Arquitectura de Computadoras 2021

Laboratorio ARMv8 en SystemVerilog

Objetivos

- Desarrollar códigos en lenguaje SystemVerilog para describir circuitos secuenciales y combinacionales vistos en el teórico y el práctico.
- Utilizar la herramienta QUARTUS para analizar y sintetizar el código SystemVerilog.
- Aprender a reutilizar código SystemVerilog mediante módulos estructurales.
- Mediante el uso de test bench, analizar las formas de onda y testear los resultados.

Condiciones

- Realizar el trabajo práctico en grupos de **2 ó 3 personas**.
- Subir el trabajo resuelto a la carpeta de Moodle “EntregaLab1”. La fecha límite de entrega es el **lunes 15 de noviembre** (inclusive). Los trabajos que no se entreguen antes de esa fecha se consideran desaprobados.
- Quienes estén en condiciones de promocionar, deben defender en un coloquio el trabajo presentado. Deben presentarse todos los integrantes del grupo que cumplan con las condiciones y responder preguntas acerca del trabajo realizado. El coloquio es el **miércoles 17 de noviembre**. Ese día deben conectarse a las 9hs al Meet del práctico y allí coordinaremos el orden de las presentaciones. En caso de no poder asistir al coloquio en esa fecha, coordinar **previamente** con los docentes.
- En caso de cambiar su condición a “promocionada/o” luego de rendir los recuperatorios, deberán defender el trabajo el **viernes 26 de noviembre**.

Formato de entrega

- Deben entregar un archivo comprimido (tar, zip, rar, etc.), con el nombre: “ApellidoNombre1_ApellidoNombre2_ApellidoNombre3”.
- Utilizar los nombres de los módulos y señales indicados en las guías de práctico.
- Entregar el proyecto de Quartus completo de cada ejercicio resuelto (antes de enviarlo ir a "project/clean project" para eliminar todos los archivos que no son necesarios) y un pequeño informe (**en formato pdf**) que cumpla con los requerimientos descriptos al final de los ejercicios.
- No está permitido compartir código entre grupos.

Calificación

El ejercicio 1 es obligatorio. Su resolución y presentación debe estar aprobada para obtener la regularidad de la materia. Quienes resuelvan el ejercicio 2 estarán habilitados para promocionar, si cumplen con los demás requisitos. Quienes resuelvan el ejercicio 3 (no es necesario que hayan hecho el 2) obtendrán 1 punto extra para el primer parcial.

* *Importante*: verificar que no se produzcan advertencias de la herramienta durante el proceso de síntesis relacionadas con una mala interpretación del circuito a implementar.

DESARROLLO

Ejercicio 1 (obligatorio)

El laboratorio está basado en la implementación de un microprocesador ARMv8 con *Pipeline*, en versión reducida. Para comenzar el diseño, descargar de Moodle el set de archivos <PipelinedProcessorPatterson-Modules> (.sv) con la descripción de los módulos del procesador con pipeline. A continuación crear un proyecto cuya *Top-level Entity* sea <processor_arm> y agregar los archivos .sv del paquete y todos los archivos de descripción desarrollados en el la guía de práctico n°1. Finalmente verificar las conexiones resultantes según los diagramas de las figuras 1 y 2.

Recomendaciones importantes:

- Recordar inicializar los registros X0 a X30 con los valores 0 a 30 respectivamente en la implementación del bloque *#regfile*.

- Se debe modificar el bloque *#decode* a fin de agregar el puerto de entrada resaltado con un círculo rojo en la Fig. 2.

- Se debe modificar el bloque *#regfile* a fin de que si alguno, o ambos registros leídos por una instrucción en la etapa *#decode,* están siendo escritos como resultado de una instrucción anterior en la etapa *#writeback* se obtenga a la salida de *#regfile* el valor actualizado del registro.

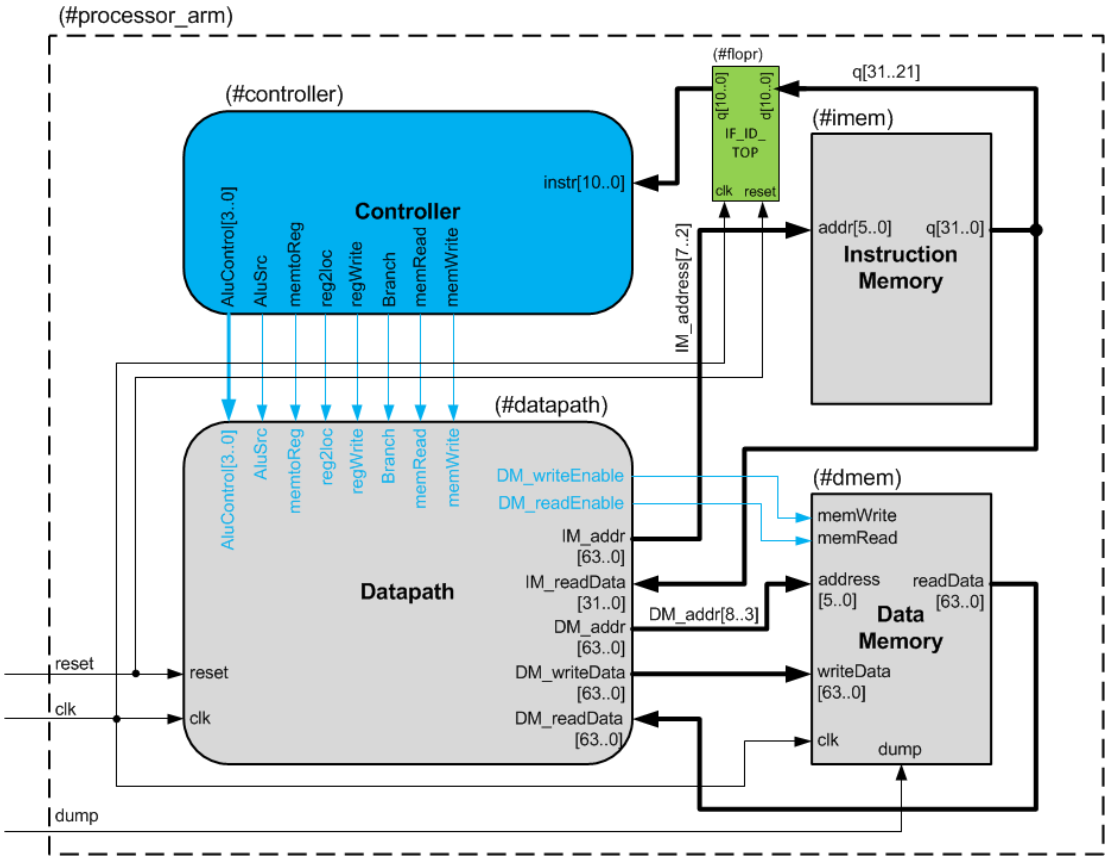


Figura 1: Esquema del *top_level_entity*

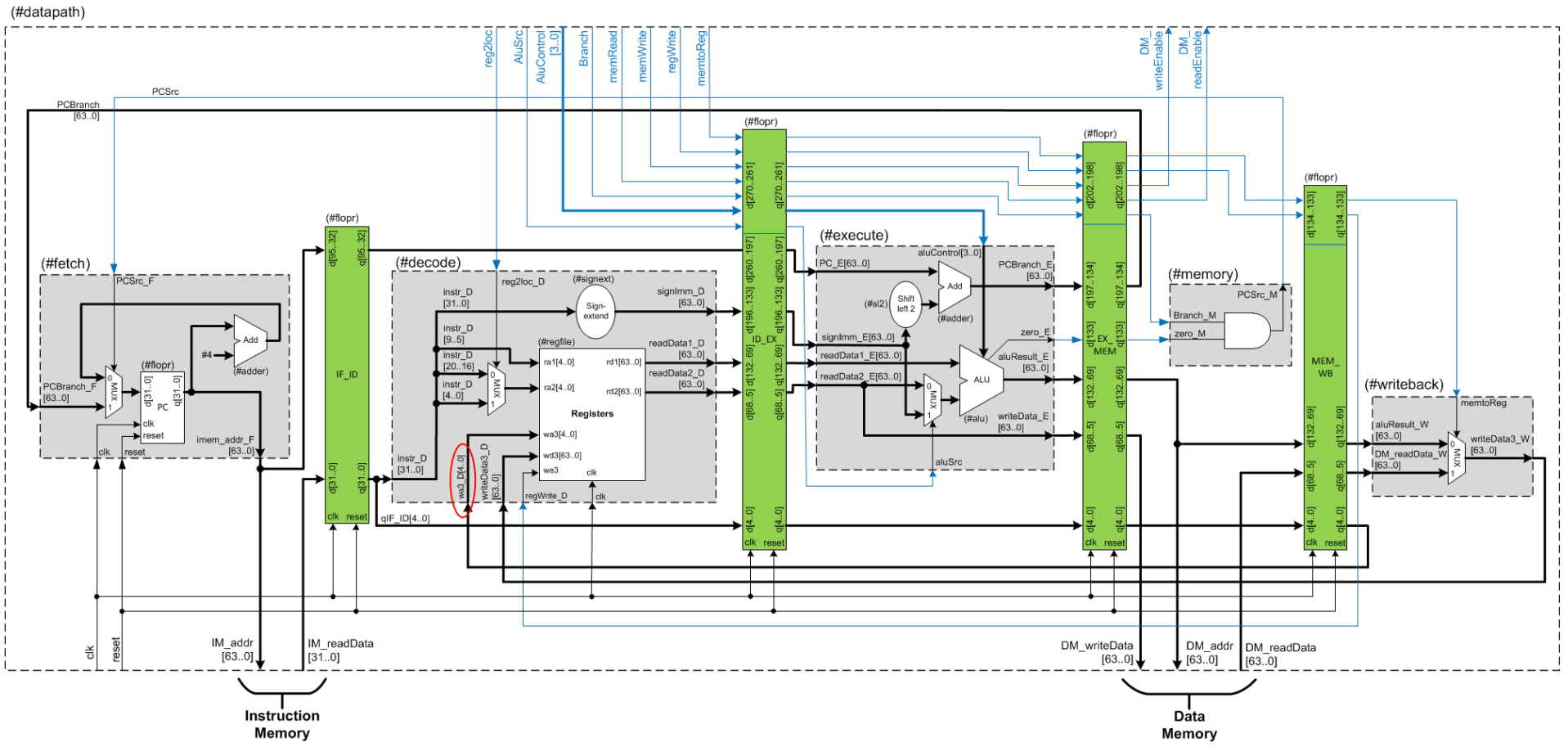


Figura 2: Esquema del *datapath*

Una vez concluida la implementación del procesador completo con pipeline, se debe verificar su correcto funcionamiento utilizando el siguiente código:

```
                                // Dirección:valor
            STUR X1, [X0, #0]     // MEM 0:0x1
            STUR X2, [X0, #8]     // MEM 1:0x2
            STUR X3, [X16, #0]    // MEM 2:0x3
            ADD X3, X4, X5
            STUR X3, [X0, #24]    // MEM 3:0x9
            SUB X3, X4, X5
            STUR X3, [X0, #32]    // MEM 4:0xFFFFFFFFFFFFFFFF
            SUB X4, XZR, X10
            STUR X4, [X0, #40]    // MEM 5:0xFFFFFFFFFFFFFFF6
            ADD X4, X3, X4
            STUR X4, [X0, #48]    // MEM 6:0xFFFFFFFFFFFFFFF5
            SUB X5, X1, X3
            STUR X5, [X0, #56]    // MEM 7:0x2
            AND X5, X10, XZR
            STUR X5, [X0, #64]    // MEM 8:0x0
            AND X5, X10, X3
            STUR X5, [X0, #72]    // MEM 9:0xA
            AND X20, X20, X20
            STUR X20, [X0, #80]   // MEM 10:0x14
            ORR X6, X11, XZR
            STUR X6, [X0, #88]    // MEM 11:0xB
            ORR X6, X11, X3
            STUR X6, [X0, #96]    // MEM 12:0xFFFFFFFFFFFFFFFF
            LDUR X12, [X0, #0]
            ADD X7, X12, XZR
            STUR X7, [X0, #104]   // MEM 13:0x1
            STUR X12, [X0, #112]  // MEM 14:0x1
            ADD XZR, X13, X14
            STUR XZR, [X0, #120]  // MEM 15:0x0
            CBZ X0, loop1
            STUR X21, [X0, #128]  // MEM 16:0x0(si falla CBZ =21)
loop1:      STUR X21, [X0, #136]  // MEM 17:0x15
            ADD X2, XZR, X1
loop2:      SUB X2, X2, X1
            ADD X24, XZR, X1
            STUR X24, [X0, #144]  // MEM 18:0x1 y MEM 19=0x1
            ADD X0, X0, X8
            CBZ X2, loop2
            STUR X30, [X0, #144]  // MEM 20:0x1E
            ADD X30, X30, X30
            SUB X21, XZR, X21
            ADD X30, X30, X20
            LDUR X25, [X30, #-8]
            ADD X30, X30, X30
            ADD X30, X30, X16
            STUR X25, [X30, #-8]  // MEM 21:0xA (= MEM 9)
finlup:     CBZ XZR, finlup
```

TENER EN CONSIDERACIÓN LOS EVENTUALES PROBLEMAS DE HAZARD (de datos y de control) QUE CONTIENE EL PROGRAMA UTILIZADO Y PLANTEAR LAS MODIFICACIONES NECESARIAS PARA EVITAR SU OCURRENCIA. Para esto, agregar instrucciones tipo “nop”, las cuales se pueden implementar como ADD XZR, XZR, XZR.

Para el informe:

- Mostrar el contenido de memoria al finalizar la ejecución del código original (sin el agregado de instrucciones nop).
- Tomar una captura de la pantalla “Wave” de ModelSim de una parte de la ejecución de este código que muestre claramente la ocurrencia de un hazard de datos, a partir del análisis de las señales involucradas.
- Tomar una captura de la pantalla “Wave” de ModelSim de una parte de la ejecución de este código que muestre claramente la ocurrencia de un hazard de control, a partir del análisis de las señales involucradas.
- Mostrar el programa modificado que se utilizó para verificar el comportamiento del procesador (con el agregado de instrucciones nop).

Ejercicio 2 (para promocionar)

Sin afectar el funcionamiento de las instrucciones ya implementadas en la versión reducida del **microprocesador con pipeline**, agregar las instrucciones de saltos condicionales *b.cond* y las instrucciones aritméticas que configuran las banderas del microprocesador: *adds* y *subs*. Estas últimas instrucciones son necesarias para que los saltos condicionales (b.cond) puedan determinar si deben saltar o no.

Para que este ejercicio sea considerado correctamente implementado, el microprocesador debe cumplir los siguientes requerimientos:

- La ALU debe generar las 4 banderas existentes en el microprocesador estudiado: Zero, Negative, Carry y oVerflow. Además debe generar la señal *write_flags* que indica si se deben actualizar las banderas o no.

- En las nuevas instrucciones adds y subs la señal *alucontrol* debe ser similar al de las add y sub (respectivamente) pero con un ‘1’ en el bit más significativo.

- Debe existir un registro de flags llamado *CPSR_flags*, de cuatro bits, que almacene el estado de las banderas (Z, N, C, V). Este registro debe escribirse únicamente cuando se ejecuta una instrucción que setee flags (adds o subs).

- Todas las demás instrucciones, incluida la de salto cbz, deben continuar funcionando normalmente.

- Debe crearse una nueva señal de control llamada *condBranch* que indica si se debe realizar un salto condicional.

- Se deben agregar las 14 condiciones de saltos existentes en el set de instrucciones LEGv8.

- El salto condicional es una instrucción tipo CB. En los cinco bits menos significativos se indica qué condición se debe cumplir para que se tome el salto de la forma en que se muestra en la siguiente tabla:

Instruction	Rt [4:0]	Instruction	Rt [4:0]
B.EQ	00000	B.VC	00111
B.NE	00001	B.HI	01000
B.HS	00010	B.LS	01001
B.LO	00011	B.GE	01010
B.MI	00100	B.LT	01011
B.PL	00101	B.GT	01100
B.VS	00110	B.LE	01101

Recomendaciones:

- Crear un nuevo bloque llamado *bCondCheck* en la etapa de memory que verifique la condición de salto.

- En caso de que en la ALU hayan implementado la resta utilizando el operador de la resta de system verilog (a - b), se recomienda modificarlo a la siguiente forma: complementar el operando b y sumar: a + complemento_2(b). Esto se debe a que la bandera de Carry se genera de forma distinta en ambos casos y es necesario que la bandera de carry sea generada de esta forma en particular para que puedan utilizarse las condiciones de salto impuestas en las siguientes tablas:

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	~(Z=0 & N=V)	B.LS	~(Z=0 & C=1)
>	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
≥	B.GE	N=V	B.HS	C=1

Signed and Unsigned numbers	
Instruction	CC Test
Branch on minus (B.MI)	N= 1
Branch on plus (B.PL)	N= 0
Branch on overflow set (B.VS)	V= 1
Branch on overflow clear (B.VC)	V= 0

Para el informe:

- Describir brevemente qué modificaciones se introdujeron (en qué módulos y con qué finalidad). Mostrar el diagrama del nuevo microprocesador, indicando las señales y módulos agregados (de ser necesarios).
- Escribir una sección de código assembler, donde se pruebe el funcionamiento de las nuevas instrucciones y se verifique que las que ya estaban implementadas continúen funcionando correctamente. Los resultados obtenidos al ejecutar cada una de las instrucciones deben guardarse en la memoria de datos.
- Mostrar el contenido de memoria al finalizar la ejecución del código anterior.

Ejercicio 3 (+1 punto)

Para la resolución de este ejercicio se debe trabajar sobre el procesador de 1 ciclo con excepciones del TP3, y **NO sobre el procesador con pipeline.**

3.0 Agrandar los módulos de memoria

El paso previo a la implementación de la consigna propuesta es necesario adaptar los tamaños de los bancos de memoria de datos y de instrucciones.

Para esto es necesario modificar la implementación del bloque de memoria *Imem* a 128 palabras de 32 bits. Por otro lado adjunto con esta guía de laboratorio le damos resuelto un nuevo bloque *Dmem* de 256 palabras de 64 bits.

TIP: Recordar modificar las conexiones de ambos módulos con el bloque de *processor_arm.*

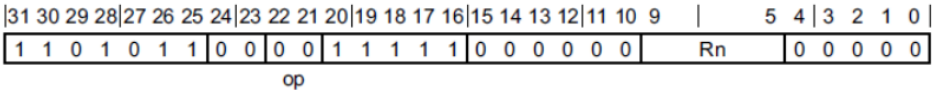
3.1 Implementación de la instrucción BR

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 a fin que permita la ejecución de la instrucción BR (Branch to Register).

Los datos para la implementación de esta instrucción (OpCode, Tipo de instrucción, comportamiento, etc.) deben obtenerse de la Reference Data LEGv8 (GreenCard LEGv8) del libro “Computer Organization and Design - ARM Edition”. Es necesario mencionar que esta instrucción posee un error de tipeo en la versión original de la GreenCard, tal como se señala a continuación:

Branch to Register	BR	R	6B0	PC = R[Rt]

El campo de la instrucción utilizado para indicar el registro que contiene la dirección de salto es en realidad **Rn**, en lugar de Rt, como lo indica la GreenCard con errores. El binario de la instrucción completa, queda conformado de la siguiente manera:



Se debe observar que la instrucción BR es del tipo R y pertenece al grupo de instrucciones de saltos INCONDICIONALES. Para la implementación se deben realizar todas las modificaciones que se consideren necesarias tanto en el bloque #datapath, como en #controller.

Una vez finalizada la implementación, cargar un código en el Instruction Memory donde se compruebe el correcto funcionamiento del set completo de instrucciones, incluida la BR.

3.2 Implementación de una ISR (Interrupt Service Routine)

El objetivo de este ejercicio es poner en práctica los conocimientos desarrollados en el Práctico 3 sobre el comportamiento de las excepciones de nuestro procesador de un ciclo, mediante la escritura de 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, en primer lugar determinar qué tarea de las que actualmente gestiona el OS generó la excepción, 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 el Práctico 3 (ERET y MRS).

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.

Con fines de claridad analicemos el siguiente ejemplo: Supongamos que una instrucción que se está por ejecutar posee un OpCode inválido y, así, se genera una excepción. En ese momento los bloques TBC estaban como se muestran a continuación (considerar organizacion little endian):

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

Una vez que se determina que se trata de una excepción por OpCode inválido, se determina que la instrucción corresponde a la tarea ID 0x00..C0CA00, ya que su Status es 0x00..04. A continuación debe eliminarse esta tarea cambiando el Status por 0x00..00.

Una vez concluido esto, se debe determinar cuál será la próxima tarea a ejecutarse. En nuestro ejemplo, las otras dos tareas presentes se encuentran una “bloqueada” (ID= 0x00..FFD000) y otra “lista para ejecutarse” (ID= 0x00..CAFE00). Esta última debe cambiar su Status por el valor 0x4 (“en ejecución”) ya que es la de mayor prioridad que está lista para ser ejecutada y se debe retornar a la dirección 0x00...0020 (valor contenido en Puntero Link).

Para el informe:

- Describir brevemente qué modificaciones se introdujeron (en qué módulos y con qué finalidad). Mostrar el diagrama del nuevo microprocesador, indicando las señales y módulos agregados (de ser necesarios).
- Escribir una sección de código assembler que cumpla con las consignas propuestas.
- Mostrar el contenido de memoria al finalizar la ejecución del código anterior, de tal forma que coincidan con el resultado mostrado en el ejemplo. Se les provee un nuevo módulo con la memoria DMEM que contiene precargados el contenido del ejemplo en las direcciones indicadas.