Arquitectura de Computadoras 2023

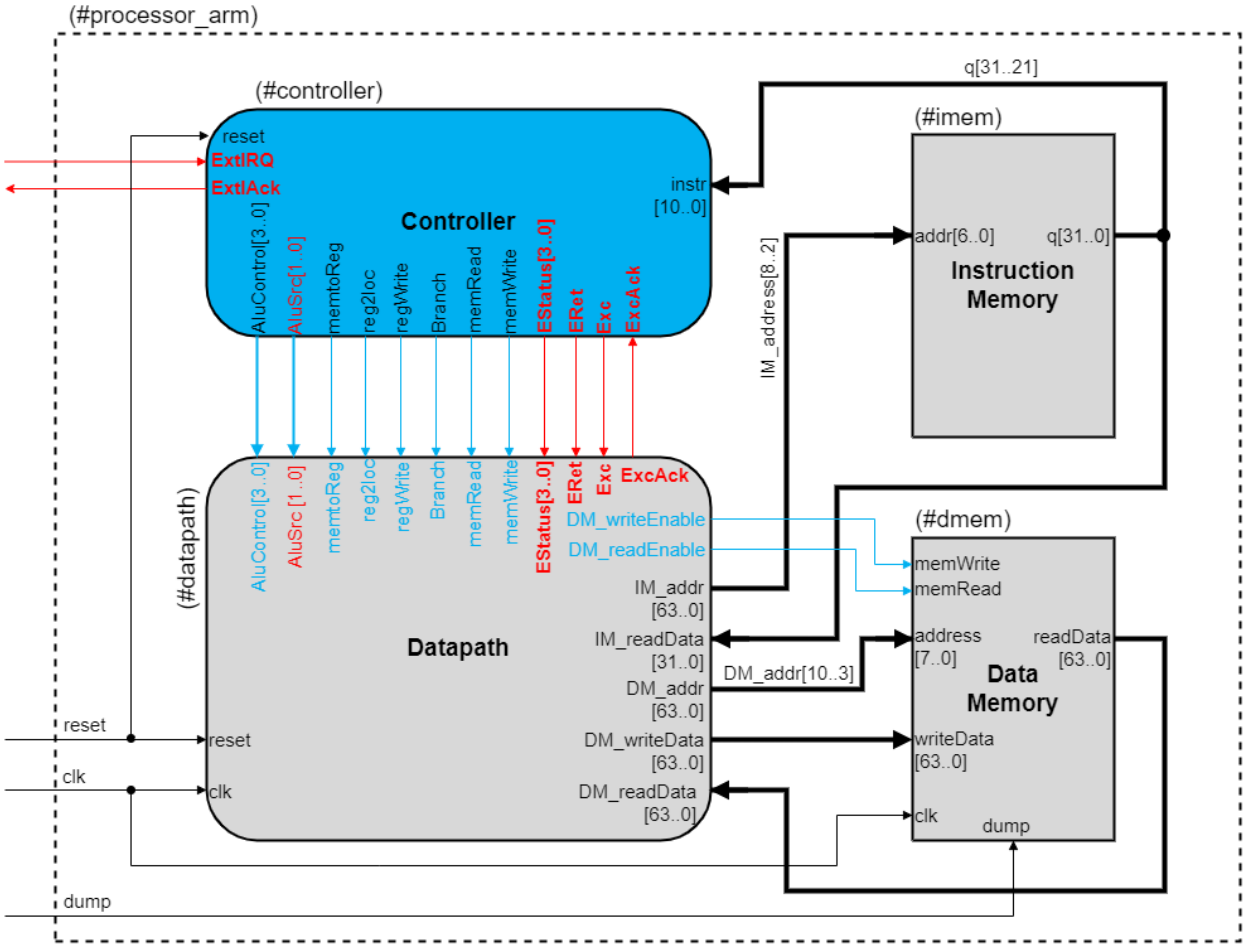
Práctico N° 3: Procesador de un ciclo con excepciones

Introducción

El objetivo de este práctico es aplicar los conceptos aprendidos de Excepciones e Interrupciones de E/S y dotar a nuestro procesador de un ciclo (construido y probado en los prácticos 1 y 2) de los recursos necesarios para procesar una excepción, tanto por eventos propios de la CPU, como un evento de interrupción de E/S. Las Fig. 1 y 2 muestran una posible implementación propuesta, donde los bloques agregados o modificados están resaltados en color rojo. Este práctico será desarrollado sobre un proyecto en System Verilog que se adjunta con esta guía.

Todos los recursos nuevos del *datapath* se implementaron en un nuevo módulo llamado *#exception* (con excepción de los dos MUX agregados en los módulos *#fetch* y *#execute*), respetando los nombres de las figuras. El diagrama de la figura no muestra este nuevo módulo para facilitar la visualización de las modificaciones y su funcionamiento.

Figura 1: Top Level ARM processor



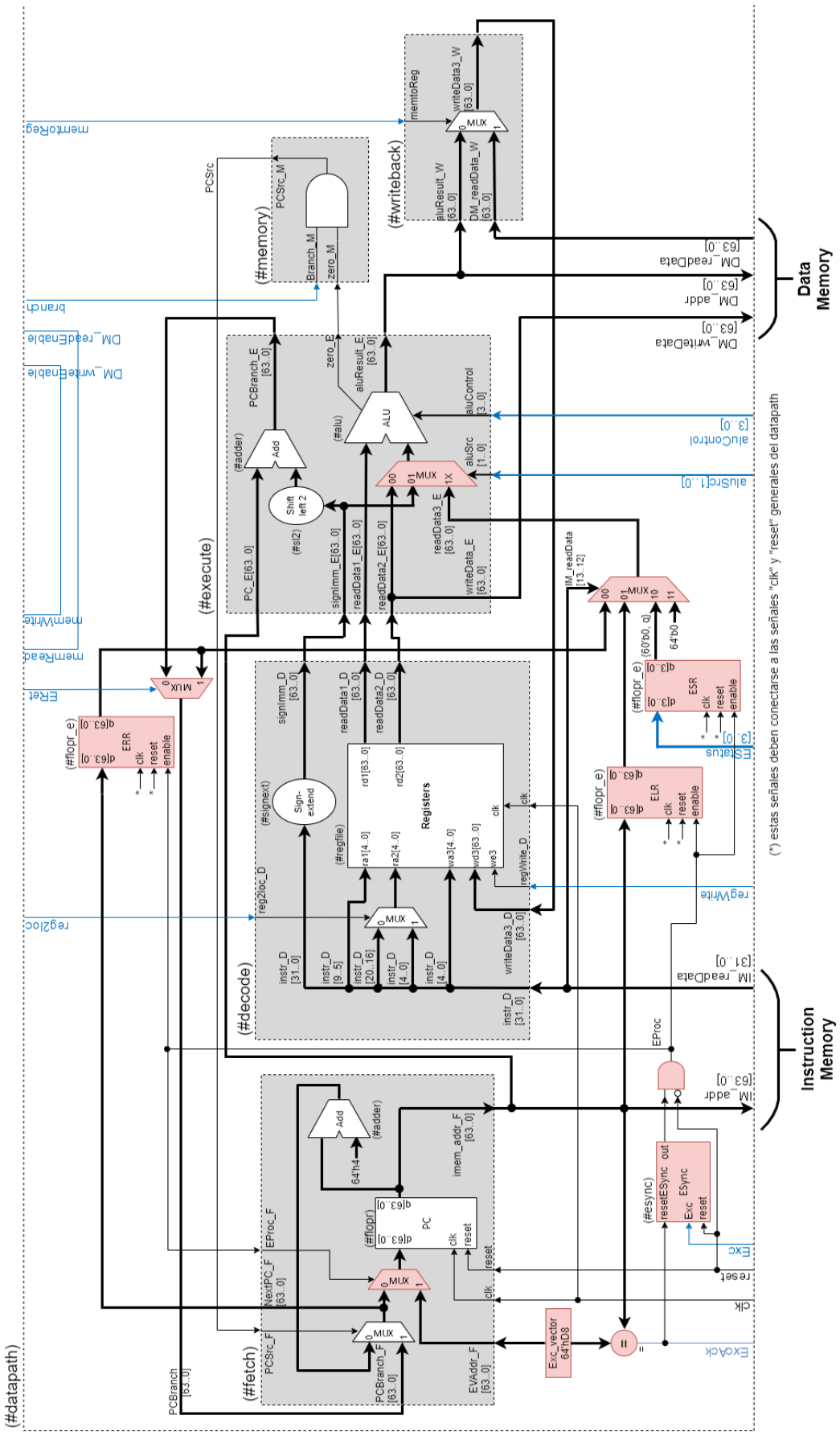


Figura 2: Datapath ARM processor

En la implementación del esquema dado, fue necesaria la creación de los siguientes módulos:

- *#flopr_e*: Este módulo posee el mismo funcionamiento de un FF-D pero con el agregado de una entrada de *enable*, de tal forma que solo actualiza el valor de su salida si el valor de dicha entrada es *enable* = 1. Caso contrario no se realiza ninguna acción.
- *#mux4*: Se trata de un multiplexor de cuatro canales de entrada de N bits de ancho y una entrada de selección de dos bits.
- *#ESync*: Es un bloque que sincroniza los eventos de excepción del microprocesador. Su funcionamiento puede sintetizarse de la siguiente forma:
  - La salida *out* debe valer '1' ante un flanco ascendente de la entrada *Exc*
  - La salida *out* debe valer '0' ante un flanco ascendente de las entradas *resetESync* y/o *reset.*
  - Sin cambios en *out* para cualquier otro evento de entrada.
- *#comp_n*: Circuito combinacional que compara dos números de N bits c/u. Si ambos números son iguales la salida es '1', caso contrario, '0'.
- El bloque llamado como *Exc_vector* de la fig. 2 no fue implementado como tal, ya que solo representa el valor constante de la dirección donde se ubica el vector de interrupciones. En este caso, el vector apunta a la instrucción 54, dejando disponibles 53 instrucciones para el programa principal y 75 instrucciones para el código de la ISR (53 + 75 = 128 palabras de 32 bits de *#imem*). Las entradas conectadas a este bloque se reemplazan por el valor literal 216 en 64 bits (64'hD8). Esta constante resulta de multiplicar las 54 instrucciones por las 4 palabras de memoria (bytes) que ocupa cada instrucción.

Ejercicio 1: Implementación del bloque *#controller*

A partir del módulo *#controller* del **procesador de un ciclo <u>sin</u> excepciones**, introducir las modificaciones necesarias a fin generar el comportamiento de las señales de control de los nuevos recursos, y agregar a la ISA del procesador dos instrucciones nuevas: ERET y MRS. A continuación una breve descripción de cada una de ellas:

a) **ERET** (Exception Return) ✓

Esta instrucción es un salto incondicional que se utiliza para retornar al flujo de programa original, previo a la ocurrencia del evento de excepción. El registro PC toma el valor de la dirección que contiene la instrucción que se hubiera ejecutado de no haber ocurrido el evento de excepción, la cual se almacenó en el registro de sistema ERR (Exception Return Register).

Sintaxis: `ERET` Tipo: R Acción: PC = R[ERR]

OpCode: 1101011_0100

31 21	20 16	15 10	9 5	4 0
opcode	**Rm**	**shamt**	**Rn**	**Rd (Rt)**
1 1 0 1 0 1 1 0 1 0 0	1 1 1 1 1	0 0 0 0 0 0	1 1 1 1 1	0 0 0 0 0

b) **MRS** (Move (from)SystemReg to GeneralPurposeReg)
Instrucción que copia el contenido de un registro de sistema a un registro de propósito general. Se utiliza para poder acceder a la información que proporcionan los registros de sistema ante la ocurrencia de un evento de excepción.

Sintaxis: `MRS <Rt>, <systemReg>` Tipo: S (new!)
Acción: R[Rt] = R[systemReg]
OpCode: `1101010100(1)`

argumento <systemReg> = "S<2+op0>_<op1>_<CRn>_<CRm>_<op2>"
- S2_0_C0_C0_0 → ERR → (CRn = "0000")
- S2_0_C1_C0_0 → ELR → (CRn = "0001")
- S2_0_C2_C0_0 → ESR → (CRn = "0010")
- S2_0_C3_C0_0 → Reservado → (CRn = "0011")

31 21	20 19	18 16	15 12	11 8	7 5	4 0
OpCode	**op0**	**op1**	**CRn**	**CRm**	**op2**	**Rt**
1 1 0 1 0 1 0 1 0 0 1	1 X	X X X	C C C C	X X X X	X X X	R R R R R

Funcionamiento del bloque *#controller*:

- Agregar la entrada **reset** al bloque *#maindec*. Si esta vale '1', todas las salidas del bloque *#controller* deben valer '0'.
- Agregar un puerto de salida *NotAnInstr* al módulo *#maindec*, de tal forma que este tome el valor '1' cuando se ingresa un **opcode invalido**, caso contrario debe valer '0'. El resto de las señales de control deben tomar los valores indicados en la tabla correspondientes a la fila "Invalid OpCode".
- La salida ***Exc*** de *#controller* debe ser el resultado de una operación OR entre la entrada *ExtIRQ* y la señal interna *NotAnInstr.*
- Ante la ocurrencia de una flanco ascendente en ***ExtIRQ*** (indicada en la tabla por la fila "External IRQ") las señales de control generadas en *#maindec* no sufren modificaciones.
- Las señales ***EStatus*** deben tomar los valores indicados en la tabla, según el caso correspondiente.
- La salida del modulo *#controller*, **ExtIAck** es '1' cuando *ExcAck* = '1' y *ExtIRQ* = *'1', caso contrario debe valer '0'*.

A continuación se resume el comportamiento de las señales de control en función de las instrucciones y eventos de excepción. Las encerradas en trazo grueso corresponden a aquellas generadas por *#maindec.* Las modificaciones respecto al procesador original se encuentran resaltadas en color rojo. Las 'X' representan condiciones sin cuidado, mientras que el símbolo '-' representa que no debe tomarse ninguna acción.

Instruct	reg2 loc	ALUSrc [1..0]	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	ALU Op[1..0]	ERet	EStatus [3..0]
R-type	0	00	0	1	0	0	0	10	0	0000
LDUR	X	01	1	1	1	0	0	00	0	0000
STUR	1	01	X	0	0	1	0	00	0	0000
CBZ	1	00	X	0	0	0	1	01	0	0000
ERET	0	00	X	0	0	0	1	01	1	0000
MRS	1	1X	0	1	0	0	0	01	0	0000
Invalid OpCode	X	XX	0	0	0	0	0	XX	0	0010
External IRQ	-	- -	-	-	-	-	-	- -	-	0001

El módulo *#alu_control* **NO** sufre modificaciones en su funcionamiento, ya que las nuevas instrucciones utilizan el caso de ALUOp = '01', al igual que en el caso de la instrucción CBZ.

Instruction	ALUOp	Instruction Op	OpCode field	ALU Action	ALU Control
CBZ **ERET** **MRS**	01	Comp & branch on Z **Exception return** **Move SReg to GPReg**	XXXXXXXXXXX	Pass input b	0111

Ejercicio 2: Implementación de la instrucción BR (Branch to Register)

Para que un procesador pueda ejecutar correctamente las tareas asociadas a un vector de excepciones real, es indispensable que el mismo sea capaz de cambiar completamente de contexto de ejecución. Eso incluye poder saltar a cualquier bloque de código contenida en memoria de programa. Es por esto que se propone en este ejercicio la modificación del procesador de un ciclo con excepciones a fin que permita la ejecución de la instrucción BR.

Las especificaciones para la implementación de esta instrucción pueden obtenerse de la Reference Data LEGv8 (GreenCard LEGv8) del libro "Computer Organization and Design - ARM Edition", y se resumen a continuación:

- BR: Instrucción de salto incondicional. El registro PC toma el valor almacenado en el registro referenciado por Rn.

  Sintaxis: `BR <Rn>` Tipo: R Acción: PC = R[Rn]
  OpCode: 1101011_0000

31 21	20 16	15 10	9 5	4 0
opcode	**Rm**	**shamt**	**Rn**	**Rd (Rt)**
1 1 0 1 0 1 1 0 0 0 0	1 1 1 1 1	0 0 0 0 0 0	R R R R R	0 0 0 0 0

Ejercicio 3: TestBench

Escribir el módulo TestBench en SystemVerilog y un programa en ASM LEGv8 a fin de corroborar el correcto funcionamiento del procesador y el tratamiento de las excepciones:

- El TestBench debe permitir la ejecución del programa cargado en *#imem* y generar en ciertos momentos pulso sobre la señal *ExtIRQ* y corroborar el procesamiento de la interrupción mediante el assert de la señal *ExtIACK* en el siguiente ciclo de CLK. Recordar que el código del vector de excepción debe estar ubicado en la instrucción número 54 (.org 0xD8) de la memoria de instrucciones *#imem*.

- El programa de assembler puede ser escrito sobre el archivo template <main.s> disponible en los adjuntos de esta guía. Este programa debe contener una secuencia de prueba que denote el correcto funcionamiento de todas las instrucciones del microprocesador, (LDUR, STUR, CBZ, ADD, SUB, AND, ORR, incluidas BR, ERET y MRS). El compilado del programa debe cargarse en la inicialización del módulo *#imem*.

- El programa principal (no ISR) debe contener además una instrucción de opcode no válido (opcode corrupto o inexistente en nuestra ISA).

- Como mínimo, se pretende que el código de la ISR identifique que tipo de interrupción se trata y lleve la cuenta en dos registros (X29 y X30 por ejemplo) el número de interrupciones de cada tipo que se procesaron.

- En caso de que se procese una instrucción no válida, debe saltarse esa instrucción y continuar con el flujo normal del programa (retornar a ELR+4), en el caso de una interrupción externa se retorna al flujo normal del programa (usando ERET).

- Se recomienda modelar el programa principal en un bucle infinito de forma en que puedan registrar varias interrupciones externas y que este no contenga la instrucción no válida.

Aclaración: Cada vez que se desea producir una interrupción externa se debe generar un flanco ascendente en la entrada ExtI. Esta debe permanecer en ‘1’ durante el tiempo que dure un ciclo de clock, sin embargo, este evento debe estar desfazado respecto al flanco ascendente de clock.

A TENER EN CUENTA:

- Para obtener el código ensamblado en hexadecimal, se debe escribir el programa a implementar en el archivo <main.asm>, y luego escribir por terminal:

  ```
  $ make
  ```

  y copiar en el módulo **imem** las instrucciones generadas en “main.list”, respetando el formato del ejercicio 4 de la guía 1.

  Nota: verificar que se tiene instalada la toolchain de aarch64, caso contrario escribir:

  ```
  $ sudo apt install gcc-aarch64-linux-gnu
  ```

- Antes de comenzar, verificar que los registros X0 a X30 estén inicializados con los valores deseados del módulo **regfile**.

Ejercicio 4

Analizar el comportamiento del procesador con manejo de excepciones implementado y responder con V (verdadero) o F (falso) a las siguientes afirmaciones. En caso de ser falsa, elaborar la respuesta correcta:

a) La ocurrencia de una excepción en un procesador causa que la ejecución secuencial del código se vea interrumpida necesariamente.
b) La diferencia entre un evento de interrupción y de excepción es que la primera es causada por un recurso del procesador, mientras que la segunda se trata de un evento proveniente de un controlador de E/S externo.
c) Los registros de excepción son utilizados típicamente en la ISR o ESR (Interrupt /Exception Service Routine) para poder procesar debidamente el evento ocurrido.
d) La dirección donde se aloja la ISR (vector de interrupciones) es fija sólo si el sistema posee solo un módulo de E/S que genera interrupción.
e) El registro ESR (Exception Syndrome Register) contiene una referencia a la dirección de memoria de la instrucción en ejecución al momento de la ocurrencia de la excepción.
f) Si un procesador no reconoce una instrucción se genera un evento de excepción.
g) Que una fuente de interrupción externa sea “enmascarable” significa que el procesador puede retrasar su ejecución si tiene eventos de mayor prioridad pendientes.
h) La instrucción ERET (Exception Return) retorna a la posición del PC (Program Counter) al momento de la excepción más cuatro (PC excepción + 4).

Ejercicio 5

Considere que la siguiente sección de código está presente en el vector de excepciones del procesador implementado.

```
1>    exc_vector: MRS X9, S2_0_C2_C0_0
2>                CMP X9, 0x01
3>                B.EQ trap
4>                MRS X9, S2_0_C0_C0_0
5>                BR X9
6>    trap:       B trap
```

Seleccionar las respuestas correctas de las siguientes afirmaciones:

01. “Ante la ocurrencia de una excepción, el código queda atrapado en un lazo infinito…”
    a. … solo si se trata de una excepción de OpCode invalido.
    b. … solo si se trata de una excepción por interrupción externa.
    c. … siempre, independientemente del tipo de excepción.
    d. Ninguna de las anteriores es correcta
02. “Si suponemos que la instrucción de la línea 2> está corrompida en memoria de forma permanente (generando un OpCode invalido), considerando la implementación particular de nuestro procesador, el mismo...”:
    a. … no realiza ninguna acción porque ya está en el vector de excepciones y retorna normalmente.
    b. … queda atrapado en el bucle infinito del label “trap”.
    c. … genera un comportamiento impredecible, porque este caso no está contemplado en la lógica de excepciones.
    d. Ninguno de los anteriores es correcta
03. “Este código retorna a la dirección de memoria donde se encuentra …”:
    a. … la instrucción que generó la excepción por OpCode invalido
    b. … la siguiente instrucción que debía ejecutarse de no haberse producido la excepción por OpCode invalido.
    c. … la instrucción que estaba en ejecución al generarse una excepción por interrupción externa.
    d. … la siguiente instrucción que debía ejecutarse de no haberse producido una excepción por interrupción externa.

Ejercicio 6

Considere que la siguiente sección de código está presente en el vector de excepciones del procesador implementado.

```
1>      exc_vector:   MRS X9, S2_0_C2_C0_0
2>                    CMP x9, 0x01
3>                    B.NE end
4>                    MRS X10, S2_0_C1_C0_0
5>                    MOVZ X9, #0x8B1F, LSL #16
6>                    MOVK X9, #0x03FF, LSL #0
7>                    STURW W9, [X10, #0]
8>      end:          ERET
```

Seleccionar las respuestas correctas de las siguientes afirmaciones:

01. “Ante la ocurrencia de una excepción por OpCode invalido, este código... ”
    a. … siempre retorna a la dirección de memoria de la instrucción que generó la excepción + 4.
    b. … reemplaza la instrucción corrupta que generó la excepción por una instrucción válida.
    c. Ninguna es correcta
    d. Ambas son correctas.
02. “Si suponemos que la instrucción de la línea 3> está corrompida en memoria de forma permanente (generando un OpCode invalido), considerando la implementación particular de nuestro procesador, el mismo...”
    a. … genera un comportamiento impredecible, porque este caso no está contemplado en la lógica de excepciones.
    b. … queda atrapado en el bucle infinito.
    c. … no realiza ninguna acción porque ya está en el vector de excepciones y retorna normalmente.
    d. Ninguna de los anteriores es correcta.
03. “Ante la ocurrencia de una excepción por interrupción externa, este código... ”
    a. … no realiza ninguna acción.
    b. … siempre retorna a la dirección de memoria de la instrucción que generó la excepción + 4.
    c. Ambas son correctas.
    d. Ninguna es correcta

Ejercicio 7

Considere que la siguiente sección de código está presente en el vector de excepciones del procesador implementado. En el caso de una excepción por OpCode invalido, este código deberá ejecutar un procedimiento alojado en la dirección 0x0400, usando X0 como argumento que contenga la dirección del OpCode invalido.

Completar el código con los argumentos faltantes:

```
esr_address:
            mrs x9,________
            subis xzr, x9, #____
            b.ne esr_end
            mrs ____, s2_0_c1_c0_0
            add x10, x0, xzr
            movz x9, #0x______, lsl #0
            br __________
esr_end:    eret
```

Ejercicio 8

Considere que la siguiente sección de código está presente en el vector de excepciones de un microprocesador LEGv8 (ISA completa), con el mismo tratamiento de excepciones utilizado hasta el momento.

- En el caso de una excepción por **Interrupción externa (IRQ)**, este código deberá ejecutar la **ISR** alojada en la dirección con etiqueta "***isr_proc***". Una vez que se retorne de la ISR, se debe retomar el flujo original del programa previo a la ocurrencia de la interrupción.
- En caso de un **OpCode invalido** se debe reemplazar el contenido de la memoria que contiene la instrucción corrompida con el valor **0x8B1F03FF**. Luego se debe forzar la ejecución de este nuevo OpCode.
- Cualquier otra fuente de excepción el procesador debe quedar atrapado en un lazo infinito dentro del vector de excepciones.

1. Completar el código con los argumentos faltantes.

```
 exc_vector:   mrs  x9, _____________
               subis xzr, x9, ____
               b.ne jmp1
question1:     ____ isr_proc
               eret
jmp1:          subis xzr, ____, 0x02
               b.ne exc_trap
               movz x9, _______, lsl #16
               movk x9, #0x03FF, lsl #0
               mrs  ___, s2_0_c1_c0_0
               sturw ___,[x10, #0]
               ____ x10
exc_trap:      b exc_trap
```

2. Seleccionar todas las respuestas correctas de las siguientes afirmaciones:
    a. Si suponemos que la posición de la etiqueta "*question1*" está corrompida en memoria de forma permanente (generando un OpCode invalido), considerando la implementación particular de nuestro procesador:
        i. Ante una excepción por IRQ queda atrapado en un bucle infinito.
        ii. Ante una excepción por OpCode Invalido queda atrapado en un bucle infinito.
        iii. Se genera un comportamiento impredecible ante cualquier excepción, porque este caso no está contemplado en la lógica de excepciones.
        iv. No se realiza ninguna acción porque ya está en el vector de excepciones y retorna normalmente.
        v. En cualquier caso se reemplaza el OpCode Invalido y se retorna al flujo original del programa, previo a la ocurrencia de la primera interrupción.
        vi. Ninguna es correcta, ya que la lógica dependerá del tipo de excepción.
    b. Ante la ocurrencia de una excepción por OpCode invalido, este código...
        i. no es posible para el procesador determinar la dirección de retorno para este contexto.
        ii. siempre queda atrapado en el lazo "*exc_trap".*
        iii. siempre retorna a la dirección de memoria de la instrucción que generó la excepción + 4.
        iv. siempre retorna a la dirección de memoria de la instrucción que generó la excepción.
        v. Ninguna es correcta

Ejercicio 4: Implementación de una ISR (Interrupt Service Routine)

El objetivo de este ejercicio es escribir un código en assembler que implemente uno de los procesos más comunes asociados al vector de excepciones de un Sistema Operativo (OS) multitarea. Este proceso debe, ante la ocurrencia de una excepción por OpCode invalido, determinar qué tarea de las que actualmente gestiona el OS generó la excepción, luego eliminarla del planificador (Scheduler) y continuar con la ejecución de la próxima tarea disponible de prioridad más alta. Para implementar este código se dispone SOLO de las instrucciones LEGv8 implementadas en nuestro procesador de un ciclo, más las instrucciones que incorporamos en los ejercicios 1 y 2 de esta guia (ERET, MRS y BR).

Como ocurre en la mayoría de los OS multitarea, éste realiza la gestión de las tareas existentes a través de estructuras de datos residentes en memoria, llamadas Bloques de Control de Tareas (TBC, Task Block Control). Cada vez que se crea una tarea en el OS, se crea un TBC asociado a la misma. De igual forma, cuando una tarea se elimina del planificador, su bloque TBC también se elimina.

En nuestro OS imaginario, cada TBC de 48 bytes está formado por una estructura de 6 elementos de 64 bits (8 bytes) c/u, tal como se detalla en la siguiente tabla:

Dirección	Campo TBC (tamaño)	Descripción
[Task n + 00h]	Task ID (8 bytes)	Número único que identifica cada tarea
[Task n + 08h]	Prioridad (8 bytes)	Número de prioridad asignado a cada tarea. Un valor de prioridad único por tarea. El número más bajo representa el valor de prioridad más alto.
[Task n + 10h]	Task Status (8 bytes)	Valores permitidos: 0: Tarea eliminada 1: Tarea no asignada al planificador 2: Tarea bloqueada 3: Tarea pausada 4: Tarea en ejecución 5: Tarea lista para ejecutar
[Task n + 18h]	Puntero Inicio (8 bytes)	Dirección de memoria que contiene la primer instrucción de la tarea
[Task n + 20h]	Puntero Final (8 bytes)	Dirección de memoria que contiene la última instrucción de la tarea
[Task n + 28h]	Puntero Link (8 bytes)	Dirección de retorno, que apunta a la siguiente instrucción de la tarea que debe ejecutarse una vez que la tarea pase a estado "en ejecución".
[Task n+1 + 00h]	...	...

Los bloques están ordenados en memoria de manera consecutiva (uno a continuación del otro) y siempre ordenados según su prioridad de forma descendente, de forma tal que el bloque de prioridad más alta es el primero, y el de menor prioridad corresponde al último bloque. La dirección de memoria de inicio del primer TBC (correspondiente a la Task con prioridad más alta) se encuentra en la dirección *DM_addr* = 0x00...0400. Todos los TBC están alojados en forma contigua en memoria. Esto quiere decir que el próximo TBC se encuentra en la dirección 0x430… 0x460…etc. Un TBC con ID = 64'd0 indica el final de la lista de TBC's.

Implementacion del codigo:

Se debe implementar un código assembler ubicado en la sección correspondiente al vector de excepciones (*IM_address* = 0x00...00D8) que realice el procesamiento equivalente al de una excepción por OpCode inválido por parte de un OS. En forma resumida, el código a implementar debe realizar las siguientes acciones:

1- Haciendo uso de los registros especiales de excepción, determinar si la excepción a procesar es la de un OpCode inválido. Si fuera de cualquier otro tipo, no se realiza ninguna acción y se retorna del vector de excepciones normalmente.

2- Determinar qué tarea contiene el OpCode inválido, es decir, la que causó la excepción. Para esto debe encontrar la tarea cuyo estado sea: "en ejecución".

3- Una vez identificada la tarea se debe cambiar el Status de la misma a "Tarea eliminada".

4- Determinar la próxima tarea a ejecutar. Para esto se debe ubicar el primer TBC (prioridad más alta) cuyo Status sea igual a: "Tarea lista para ejecutar".

5- Cambiar el status de la nueva tarea a "en ejecución".

6- Saltar a la dirección de retorno de la nueva tarea en ejecución, contenido en campo "Puntero Link".

Nota: El código a implementar sólo debe contener los procedimientos descritos anteriormente. El mantenimiento de los TBC y su asignación en memoria (creación, eliminación, ordenamiento, etc.) son realizados por otras instancias del OS.

A continuación se da un ejemplo de un posible esquema de TBC contenidos en memoria (en little endian) antes de la ocurrencia de la excepción (Columna sombreada en gris). Luego de la ejecución de la ISR pedida, los bloques deberían quedar como se muestra en la columna de la derecha. Este ejemplo puede ser utilizado para corroborar el correcto funcionamiento del código generado.

Address	Campo TBC	Contenido antes exc	Contenido después exc
0x00...0400	Task ID	00 D0 FF 00 00 00 00 00	00 D0 FF 00 00 00 00 00
0x00...0408	Priority	02 00 00 00 00 00 00 00	02 00 00 00 00 00 00 00
0x00...0410	Task Status	02 00 00 00 00 00 00 00	02 00 00 00 00 00 00 00
0x00...0418	Task Start Pointer	98 00 00 00 00 00 00 00	98 00 00 00 00 00 00 00
0x00...0420	Task End Pointer	B8 00 00 00 00 00 00 00	B8 00 00 00 00 00 00 00
0x00...0428	Task Link Pointer	A0 00 00 00 00 00 00 00	A0 00 00 00 00 00 00 00
0x00...0430	Task ID	00 CA C0 00 00 00 00 00	00 CA C0 00 00 00 00 00
0x00...0438	Priority	05 00 00 00 00 00 00 00	05 00 00 00 00 00 00 00
0x00...0440	Task Status	04 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0x00...0448	Task Start Pointer	60 00 00 00 00 00 00 00	60 00 00 00 00 00 00 00
0x00...0450	Task End Pointer	88 00 00 00 00 00 00 00	88 00 00 00 00 00 00 00
0x00...0458	Task Link Pointer	70 00 00 00 00 00 00 00	70 00 00 00 00 00 00 00
0x00...0460	Task ID	00 FE CA 00 00 00 00 00	00 FE CA 00 00 00 00 00
0x00...0468	Priority	06 00 00 00 00 00 00 00	06 00 00 00 00 00 00 00
0x00...0470	Task Status	05 00 00 00 00 00 00 00	04 00 00 00 00 00 00 00
0x00...0478	Task Start Pointer	10 00 00 00 00 00 00 00	10 00 00 00 00 00 00 00
0x00...0480	Task End Pointer	58 00 00 00 00 00 00 00	58 00 00 00 00 00 00 00
0x00...0488	Task Link Pointer	20 00 00 00 00 00 00 00	20 00 00 00 00 00 00 00
0x00...0490	Task ID	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

TIP: Para facilitar la implementación de la ISR, se inicializó la **dmem** con 0x400 en la posición 0, y 0x30 en la posición 1. Estos valores literales pueden ser usados en los cálculos de las direcciones para el manejo de los TBC.

A partir de la dirección 128 están inicializados los valores del ejemplo de la tabla con el mismo propósito.