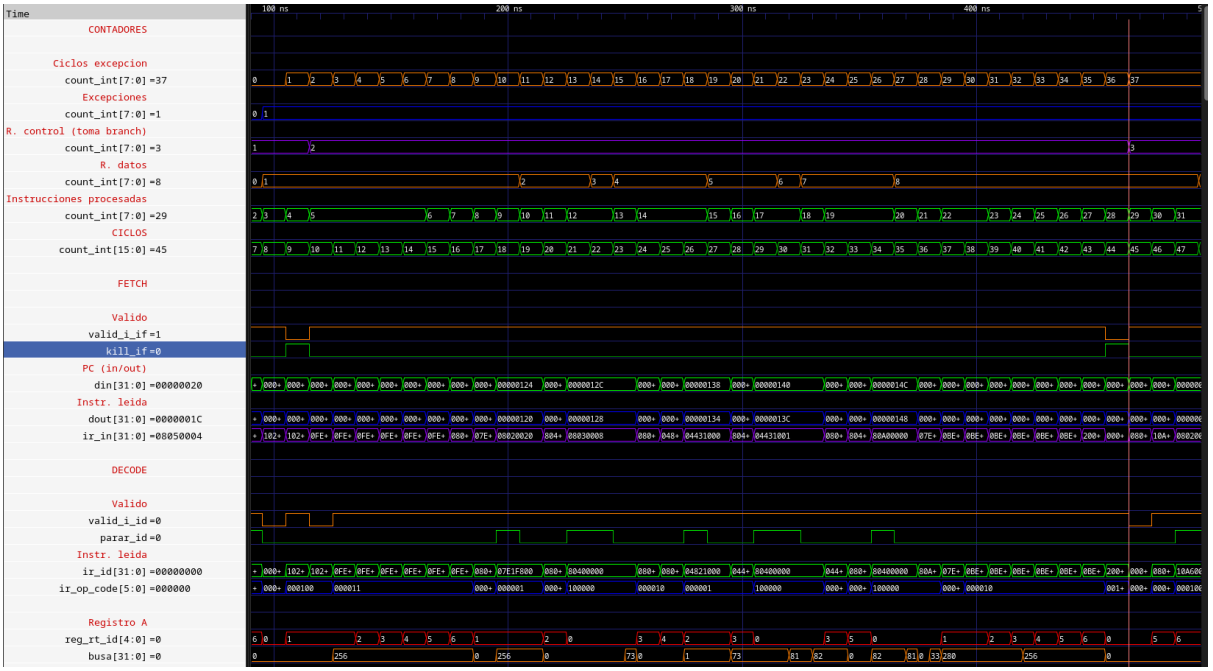

Figura 13: Fragmento del cronograma del test 4. Se puede observar la ejecución de una IRQ. Los contadores se encuentran en la parte superior.

Se puede apreciar como se cuenta una excepción al inicio. Se producen dos riesgos de control: en el salto del vector de excepciones a la rutina de tratamiento y en el RTE. Al comenzar a ejecutar la rutina no se contabilizan instrucciones completas hasta 5 ciclos después, cuando llegan las primeras instrucciones al WB. También no completa instrucciones cada vez que hay dependencias de datos.

3. Conclusiones y Autoevaluación

Consideramos que el trabajo realizado ha sido interesante y muy útil, tanto para la asignatura como para futuras modificaciones que debamos implementar en el procesador, la correcta realización del proyecto ha conllevado a conocer, entender y profundizar acerca del funcionamiento del MIPS, teniendo que comprender las relaciones con cada uno de sus componentes y la función que tienen dentro de la ruta de datos y la unidad de control, además de la importancia que tienen la unidad de detención y de anticipación dentro del programa para gestionar las dependencias entre registros, la limpieza de instrucciones, entre otras cosas.

Autoevaluación, Adrián Arribas: Considero que he necesitado mucho más tiempo en comparación a mi compañero para comprender las tareas a realizar para el proyecto debido a la gran dificultad que a mi me ha conllevado, es decir, he utilizado más tiempo en entender el funcionamiento del nuevo programa y la finalidad de los cambios necesarios que en realizar el código para completar las tareas, conllevando a que mi compañero haya tenido que avanzar y dedicar más tiempo para completar el proyecto. Sin embargo, opino que nos hemos gestionado bien para realizar el seguimiento del trabajo, hemos concertado las reuniones necesarias y considero que no hemos perdido el tiempo realizando trabajo inutil. En base a esto mi conclusión es que podría haber servido de mayor ayuda para mi compañero si yo comprendiera mejor los contenidos de la asignatura y si el resto de trabajos de otras asignaturas me hubieran permitido dedicarle más tiempo, por lo tanto mi nota es de un 7 sobre 10.

Autoevaluación, Dorian Wozniak: En general, el proyecto ha sido una experiencia positiva. En mi caso particular, esta es mi segunda vez cursando la asignatura, y, junto con el conocimiento del año pasado, me ha facilitado mucho el desarrollo del proyecto (y el progreso de la asignatura en general).

Durante el proyecto, hemos tenido algunos contratiempos: el primero y mas importante, mi compañero Adrian ha tenido problemas instalando las herramientas necesarias (en concreto, GHDL) debido a su equipo personal, por lo que buena parte de la depuración de los programas ha recaído en mi. Esto se ha notado en el tiempo que se ha tardado en detectar dos errores graves: una errata en la unidad de detención al copiar y pegar sentencias para detectar dependencias, y no considerar marcar los BEQ y RTE como inválidos al saltar, lo cual provocaba que al hacer varios IRQ seguidos, al terminarse, acababan en un bucle con la propia instrucción RTE (al tomarla como punto de partida para la ejecución).

Sin embargo, hemos sido capaces de repartir responsabilidades óptimamente y realizar las decisiones correctas de forma conjunta. Por ejemplo, Adrian hizo un diagrama del MIPS útil durante el desarrollo de instrucciones, y algunas de las pruebas para las anticipaciones y detenciones.

Consideraría un 9 sobre 10 para el proyecto, asumiendo que las pruebas son generalmente correctas (una vez resueltos los errores anteriores todos los códigos funcionan de la forma esperada) y significativas, y con algún fallo en la memoria.

4. Cuantificación de horas dedicadas

Horas/Tarea	Dorian	Adrián	Total
Estudio del MIPS, VHDL, entorno, instalación…	3	5	8
Adición de las nuevas instrucciones	2	1	3
Gestión de excepciones	2	2	4
Gestión de riesgos	4	2	6
Depuración, verificación y programas de prueba	15	4	19
Memoria	5	4	9
Total	31	28	**59 horas**

Anexo: Descripción de tests diseñados

Test 2:

(El test 1 es el código retardado sin NOP)

Código de alto nivel

```
void main() {
   int a = 1, b = 2, c = 3, d = 4;

   int i = a + b; // = 3
   int j = i + c; // = 7
   int k = i + j; // = 10
   int l = a + j; // = 8

   int e = 8;
   int m = e + i; // = 11
   int n = e + j; // = 15

   int f = 16;
   int o = k + f; // = 26
   int p = l + f; // = 24

   while(1);
}
```

Memoria de datos

@	Valor inicial	Valor final (si cambia)	Contenido
0x0	0x0		Constante 0
0x4	0x1		Constante 1
0x8	0x2		Constante 2
0xC	0x4		Constante 4
0x10	0x8		Constante 8
0x14	0x10		Constante 16
…	…	…	…

0x20	0x0	10	Contenidos r7
0x24	0x0	15	Contenidos r12

Banco de registros resultado

Registro	Resultado
1	1
2	2
3	4
4	8
5	3
6	7
7	10
8	8
9	8
10	16
11	11
12	15
13	26
14	24

Memoria de instrucciones

Etiqueta	**@**	**Código operación**	**Instr. ensamblador**	**Acción**	**Observaciones**
Reset	0x0	10210003	beq r1, r1, INI	Salta a INI (0x10)	Al recibir reset
IRQ	0x4	1021003E	beq r1, r1, RTI	Salta a RTI (0x1 00)	Al recibir IRQ
DAbort	0x8	1021005D	beq r1, r1, RT_Abort	Salta a RT_Abort (0x180)	AL recibir Data_Abort
UNDEF	0xc	1021006C	beq r1, r1, RT_UNDEF	Salta a UNDEF (0x1C0)	Al recibir UNDEF
INI	0x10	08010004	lw r1, 4(r0)	r1 = 1	
	0x14	08020008	lw r2, 4(r0)	r2 = 2	
	0x18	0803000C	lw r3, 4(r0)	r3 = 4	
	0x1C	08040010	lw r4, 4(r0)	r4 = 8	

	0x20	04222800	add r5, r1, r2	r5 = 1 + 2	Sin anticipación
	0x24	04A33000	add r6, r5, r3	r6 = 3 + 4	Anticipa rs MEM
	0x28	04A63800	add r7, r5, r6	r7 = 3 + 7	Anticipa rs WB, rt MEM
	0x2C	04264000	add r8, r1, r6	r8 = 1 + 7	Anticipa rt WB
	0x30	0C070020	sw r7, 20(r0)	Mem(20) = 10	Anticipa rt WB
	0x34	08090010	lw r9, 10(r0)	r1 = 8	
	0x38	04A95800	add r11, r9, r5	r11 = 8 + 3	Parada 1 ciclo + anticipación rs WB
	0x3C	05266000	add r12, r9, r6	r12 = 8 + 7	Anticipación rs WB
	0x40	0C0C0024	sw r12, 24(r0)	Mem(24) = 15	Anticipa rt MEM
	0x44	080A0014	lw r10, 14(r0)	r10 = 16	
	0x48	04EA6800	add r13, r7, r10	r13 = 10 + 16	Parada 1 ciclo + anticipación rt WB
	0x4C	050A7000	add r14, r8, r10	r14 = 8 + 16	Anticipación rt WB
end	0x50	1000FFFF	beq r0, r0, end	Bucle infinito	
...	...	...	...	...	...
RT_Abort	0x180	1000FFFF	beq r0, r0, RT_Abort	Bucle infinito	
...	...	...	...	...	...
RT_UNDEF	0x1C0	1000FFFF	beq r0, r0, RT_UNDEF	Bucle infinito	

Test 3:

Memoria de datos

Ver memoria test 2

Memoria de instrucciones

Etiqueta	@	Código operación	Instr. ensamblador	Acción	Observaciones
Reset	0x0	10210003	beq r1, r1, INI	Salta a INI (0x10)	Al recibir reset
IRQ	0x4	1021003E	beq r1, r1, RTI	Salta a RTI (0x100)	Al recibir IRQ
DAbort	0x8	1021005D	beq r1, r1, RT_Abort	Salta a RT_Abort (0x180)	AL recibir Data_Abort
UNDEF	0xc	1021006C	beq r1, r1, RT_UNDEF	Salta a UNDEF (0x1C0)	Al recibir UNDEF
INI	0x10	08010000	lw r1, 0(r0)	r1 = 0	
	0x14	08020004	lw r2, 4(r1)	f2 = 1	
	0x18	04221800	add r3, r1, r2	r3 = 0 + 1	
	0x1C	1022FFFC	beq r1, r2, INI	Salta a INI	1 ciclo parada, no salta
	0x20	08030008	lw r3, 8(r1)	r3 = 4	
	0x24	10630006	beq r3, r3, next	Salta a next	2 ciclos, toma salto
…	…	...	…	…	…
next	0x40	04221800	add r3, r1, r2	r3 = 0 + 1	
	0x44	80600000	wro r3	Output 1	2 ciclos
	0x48	04432000	add r4, r2, r3	r3 = 1 + 1	
	0x4C	04822800	add r5, r4, r2	r4 = 2 +1	
	0x50	80800000	wro r4	Output 2	1 ciclo
	0x54	04223000	add r6, r1, r2	r6 = 0 + 1	
	0x58	04643800	add r7, r3, r4	r7 = 1 + 2	
	0x5C	10C1FFF3	beq r6, r1, INI	Salta a INI	1 ciclo, no salta
	0x60	04224000	add r8, r1, r2	r8 = 0 + 1	
	0x64	11080006	beq r8, r8, next2	Salta a next2	2 ciclos, salta
…	…	…	…	…	…
	0x80	08010000	lw r1, 0(r0)	r0 = 0	

	0x84	80200000	wro r1	Output 0	2 ciclos
	0x88	08020004	lw r2, 4(r0)	r2 = 1	
	0x8C	08030008	lw r3, 8(r0)	r3 = 2	
	0x90	80400000	wro r2	Output 1	1 ciclo
end	0x94	1000FFFF	beq r0, r0, end	Bucle infinito	
...	...	...	...	...	...
RT_Abort	0x180	1000FFFF	beq r0, r0, RT_Abort	Bucle infinito	
...	...	...	...	...	...
RT_UNDEF	0x1C0	1000FFFF	beq r0, r0, RT_UNDEF	Bucle infinito	

Test 4:

Código de alto nivel

```
// Constantes
const int sp_init = 256;
const int zero = 0;
const int one = 1;
const int eight = 8;
const int twentyfour = 24;

// Para calcular fibonacci
int digito1 = 0;
int digito2 = 1;
int iter = 1;

// Para IRQ
const int i_char = 73;  // 0x49 = 'I'
const int bang = 33     // 0x21 = '!'

// Rutina servicio IRQ: Calcula y escribe "IRQ!" como salida
// IRQ! = x049 0x52 0x51 0x21
void RTI(void) __irq
{
   int c = i_char;
   Print(c);           // 'I'
   c += 4 - 1;
   Print(c);           // 'R'
   c -= 1;
   int c2 = bang;
   Print(c);           // 'Q'
   Print(c2);          // '!'
}

// Cualquier acceso mal alineado es fin de ejecución
void Abort(void) __abort { while(1); }

// Cualquier instrucción indefinida es fin de ejecución
void UNDEF(void) __undef { while(1); }

void main()
{
   int i = 0;

   while (i < iter)
   {
       int aux = digito1 + digito2;    // Obtiene el siguiente número de
fibonacci
       Print(digito2);                 // Imprime el digito 2
```

```
        digito1 = digito2;              // Actualiza variables
        digito2 = aux;
        i++;
    }

    while(1);
}
```

Memoria de datos

@	Valor inicial	Valor final (si cambia)	Contenido
0x0	0x100		sp_init
0x4	0x0		Cte. 0
0x8	0x1		Cte. 1
0xC	0x8		Cte. 8
0x10	0x18		Cte. 24
0x14	0x0	0x179 (377)	digito1
0x18	0x1	0x262 (610)	digito2
0x1C	0xf		iter
0x20	0x49		i_char
0x24	0x21		bang
…			
0x100	0x0		Inicio pila

Memoria de instrucciones

Etiqueta	@	Código operación	Instr. ensamblador	Acción	Observaciones
Reset	0x0	10210003	beq r1, r1, INI	Salta a INI (0x10)	Al recibir reset
IRQ	0x4	1021003E	beq r1, r1, RTI	Salta a RTI (0x1 00)	Al recibir IRQ
DAbort	0x8	1021005D	beq r1, r1, RT_Abort	Salta a RT_Abort (0x180)	AL recibir Data_Abort

UNDEF	0xc	1021006C	beq r1, r1, RT_UNDEF	Salta a UNDEF (0x1C0)	Al recibir UNDEF
INI	0x10	081F0000	lw r31, 0(r0)	r31 = SP	Será seguro interrumpir tras 4 ciclos
	0x14	0806001C	lw r6, 1C(r0)	r6 = iter	
	0x18	8040008	lw r4, 8(r0)	r4 = 1	
	0x1C	8050004	lw r5, 4(r0)	r5 = 0 (i)	
Main	0x20	10A60008	beq r5, r6, end	Salta a end si i = 15	2 ciclos de parada en la 1ª iteración
	0x24	8020018	lw r2, 18(r0)	r2 = digito2	
	0x28	80400000	wro r2	IO_output <- r2	2 ciclos de parada
	0x2C	8010014	lw r1, 14(r0)	r1 = digito1	
	0x30	4411800	add r3, r2, r1	r3 = digito1 + digito2	1 ciclo parada + anticipación WB rt
	0x34	0C030018	sw r3, 18(r0)	Mem(digito2) = digito1 + digito2	
	0x38	0C020014	sw r2, 14(r0)	Mem(digito1) = digito2	
	0x3C	04A42800	add r5, r5, r4	i++	
	0x40	1000FFF7	beq r0, r0, Main	Salto a Main	Siempre genera riesgos control
end	0x44	1000FFFF	beq r0, r0, end	Bucle infinito	
...	...	...	...	...	...
RTI	0x100	0FE10000	sw r1, 0(r31)	Guarda registros en pila	
	0x104	0FE20004	sw r2, 4(r31)		
	0x108	0FE30008	sw r3, 8(r31)		
	0x10C	0FE4000C	sw r4, C(r31)		
	0x110	0FE50010	sw r5, 10(r31)		
	0x114	0FE60014	sw r6, 14(r31)		
	0x118	8010010	lw r1, 10(r0)	r1 = 24	
	0x11C	07E1F800	add r31, r1, r31	SP = SP + 24	1 ciclo de parada + anticipación WB rs
	0x120	8020020	lw r2, 20(r0)	r2 = 'I'	
	0x124	80400000	wro r2	IO_output <- r2	2 ciclos de parada
	0x128	8030008	lw r3, 8(r0)	r3 = 1	
	0x12C	0804000C	lw r4, C(r0)	r4 = 1	

	0x130	4821000	add r2, r4, r2	r2 = 'I' + 8 = 'Q'	1 ciclo parada + anticipación WB rs
	0x134	4431000	add r2, r2, r3	r2 = 'Q' + 1 = 'R'	Anticipación MEM rs
	0x138	80400000	wro r2	IO_output <- r2	2 ciclos de parada
	0x13C	4431001	sub r2, r2, r3	r2 = 'Q' - 1 = 'Q'	
	0x140	8050024	lw r5, 24(r0)	r5 = '!'	
	0x144	80400000	wro r2	IO_output <- r2	1 ciclo de parada
	0x148	80A00000	wro r5	IO_output <- r5	1 ciclo de parada
	0x14C	07E1F801	sub r31, r31, r1	SP = SP - 20	
	0x150	0BE10000	lw r1, 0(r31)	Recupera registros de pila	
	0x154	0BE20004	lw r2, 4(r31)		
	0x158	0BE30008	lw r3, 8(r31)		
	0x15C	0BE4000C	lw r4, C(r31)		
	0x160	0BE50010	lw r5, 10(r31)		
	0x164	0BE60014	lw l6, 14(r31)		
	0x168	20000000	rte	Sale de RTI	
...	...	...	...	...	...
RT_Abort	0x180	1000FFFF	beq r0, r0, RT_Abort	Bucle infinito	
...	...	...	...	...	...
RT_UNDEF	0x1C0	1000FFFF	beq r0, r0, RT_UNDEF		

Acciones banco de pruebas

```
-- Genera una interrupción durante el primer ciclo de un riesgo de datos en
el primer BEQ
-- No se hace la parada y vuelve al LW
WAIT FOR CLK_period * 7;
IRQ <= '1';
WAIT FOR CLK_period;
IRQ <= '0';

-- Genera una interrupción en el segundo ciclo de parada del WRO
-- Vuelve a WRO porque por la 1a parada no hay instrucción en EX
WAIT FOR CLK_period * 57;
IRQ <= '1';
WAIT FOR CLK_period;
IRQ <= '0';
```

```
-- Genera una interrupción en el primer ciclo de parada del ADD
-- Vuelve al LW porque no se hace la parada
WAIT FOR CLK_period * 53;
IRQ <= '1';
WAIT FOR CLK_period;
IRQ <= '0';

-- Genera una interrupción en el segundo SW
-- Vuelve al LW porque el primer SW no pasa a EX
WAIT FOR CLK_period * 54;
IRQ <= '1';
WAIT FOR CLK_period;
IRQ <= '0';

-- Interrumpe constantemente durante un tiempo.
-- Se realiza tras un salto, por lo que vuelve al IF (1er BEQ)
-- Tras las RTE también se vuelve desde la instr. en IF
WAIT FOR CLK_period * 56;
IRQ <= '1'; -- Ignora las IRQ mientras está tratando una
WAIT FOR CLK_period * 120;
IRQ <= '0';
```