Universidad de Buenos Aires

Arquitectura de Microprocesadores

Preguntas Orientadoras Sobre La Arquitectura ARM

Alumno: Gonzalo Vila

19-4-2022

Contenido

Guía de preguntas Cortex-M

1. Describa brevemente las diferencias entre las familias de procesadores Cortex M0, M3 y M4.

La familia de procesadores Cortex-M se presenta como una alternativa a otras familias Cortex como Cortex-A y Cortex-R. Su característica distintiva es la de presentar circuitos integrados de bajo costo y consumo. Esto explica su popularidad en distintos ámbitos como la electrónica de consumo masivo o su uso en *IoT*.

Dentro de la familia M existen varios modelos. Algunos de los más populares son el Cortex M0, M3 y M4.

El primer punto de comparación es la arquitectura en la que están basados, mientras que el Cortex-M se apoya en la arquitectura ARMv6-M, el M3 y M4 en la ARMv7-M. Esto hace que el set de instrucciones del M0 sea más pequeño. El M0 se encuentra optimizado para bajo consumo.

El M0 se muestra menos apto para escenarios en donde se debe procesar datos complejos, es allí en donde M3 y M4, con un mayor set de instrucciones se destacan.

M3 y M4, por estar basados en la misma arquitectura comparten muchas características. M4 es superior a M3 fundamentalmente en el procesamiento de señales digitales. Algunos *benchmarks* muestran diferencias de consumo energético durante operaciones de punto flotante hasta tres veces más favorables para el M4 debido a sus optimizaciones para este escenario.

Si no realizamos un uso intensivo de las ventajas del M4, M3 es altamente comparable en términos de prestaciones y posee un menor costo.

2. ¿Por qué se dice que el set de instrucciones Thumb permite mayor densidad de código? Explique.

Mayor densidad de código implica que se puede realizar la misma tarea con un programa más pequeño. Estos es algo deseable ya que se puede fabricar un microcontrolador más sencillo, reduciéndose así los costos y el consumo energético.

El set de instrucciones *Thumb* está basado en la tecnología *Thumb-2*. Soporta el uso de instrucciones de 16 y 32 bits.

Disponer de instrucciones de 16 y 32 *bits* permite realizar optimizaciones y buscar un balance entre tamaño del código y performance. Por ejemplo, se podría utilizar operaciones de 32 *bits* en el código del manejador de una interrupción importante donde necesitamos mejor performance. En otras áreas donde priorizamos que el código sea más pequeño, podríamos utilizar instrucciones de 16 *bits*.

3. ¿Qué entiende por arquitectura load-store? ¿Qué tipo de instrucciones no posee este tipo de arquitectura?

La arquitectura *Load-Store* implica que, para realizar una operación, se debe leer los datos de la memoria y escribirlos en registros del procesador. Una vez que los datos se encuentran en registros, allí se los puede procesar y de ser necesario volcar el resultado nuevamente a la memoria. Cada uno de los pasos descriptos se realizará utilizando una operación independiente.

Load-Store, se diferencia de la arquitectura *Register-Memory* en que esta última puede combinar en una operación datos almacenados en memoria y registros. Este tipo de instrucciones no existen en *Load-Store*.

4. ¿Cómo es el mapa de memoria de la familia?

Los procesadores Cortex-M utilizan direccionamiento de memoria de 32 *bits*. Esto resulta en un espacio de memoria de 4 *GB*. Los datos e instrucciones comparten el mismo espacio de direcciones.

Los 4 *GB* de espacio de memoria se subdividen en las siguientes regiones:

Code: 512 MB para el código del programa. Incluye la tabla de vectores.

SRAM: Normalmente utilizada para conectar SRAM (usualmente *on-chip*)

Peripherals: Normalmente utilizada para conectar periféricos *on-chip*

RAM: Puede almacenar código y datos de programa.

Devices: Contiene dos slots de 512 *MB* (1 *GB* total). Se utiliza para conectar periféricos *off-chip*.

System: Contiene varias partes

- Internal Private Peripheral Bus (PPB): Se utiliza para acceder a componentes del sistema tales como NVIC, Systick y a componentes de *debug*. En la mayoría de los casos esta memoria solo puede ser accedida por código que se ejecute en modo privilegiado.
- External Private Peripheral Bus: Se incluye para que el proveedor pueda agregar componentes de propios. Este espacio de memoria solo puede ser accedido por código ejecutándose en modo privilegiado.
- Vendor-Specific Area
- Resto de la memoria: Se utiliza para componentes específicos del proveedor. Muchas veces no se utiliza.

5. ¿Qué ventajas presenta el uso de los "shadowed pointers" del PSP y el MSP?

El concepto de "*shadowed pointers*" es relevante por ejemplo en sistemas embebidos que utilizan un sistema operativo o un sistema operativo de tiempo real. En este caso, parte del *kernel* y los manejadores de excepciones utilizan el MSP (*Main Stack Pointer*) mientras que las tareas de aplicación el PSP (*Program Stack Pointer*).

Cada tarea de aplicación dispone de su espacio de *stack.* El sistema operativo actualiza el PSP durante los cambios de contexto.

Sus ventajas son:

- Si una tarea de aplicación tiene un problema que culmina en la corrupción del *stack*, el *stack* utilizado por el sistema operativo muy probablemente se encuentre intacto. Esto ayuda a hacer al sistema más robusto.
- El espacio de *stack* para cada tarea solo debe cubrir el máximo requerido más un nivel de *stack frame*. El espacio requerido para el ISR y el *nested interrupt handling* es almacenado solo en el *stack* principal.
- Hace más eficientes a los sistemas operativos creados para ARM Cortex-M.
- Se puede utilizar MPU para definir la región del *stack* que una aplicación puede utilizar. Si una tarea de aplicación provoca un *stack overflow*, el MPU puede generar una excepción de tipo *MemManage* y prevenir que escriba en direcciones por fuera del *stack space*.

6. Describa los diferentes modos de privilegio y operación del Cortex-M, sus relaciones y cómo se conmuta de uno al otro. Describa un ejemplo en el que se pasa del modo privilegiado a no privilegiado y nuevamente a privilegiado.

Los procesadores Cortex-M poseen dos modos de operación y dos niveles de privilegio.

Los modos de operación son *Thread Mode* y *Handler Mode;* los de privilegio, *Privileged* y *Non-privileged*.

El modo *handler* se utiliza para ejecutar excepciones e interrupciones, siempre utiliza un nivel de acceso de tipo *privileged*.

El modo *Thread* es a menudo llamado *User Mode* dado que es donde se ejecuta el código de la aplicación. Por defecto, el modo *Thread* se inicia en *Privileged mode*.

Desde el modo *Thread* se puede pasar a modo *non-privileged* de forma directa, pero solo se puede retornan a modo *privileged* a través del llamado a una interrupción. Esta interrupción, será ejecutada en modo handler.

7. ¿Qué se entiende por modelo de registros ortogonal? Dé un ejemplo.

Se dice que el modelo de registros es ortogonal debido a que cualquier registro puede ser utilizado para cualquier operación.

Dentro de una rutina podemos utilizar, por ejemplo, los registros R0, R1, R2 …

Todos ellos pueden ser utilizados por cualquier instrucción y no se encuentran reservados por ninguna.

8. ¿Qué ventajas presenta el uso de instrucciones de ejecución condicional (IT)? Dé un ejemplo.

Se puede utilizar la ejecución condicional para reducir el número de *branches* en el código.

Los *branches* en el código son muy costosos ya que producen que el *pipeline* deba ser descartado. Normalmente tomara tres ciclos volver a llenarlo.

Por ejemplo, una secuencia sencilla IF-THEN-ELSE normalmente recurriría a saltos. Esto puede ser reemplazado por un bloque IT.

El modo de funcionamiento de un bloque IT se puede observar en el siguiente ejemplo:

```
subs   r3, 1
ITE    EQ
ADDEQ  R1, R0, #20
ADDLE   R1, R0, #21
```

La primera instrucción actualiza los *flags* de estado. ITE marca el comienzo de un bloque de ejecución condicional IT. La instrucción siguiente hace las veces del bloque "*then*" y la subsiguiente del bloque "*else*".

Se debe destacar que todas estas instrucciones se ejecutan de manera secuencial, su naturaleza condicional hará que realicen algo o no tras la evaluación de los flags.

Es de esta forma que en ciertas circunstancias se puede eliminar el uso de "saltos" en el código y evitar el descarte del *pipeline*.

9. Describa brevemente las excepciones más prioritarias (reset, NMI, Hardfault).

La excepción *reset*, es invocada cuando se enciende el dispositivo (*power up)* o durante un *reset*. Al producirse una excepción de tipo *reset* la actividad del procesador se detiene en cualquier punto de cualquier instrucción. Tras el *reset*, la ejecución se inicia en la dirección provista por la entrada *reset* en la *vector table*. La ejecución se restablece de forma privilegiada dentro del *thread mode*. *Reset* es la excepción de mayor prioridad.

NMI es un acrónimo para *Non Maskable Interrupt;* Interrupción no enmascarable. Puede ser generada por *software* o un periférico. Esta permanentemente habilitada y tiene una prioridad fija de 2. Esto la convierte en la interrupción con la segunda mayor prioridad.

NMI no puede ser enmascarada por otra excepción. La única excepción que puede adelantársele es la de *reset*.

HardFault sucede cuando existe un error en el procesamiento de una excepción, o porque una excepción no puede ser manejada por ningún mecanismo. Tiene una prioridad de -1.

10. Describa las funciones principales de la pila. ¿Cómo resuelve la arquitectura el llamado a funciones y su retorno?

El *stack* (pila) es un mecanismo de uso de memoria que permite utilizar una porción como un *buffer* de almacenamiento de datos.

Se puede utilizar para:

- Almacenamiento temporal de los datos originales cuando una función en ejecución necesita utilizar los registros para el procesamiento de datos. Los valores iniciales pueden ser restaurados al finalizar la función de modo que el programa que la invoco no pierda los datos.
- Pasar información a funciones y subrutinas.
- Almacenamiento de variables locales.
- Almacenar el estado del procesador y los valores de los registros en caso de que se produzca una excepción (por ejemplo, una interrupción).

El primer desafío al momento de llamar a una función es que podría necesitar hacer uso de registros que están siendo utilizados por el programa principal. Estos valores deben ser preservados.

Para ello se realiza el *Stacking*, esto es, tomar el valor de los registros que se desea preservar y hacer un PUSH al *stack*.

Cuando la función termine su ejecución tenemos el problema opuesto, es decir necesitamos recuperar los datos originales para que el programa pueda continuar su ejecución. Esto lo hacemos ejecutando una operación POP por cada uno de los registros salvaguardados.

El retorno de valores desde la función se hace de forma similar, dentro de ella se hace un *push* del valor a retornar al *stack*. Desde el programa principal, este valor puede ser recuperado mediante un POP.

11. Describa la secuencia de reset del microprocesador.

La secuencia de *reset* del microprocesador puede resumirse en los siguientes pasos:

1. Después del reset el contador de programa (PC) se actualiza con la dirección 0x00000000.
2. El procesador lee el valor de la dirección 0x00000000 y lo carga en el MSP (Main Stack Pointer).
3. El procesador lee la dirección del reset handler de la dirección 0x00000004.
4. El procesador "salta" a la dirección de memoria del reset handler y comienza a ejecutar las instrucciones.
5. Se invoca a la función main del programa con el código del usuario.

El *Reset handler*, es una función normal escrita en C o *Assembly* que se encarga de inicializar el procesador y los periféricos. Por ejemplo, configura el reloj, inicializa el stack space, etc.

12. ¿Qué entiende por "core peripherals"? ¿Qué diferencia existe entre estos y el resto de los periféricos?

Los periféricos core son aquellos que se encuentran comprendidos dentro de la arquitectura de ARM. El resto de los periféricos son agregados por los fabricantes de los microcontroladores al efecto de orientar su producto a un determinado uso.

13. ¿Cómo se implementan las prioridades de las interrupciones? Dé un ejemplo

La prioridad con la que se ejecute una excepción dependerá de la prioridad asociada a la excepción y a la prioridad de la rutina que el procesador se encuentra ejecutando.
Por supuesto la afirmación anterior asume que el procesador se encuentra listo para procesar (no halted o en proceso de reset) y que las interrupciones se encuentran habilitadas.

Las interrupciones se encuentran registradas en un periférico llamado NVIC (Nested Vectored Interrupt Handler). Sera a través de él que se implemente, en un componente, el registro de las excepciones existentes y la prioridad que tienen unas sobre otras.

Para explicar el uso de prioridades podemos apelar a dos escenarios, uno es aquel en el que el procesador recibe una interrupción y luego otra de menor o igual prioridad. En este caso, la segunda quedara suspendida a la espera de que la primera concluya.

El segundo escenario es aquel en donde una excepción se encuentra en ejecución y otra de mayor prioridad es recibida. En este caso se hará un stacking de los registros de la primera y se procederá a atender a la segunda. Tras ella, La primera retomara la ejecución.

El mecanismo descripto de precedencia de excepciones es la razón por la que la que el acrónimo NVIC contiene la palabra “Nested”. Es decir las excepciones tiene niveles de importancia que se preceden unos a otros a los que llamamos “Prioridades”.

Un ejemplo del uso de prioridades en acción es cuando nuestro procesador se encuentra ejecutando una interrupción, por ejemplo, un periférico desea reportarle un dato. Durante la ejecución de esta rutina se reporta otra interrupción, definida por el usuario y con mayor prioridad que indica que el sistema debe realizar ciertas acciones de emergencia por un acontecimiento crítico. En este caso la segunda tomara precedencia y la primera podrá restablecer su ejecución solo tras su conclusión.

14. ¿Qué es el CMSIS? ¿Qué función cumple? ¿Quién lo provee? ¿Qué ventajas aporta?

CMSIS es una iniciativa de ARM para proveer librerías y una API standard para la programación de los procesadores de la familia Cortex-M. Esto permite reutilizar código e incrementar su portabilidad.

15. Cuando ocurre una interrupción, asumiendo que está habilitada ¿Cómo opera el microprocesador para atender a la subrutina correspondiente? Explique con un ejemplo.

Para explicar lo que sucede cuando ocurre una interrupción podemos tomar como ejemplo una interrupción por hardware. En este escenario, un sensor temperatura desea notificar que la temperatura excede cierto umbral. El propósito de esta interrupción es hacer que el procesador maneje ese evento “critico”.

Dentro del microcontrolador existe un periférico común a todos los microcontroladores ARM, el NVIC (Nested Vectored Interrupt Handler).

El NVIC recibe los pedidos de interrupción. Este periférico conoce los tipos de interrupciones que puede recibir, que rutina puede atender a cada uno y cuál es su prioridad relativa a los otros.

En nuestro ejemplo, el procesador será forzado a atender a una rutina de manejo para la excepción. A esta rutina la llamamos exception handler.
Sucede que al momento de presentarse la interrupción, el procesador se encontraba ejecutando código de la aplicación. Para atender la rutina, el procesador deberá hacer un “push” de los registros que estaba utilizando al stack así como la dirección de memoria que estaba utilizando en ese momento. A esta operación se la conoce como *stacking*. De esta manera, evitara que el hecho de atender la excepción, corrompa al programa en ejecución.

Habiendo tomado esta precaución, ahora el procesador es capaz de ejecutar la rutina de manejo de la excepción. Lo hará ejecutando su *exception handler*. Esto es simplemente, código C o Assembly.

En este caso podría tomar alguna acción relativa a reducir la temperatura o alertar a los operarios.

16. ¿Cómo cambia la operación de stacking al utilizar la unidad de punto flotante?

Cuando se utiliza la unidad de punto flotante, ante una excepción, se requieren 29 ciclos de procesador ya que la operación de stacking debe cargar 25 registros al stack.
Para mitigar este efecto, los procesadores cortex-M4 implementan una funcionalidad llamada “Lazy stacking”.

17. Explique las características avanzadas de atención a interrupciones: tail chaining y late arrival.

Tail chaining: Se utiliza cuando se presenta una excepción al mismo tiempo que el procesador está ejecutando otra de igual o mayor prioridad. En este caso, cuando el procesador termine de atender la primera interrupción procederá a atender la siguiente. La técnica de *Tail chaining* permite optimizar esta situación evitando recuperar los registros desde el *stack* (*unstacking*) para luego cargarlos nuevamente al *stack* (*stacking*). El procesador omite esta secuencia y directamente ejecuta el *exception handler*.

De esta forma se reduce el tiempo de espera para la atención de la segunda interrupción.

Esta técnica mejora la eficiencia energética del sistema ya que el tráfico de memoria requerido para los accesos al *stack* consume energía.

Late arrival: Si el procesador se encuentra realizando el *stacking* de una interrupción y durante el mismo se recibe una interrupción de mayor prioridad, esta última será atendida primero tan pronto como se termine el proceso de *stacking* pendiente.

18. ¿Qué es el systick? ¿Por qué puede afirmarse que su implementación favorece la portabilidad de los sistemas operativos embebidos?

Los procesadores Cortex-M poseen un *timer* integrado llamado *SysTick* (*System Tick)* que genera una excepción en intervalos regulares de tiempo.

En el contexto de un sistema operativo, se utiliza al *Systick Timer* para el manejo de tareas y cambio de contexto. Fuera de los sistemas operativos, se utiliza a este componente para implementar interrupciones, generar *delays* o medir tiempos.

El temporizador *SysTick* favorece la portabilidad de los sistemas operativos ya que es una característica que todos los procesadores Cortex-M poseen. Esto hace que esa porción del código del sistema operativo pueda funcionar en todos los procesadores de la arquitectura sin modificaciones.

19. ¿Qué funciones cumple la unidad de protección de memoria (MPU)?

El MPU es un dispositivo programable que se utiliza para definir permisos de acceso a memoria y atributos de memoria para las distintas regiones de memoria.

Los procesadores Cortex-M3 y Cortex-M4 soportan hasta 8 regiones de memoria. Cada una de ellas tiene su dirección inicial, tamaño y configuraciones.

La MPU se utiliza para hacer a los sistemas embebidos más robustos y seguros:

- Previene que las tareas de aplicación corrompan la memoria de stack o de datos utilizada por otras tareas y por el kernel del sistema operativo.
- Previene que tareas no privilegiadas accedan a ciertos periféricos que son críticos para la solidez o la seguridad del sistema.
- Se puede definir el espacio SRAM o RAM como no ejecutable para prevenir ataques de inyección de código.
- Se puede definir atributos de memoria como el *cacheability*.

Si un acceso a memoria viola los permisos definidos por la MPU, el acceso es bloqueado y se genera una excepción.

20. ¿Cuántas regiones pueden configurarse como máximo? ¿Qué ocurre en caso de haber solapamientos de las regiones? ¿Qué ocurre con las zonas de memoria no cubiertas por las regiones definidas?

Los procesadores Cortex-M3 y Cortex-M4 soportan hasta 8 regiones de memoria. Cada una de ellas tiene su dirección inicial, tamaño y configuraciones.

En caso de haber solapamiento los atributos de acceso y privilegios estarán basados en los definidos para la región de mayor numeración.

Si se intenta utilizar una zona de memoria que no está cubierta por las regiones definidas, esto bloqueara la ejecución y se lanzara una excepción.

21. ¿Para qué se suele utilizar la excepción PendSV? ¿Cómo se relaciona su uso con el resto de las excepciones? Dé un ejemplo

PendSV se utiliza fundamentalmente para realizar cambios de contexto entre tareas de un sistema operativo.

Durante un proceso de *context switching* entre dos tareas A y B, puede suceder que ocurra una interrupción con mayor prioridad (por ejemplo, systick). En este contexto el sistema operativo no debería realizar el *context switch* ya que el *handler* de la interrupción quedaría bloqueado. Incluso podría producirse una excepción si el sistema operativo intenta pasar a *thread mode* durante la ejecución de una interrupción.

PendSV demora la ejecución del context-switch hasta que todos los IRQ *handlers* hayan terminado su procesamiento. Para ello PendSV es la excepción de menor prioridad.

22. ¿Para qué se suele utilizar la excepción SVC? Explíquelo dentro del marco de un sistema operativo embebido.

La excepción SVC es muy relevante en el campo de los sistemas operativos embebidos ya que permite, en conjunto con la instrucción SVC, implementar una *API* que permita a las aplicaciones acceder a recursos del sistema.

En lugar de proveer acceso privilegiado al *hardware*, las tareas de aplicación pueden ejecutarse en modo no privilegiado y acceder a servicios que requieran permisos elevados a través de servicios del sistema operativo.

Esto es una ventaja también para el desarrollo de tareas de aplicación ya que no se requiere manejar ciertos aspectos del hardware sino conocer la llamada al servicio del sistema operativo.

Guía de preguntas ISA

1. ¿Qué son los sufijos y para qué se los utiliza? Dé un ejemplo.

Los sufijos actúan como modificadores del comportamiento habitual de una instrucción. Se los utiliza para determinar que la instrucción deberá realizar ciertas acciones adicionales u operar de determinada manera.

Por ejemplo, la operación *ldr* admite el sufijo "h". Este le indica que deberá operar con datos de 16 bits (media palabra).

2. ¿Para qué se utiliza el sufijo 's'? Dé un ejemplo.

El sufijo "s" indica que, tras ejecutarse la instrucción, los flags de estado deben ser actualizados. Estos se utilizan para almacenar información de los resultados de la última operación que actualizo los flags.
Por ejemplo, la siguiente porción de código, actualiza los flags mediante la operación subs (nótese el sufijo "s").

```
...
subs   r1, 1
bne    .asm_ejemplo_for
…
```

La instrucción bne, utilizara los flags (en este caso el flag "z") para determinar si la operación dio un resultado distinto de cero. En este caso realizara un "salto" a la etiqueta.

Actualizar los flags, en muchos casos puede no ser deseable. Por ejemplo, si solo se desea incrementar en 1 un registro y posteriormente el código no ejecuta ninguna instrucción que implique una comparación. En este caso, se ejecuta la operación sin sufijo. Por ejemplo:

```
sub r1,1
```

3. ¿Qué utilidad tiene la implementación de instrucciones de aritmética saturada? Dé un ejemplo con operaciones con datos de 8 bits.

La utilización de aritmética saturada impide que se realicen desbordes (overflow) en los resultados de las operaciones.

En caso de observarse un desborde, se ajusta el resultado de la operación al máximo valor admitido por el tipo de datos.

Por ejemplo, si dentro de nuestra función estamos realizando la multiplicación de dos números de 16 bits y luego queremos retornar el resultado a una variable en C de tipo uint8_t, es muy probable que el numero resultante exceda la capacidad de representación de una variable de 8 bits. Las operaciones de aritmética saturada nos permitirán limitar los resultados a la cantidad de bits que se desea utilizar.

La siguiente instrucción asignara un valor del registro r0 al registro r1, estableciendo el límite máximo en la capacidad de representación para datos “unsigned” de 8 bits.

usat r1, 8, r0

4. Describa brevemente la interfaz entre assembler y C ¿Cómo se reciben los argumentos de las funciones? ¿Cómo se devuelve el resultado? ¿Qué registros deben guardarse en la pila antes de ser modificados?

Los registros R0, R1, R2 y R3 son utilizados para pasar parámetros a la función. El resultado se devuelve mediante el registro R0.
Los registros a partir de R4 deben ser cargados de forma explícita en la pila ya que se corre el riesgo de corromper los datos externos a la función.

5. ¿Qué es una instrucción SIMD? ¿En qué se aplican y que ventajas reporta su uso? Dé un ejemplo.

La idea fundamental detrás de la instrucción SIMD es que con una instrucción se puede realizar al mismo tiempo el trabajo que normalmente ser realizaría con múltiples.

SIMD es un acrónimo para Single Instruction Multiple Data. Tal como lo indica su nombre, ejecuta la misma instrucción con múltiples datos al mismo tiempo. Esta capacidad hace a SIMD muy popular en el campo de procesamiento multimedia ya que provee mejoras significativas en la performance.

Un ejemplo de aplicación de instrucciones SIMD es el procesamiento de imágenes, ya que los componentes RGB de cada *pixel* pueden ser procesados en paralelo. En el caso de una imagen, una operación SIMD podría hacer eficiente la aplicación de un “filtro” que la distorsione ya que se podrá operar sobre varios píxeles en forma simultánea.