**Clase 29/04/23**

Hay diversos tipos de micros que se enfocan en algo en particular, eficiencia de consumo, punto flotante, velocidad, etc.

Hoy en día se desarrollan los programas mayoritariamente en C y una pequeña parte puede ser en Assembly.

Cada compilador tiene un target que es la arquitectura en la cuál queremos programar. Por eso en la IDE se elige qué microcontrolador se está utilizando.

Hay instrucciones del set de instrucciones de una arquitectura que tiene punto flotante, y el compilador las utiliza. Esto quiere decir que la arquitectura ya tiene incorporado en el hardware una parte que se dedica a punto flotante y con una sola instrucción se puede realizar la operación.

Pero si no tiene punto flotante, no invocará las instrucciones del procesador, sino que llamará a funciones que implementan el punto flotante con instrucciones generales. Esto demandará muchos más ciclos del reloj. Entonces esto sería la contraparte de lo que se nombró anteriormente, entonces el punto flotante se realiza por software.

El MakeFile es una utilidad que llama al compilador y le indica lo que tiene que hacer. Es la “receta” para compilar un programa.

Cortex M4 es el nombre comercial de ARM v7.

ARM vende el diseño de la arquitectura y actualmente está peleando por el mercado con RISK-V aunque ARM tiene la mayor cuota del mercado porque todos los smartphones lo utilizan.

Cada vez la línea divisoria entre microprocesador y microcontrolador se borra más. Porque las capacidades de ambos se expanden hacía abajo y hacia arriba.

Una diferencia de hace mucho tiempo era que el microcontrolador tenía el microprocesador + memoria + periféricos.

Pero hoy en día la diferencia está en el tiempo real. Generalmente, el microcontrolador debe ser capaz de sensar el entorno, procesar la información y tomar acciones en el tiempo requerido.

Los Cortex A están optimizados para correr muchas aplicaciones a la vez y no en tiempo real. Importa la cantidad de procesamiento que haga el micro y no en el tiempo que responda. Esto se asemejaría a un microprocesador. También saben utilizar un sistema operativo de propósito general. Por ejemplo, un teléfono inteligente.

En un sistema operativo de tiempo real los time slots son más pequeños que los de un sistema operativo de propósito general. En este último se le da más tiempo a cada aplicación: lo que no está bueno para el tiempo real, por ejemplo, en aplicaciones de audio.

Además, los Cortex A tienen memoria Caché y los M no. La Caché es una pequeña memoria, más rápida que se utiliza para ganar tiempo. Estos sistemas que se dedican a las aplicaciones tienen mucha más memoria, pero es más lenta, por eso es necesaria esta memoria un poco más rápida.

Entonces con estos Micros a veces se utiliza RAM y a veces Caché, y aparecen latencias desconocidas. No hay determinismo, porque va variando.

Los Cortex M utilizan a su vez SRAM que corre a la misma velocidad que el procesador y la información está disponible casi instantáneamente.

Cortex R orientado al Real Time. Es el Cortex A, pero sin Caché, por lo tanto es más lento para correr aplicaciones, pero tiene determinismo y se puede utilizar para el tiempo real.

En conclusión, hoy en día los uP y uC no se diferencian por sus periféricos como lo eran antes.

El Ensamblador a veces elige poner una instrucción en 16 o 32 bits según como le parezca más correcto, pero también se puede forzar. A veces utilizamos en 32 bits, pero podría ser más óptimo en 16.

El concepto de interrupción es que uno define un pedazo de código que responde a un evento. Por ejemplo, cambia un bit en alguna parte de la memoria y se ejecuta una rutina de interrupción. Existe una latencia desde que sucede el evento hasta que se ejecuta la rutina. En un Cortex-M esta latencia es de alrededor de 16 ciclos de reloj, la verdad que es muy poco comparada con la velocidad que tiene este micro.

Hay un dicho que dice: el 90% del tiempo se ejecuta el 5% del código, entonces para optimizaciones hay que apuntar en ese 5%, que se puede optimizar utilizando Assembly.

**Clase 06/05/23**

El M4 tiene una arquitectura con mucha capacidad para poder hacer procesamiento digital de señales.

Hay arquitecturas de ARM que son de 16 bits, 32 bits o ambas. Ambas quiere decir que tiene instrucciones de 16 y otras de 32 bits.

ARM 7, 9 y 11 quedó obsoleto y ahora está vigente la serie Cortex.

[en.wikipedia.org/wiki/ARM\_Cortex-M](https://en.wikipedia.org/wiki/ARM_Cortex-M)

La arquitectura nos indica que instrucciones tiene, como se ejecutan, etc.

Muchas instrucciones se pueden codificar en 16 o 32 bits. El micro lo detecta por la codificación de la instrucción al comienzo. El nemónico puede ser el mismo, pero la codificación es distinta. Se le puede especificar en el nemónico con .W o .N, pero hoy en día no hace falta, ya que los compiladores se dan cuenta o lo deciden ellos.

En la guerra de las consolas de los 90s, por ejemplo las consolas de 8 bits, quiere decir que era de 8 bits de bus de datos.

Pero cuando hablamos de arquitectura hablamos del largo de la instrucción.

Cada instrucción va a la memoria FLASH o RAM.

Si hay instrucciones de punto flotante, entonces la arquitectura tiene una unidad de hardware dedicada a eso. Puede ser de simple o doble precisión (float o double) -> mayor cantidad de bits para codificar el número.

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>

Arquitectura de Memoria:

* **Von Neumann:** hay un solo bus de datos que se comunica del uP con los dispositivos externos al uP. En este bus está conectada la memoria SRAM y FLASH. En este bus circulan tanto instrucciones como datos. Lo utilizan micros pequeños y de bajo costo.
* **Harvard:** hay dos buses. Uno para la memoria SRAM (datos) y otra a la FLASH (instrucciones). Lo utilizan micros más grandes y de mayor performance, pero también hay mayor consumo.

Se pasó de Von Neumann a Harvard porque es más rápido.

Cortex es todo de 32 bits, aunque se puedan usar instrucciones de 16 bits. Lo que define el tamaño de la arquitectura, es el tamaño de los registros. En Cortex todos los registros internos son de 32 bits.

Una arquitectura de 32 bits puede representar 2^32 (4.294.967.296) direcciones de memoria diferentes, lo que significa que puede acceder a un máximo de 4 GB de memoria. Si se utilizara una arquitectura de 64 bits, se podría representar 2^64 (18.446.744.073.709.551.616) direcciones de memoria diferentes, lo que permitiría acceder a una cantidad mucho mayor de memoria.

Por lo tanto, la cantidad de bits de la arquitectura puede limitar la cantidad de RAM que se puede utilizar en un sistema. Si un sistema utiliza una arquitectura de 32 bits, no podrá acceder a más de 4 GB de RAM, incluso si se instala más memoria física. Para acceder a más de 4 GB de RAM, se debe utilizar una arquitectura de 64 bits.

Para algunas aplicaciones, se le sabe agregar al micro una SDRAM y una FLASH externas. Internamente tiene otra FLASH y una SRAM.

La FLASH interna generalmente es tan rápida como el micro, y cuando se va a buscar una instrucción, entonces la memoria se la entrega casi sin latencia. La SRAM también entrega datos casi sin latencia. Pero cuando no alcanzan estas memorias, se agregan externas que ya agregan latencias.

La Caché sirve para guardar pedacitos de la SDRAM (RAM externa) para que la tenga inmediatamente. El uP le pidió un dato, luego la SDRAM hace todo el procesamiento y le entrega al micro, pero también le entrega algunos datos consecutivos más para que los guarde en Caché, entonces la próxima vez que el uP haga un pedido, le entregará la Caché. La Caché es parte de la RAM interna. También existen algoritmos que detectan cuáles son los datos más usados para guardar en Caché. Estos micros serían los A en Cortex y no son determinísticos.

Ya Cortex M4 ofrece la posibilidad de tener o no Caché.

TCM es lo segundo más rápido a lo cuál puede acceder el micro, después de los registros. Los registros no se acceden por ningún bus, sino que son directamente registros (hardware) en el mismo núcleo. La TCM no está en el núcleo, pero está muy muy cerquita en el mismo integrado.

Se le puede indicar al compilador que coloque una variable o una función para que se ejecute en el TCM.

Pero la SRAM también es igual de rápida, ¿qué cambia? La SRAM tiene sus primeros 16k aprox que son DTCM e ITCM, pero se acceden de otra manera. En Cortex hay un protocolo en buses de datos que se llama AXI. Como el acceso por la SRAM es a través de un bus, se queda a la suerte del estado de ese bus en ese momento. Por ende, si el bus está ocupado, habrá una latencia. En cambio, TCM está directamente conectado al micro y no depende de un bus y su estado.

El micro tiene una ITCM (instrucciones) y una DTCM (datos). La TCM de instrucciones se utiliza para almacenar instrucciones que se ejecutan con frecuencia, mientras que la TCM de datos se utiliza para almacenar datos que se acceden con frecuencia. Ambas son memorias de acceso rápido y bajo consumo de energía, lo que significa que pueden accederse con mayor rapidez que la memoria RAM normal y sin necesidad de acceder a la memoria flash o EEPROM.

Los dispositivos externos (periféricos) no están conectados directamente al micro, sino que están todos conectados a un bus común. Este bus puede tener retardos y puede darse que hayan esperas/latencias.

MPU (memory protection unit): bloquear una parte de la memoria para que no pueda ser cambiada, leída, etc. por el usuario. Tiene que ver con los modos de ejecución también: modo privilegiado o no privilegiado. El SO es el que tiene generalmente el mayor privilegio porque puede acceder a toda la memoria, al hardware, etc.

La memoria se divide en partes para distintas funciones. El mapa de memoria muestra todo esto. Luego queda definir un puntero, y ahí se podrá acceder a todo. Cada periférico tendrá su dirección de memoria y por lo tanto se podrá configurar.

SysTick Timer lo define ARM. Es estándar y todos los fabricantes deben respetarlo, entonces esto mejora la portabilidad. Esto pasa también con la MPU.

El systick es un temporizador integrado en varios microcontroladores, incluidos los de la familia ARM Cortex-M. Su función principal es proporcionar una interrupción periódica al procesador, lo que permite al sistema operativo o al firmware realizar tareas en intervalos regulares. Es importante para sistemas operativos.

Algunos M0 no tienen SysTick y no es bueno utilizar un RTOS, porque se debería utilizar otro timer y puede traer muchos problemas.

Para saber si un bit es un 1 o 0 o extraer datos en memoria se utilizan máscaras.

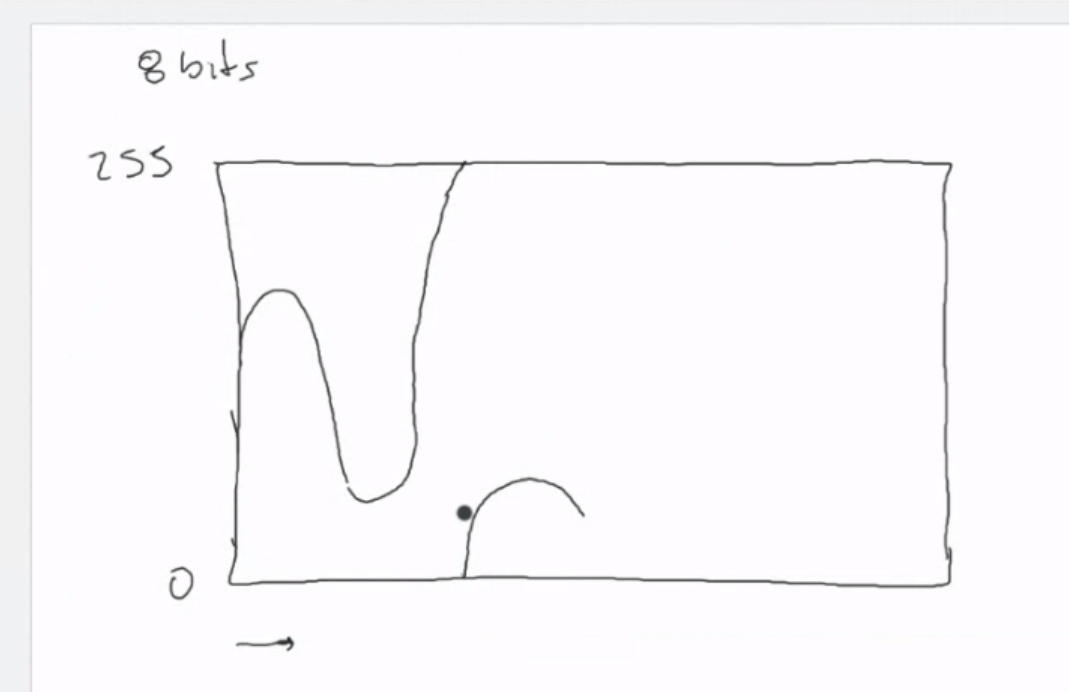
**Clase 13/05/23**

Pregunta: ¿Cómo es el mapa de memoria de la familia Cortex?

La memoria es toda continua y los periféricos se mapean en ella. Esto no es así en todas las arquitecturas. El hecho de acceder a los periféricos, configuración, dispositivos externos es algo destacable de la familia Cortex.

La arquitectura v7 incluye soporte completo para Thumb2. También agrega punto flotante y extensiones de DSP.

Saturated math se utiliza para que cuando se da overflow o underflow los números no comiencen desde el otro extremo. De esta forma queda el número saturado en máximo y mínimo. Es muy útil para DSP. En un ADC me puedo ir del rango y podrá pasar para el otro lado:



En sonido podríamos pasar de volumen bajo a alto y romper el buffer. En video un píxel podría pasar de blanco a negro.

Esto se podría hacer por software. Pero no está bueno utilizar condicionales, porque no ayuda a la performance. En el procesador existe algún lugar que indica que se produjo un overflow, se podría estar consultando todo el tiempo y eso también perjudica a la performance.

Con aritmética saturada, los valores no pasan de un extremo al otro, quedan saturados en el máximo o en el mínimo

Si dos números de 32 bits de máximo valor se multiplican, entonces el resultado no entrará en 32 bits. Tener cuidado con eso. Siempre buscar multiplicar números de 16 bits para que no se produzca overflow.

RISC (Reduced Instruction Set Computers) apunta a hacer instrucciones específicas y atómicas que hacen una cosa en particular. CISC apunta a hacer instrucciones más simples. **CHECKEAR**

Datapath es el bus interno del microcontrolador que une todos los componentes.

RISCV hace lo mismo que ARM, hace todo el diseño de la arquitectura. Pero la diferencia es que RISCV es gratis y ARM cobra licencias.

Los fabricantes chinos están empezando a utilizar mucho RISCV para no tener que pagar licencias a ARM.

Pipeline es muy importante que sea de 3 etapas porque mejora mucho la performance. Cortex tiene diferentes pipelines de diferentes formas y tamaños.

Cada instrucción pasa por tres etapas (específico de M4 y habría que ver cómo es con los otros): fetch, decode y execute. Se multiplica por 3 la velocidad del procesador. Estas etapas dependen del diseño del micro y cuáles sean los objetivos de este. En M4 se busca que sea más rápido.

Una ventaja de los procesadores RISC, las instrucciones se ejecutan en un ciclo de reloj. En CISC generalmente duran más ciclos de reloj, 5 o 10.

El Program Counter (PC) es uno de los 16 registros, indica la dirección de memoria de la próxima instrucción.

Escribir en el PC es hacer un salto incondicional, como un “goto”.

* Fetch: el micro va a donde está el PC y toma la instrucción.
* Decode: se descompone la instrucción en su configuración, prefijos, operandos, etc.
* Execute, se ejecuta la instrucción. Ver que al procesador le toma 3 ciclos de reloj ejecutar una instrucción. Pero después cuando el pipeline se llena, se empieza a ejecutar una instrucción por ciclo de reloj.

En la dirección 0 de la RAM está el MSP (Main Stack Pointer).

Fetch hace siempre PC + 1 antes de tomar una instrucción. Por lo tanto, nunca se empieza por la dirección 0.

En un llamado a función lo que hago es un salto incondicional a una posición de memoria donde está la función. Antes de ir a la función hay que guardar los registros que estaba utilizando el micro antes de saltar. Hay que guardar esto en el stack, haciendo un cambio de contexto.

El stack es un tipo de memoria (no registros) que permite guardar cosas y recuperar cosas.

El programa no empieza en el main sino en la rutina de inicialización o reset.

Arquitectura load/store, cuando los operandos siempre son los registros o valores inmediatos (en la misma instrucción). El set de instrucciones no puede operar con datos en memoria. Además, hay instrucciones para mover datos entre memoria y registros.

Los micros se pueden poner en modo bajo consumo y se puede despertar por una interrupción con un evento (cambio en un pin) o por un timer.

MPU (memory protection unit): lo implementa el Cortex y es para proteger la memoria. Con esto puedo configurar para que de la dirección tal a otra esté protegida. Cortex tiene 2 modos de privilegios (modo privilegiado: tocar toda la memoria, hacer todo lo que uno quiere, modo no privilegiado: tengo restricciones, no puedo acceder a los registros de configuración y habrá áreas de memoria protegidas que no voy a poder acceder).

Aplicación en un RTOS: Kernel en modo privilegiado y tareas de usuario en no privilegiado. Aunque FreeRTOS no tiene activado el modo privilegiado y no privilegiado, porque es muy genérico.

La memoria del SO podría estar protegida.

Cuando la tarea en modo no privilegiado quiere acceder a un espacio de memoria protegido, se produce una excepción que la maneja el SO.

En un SO cada tarea tiene su propio stack.

MSP es el Stack Pointer general del programa y el PSP es el Stack Pointer que utiliza la tarea. Ambos son registros.

Recordar que el SP es un registro que guarda la dirección de memoria del stack. Apenas inicia tendrá la dirección inicial del Stack.

En modo privilegiado el SP es el MSP puede acceder a todo, en modo no privilegiado el SP es el PSP y queda acotado a lo que puede acceder.

Cada variable global tiene una área en memoria siempre asignada para ella. Antes de que se compile el programa, estas variables tienen direcciones genéricas. Cuando se compila, el Linker Script asigna direcciones reales a cada una de las variables. Es memoria estática, está asignada antes de ejecutarse el programa. Cada variable tiene su espacio de memoria.

El Heap es una memoria dinámica que el programa puede utilizar. (Recordar memoria dinámica, función malloc). Se ubica apenas termina la memoria estática. Hay una variable que define cuánta memoria se le asigna al Heap o hasta que dirección de memoria queremos que se reserve. Malloc no usa toda la memoria que quiere, en un SO le pide memoria a éste primero. El SO es el que gestiona los recursos del micro (tanto el tiempo como la memoria). Es por esto que existen los modos de privilegio y la protección de memoria.

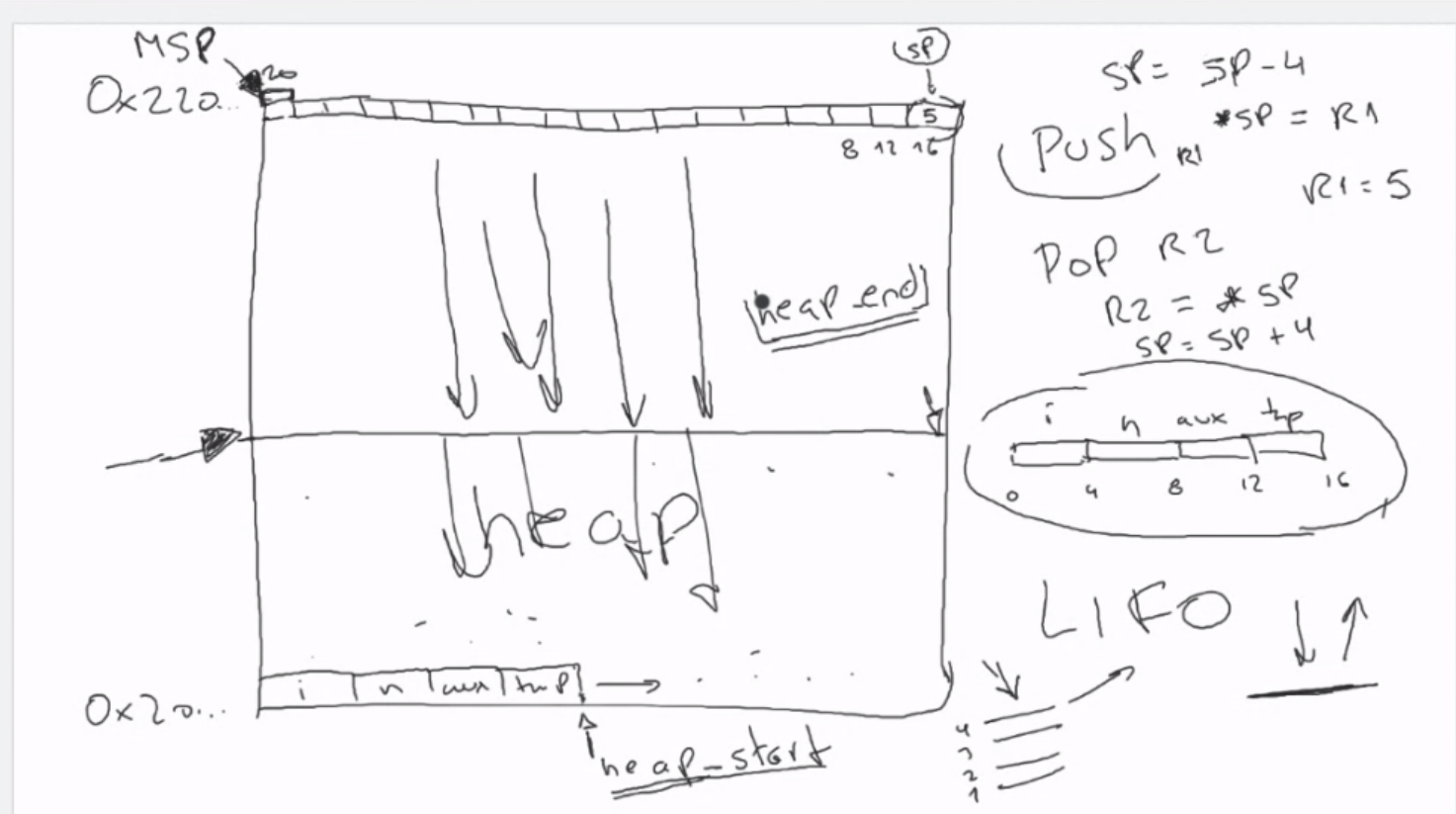
Luego viene el Stack para poder guardar cambios de contexto. Pop y Push son las instrucciones para guardar en y sacar cosas del Stack.

Los Cortex tienen una arquitectura de Stack “full descending”. La dirección inicial del MSP es el “final” del Stack o la dirección más alta del Stack.

El Stack siempre guarda de a 32 bits (tamaño de los registros). Y es una estructura de datos LIFO (pila).

Una función recursiva mal diseñada puede aumentar rápidamente el stack y no hacer nunca más pop.

Si se está en modo privilegiado y se desborda el Stack, se va a empezar a pisar datos del Heap.



Además, el stack va a estar usando espacio más allá de lo que pertenece. Puede pasar a romper las variables globales también. El Heap a su vez va a pisar información del Stack y las funciones devolverán datos erróneos.

Es muy importante calcular el tamaño de Stack que se va a necesitar. Muchos fallos se pueden dar por este problema.

Es importante tenerlo en cuenta para sistemas críticos. Primero para que no pase y segundo que si pasa, que la falla sea en modo seguro. Recordar que los sistemas críticos son aquellos de los cuáles dependen vidas humanas.

**Clase 20/05/2023**

CMSIS es una capa de abstracción de software para poder utilizar la IP de ARM entre fabricantes y entre los modelos de la familia Cortex.

Con el tiempo se fue agregando CMSIS-DSP, que permite funciones iguales en todos los micros. La velocidad en cada micro, dependerá de él, pero lo implementaría de la misma manera.

Utilizando CMSIS-RTOS es que se puede usar RTOS en cualquier micro sin cambiar el código, pero solo sirve en Cortex.

EL CMSIS puede manejar todo el hardware de abajo (en azul, ver filmina).

También CMSIS-SVD tiene en que posición de memoria está que periférico. Para cada fabricante es diferente, entonces el fabricante da a los desarrolladores este archivo.

FAT-FS es un middleware permite manejar los datos de las memorias SD, RAM, etc. tomándolo como si fuese un sistema de archivos FAT. Se utiliza bastante para leer y grabar archivos en una SD, pendrive, etc.

Los saltos o bifurcaciones interrumpen el pipeline, por eso existe el “speculated branch” que es una función estadística que especula a donde se hará el salto para no interrumpirlo.

El M7 tiene algo ya de Caché pero pierde determinismo.

SP se puede apuntar a MSP o PSP (que están en memoria) según un bit en un registro de control. Sólo una a la vez.

PC tiene la dirección de memoria de la próxima instrucción a ejecutar.

El ejecutable .bin es un archivo binario con las instrucciones del programa, esto se carga en el micro, se flashea o se pone en la memoria flash.

En un sistema embebido, la posición 0 tiene el SP inicial que a su vez es el valor de MSP. Luego sigue la dirección en la flash de la función reset. Cuando se llama la función reset, esta configurará los periféricos y llamará a main(). Seguramente después de main() no haya nada, ninguna instrucción. Va a ver todo 0, y generará una excepción, seguramente el programa termina.

Lo que sigue después de la posición 0, es una dirección de memoria de la instrucción a donde debe saltar (la función de reset), pero no es una función branch. Este comportamiento está hardcodeado en la arquitectura.

Luego, de la posición 0 se guarda el vector de interrupciones, que es un arreglo con punteros a funciones. En la posición 1 estará la función de reset y así sucesivamente. Lo primero que tiene la Flash es el vector de interrupciones. Además, las interrupciones se ordenan por prioridad, siendo más prioritarias las primeras. Esto pasa en muchas arquitecturas. En Cortex se tiene la posibilidad de configurar esto.

Las prioridades en las interrupciones sirven para determinar cuál atender en caso de que dos ocurran.

En el vector de interrupciones, no se tiene la interrupción en sí, sino la dirección de memoria de la función de la rutina que atiende dicha interrupción.

Bit menos significativo en 1 es instrucción Thumb. Si es 0 es de 32 bits.

Mejor dicho, si en el vector de interrupciones se tiene una dirección de memoria que termina en 1, significa que la instrucción en dicha dirección es Thumb. Ver filmina 43. La instrucción de la dirección 0x00..0004 salta a 0x00..0101, quiere decir que salta a 0x00..0100, pero que allí hay una instrucción Thumb.

Si tengo una instrucción Thumb, el PC tiene que aumentar en 2 y no en 4 bytes.

En las instrucciones Thumb, se actualizan las banderas por defecto. El PSR = Program Status Register tiene las banderas del programa y otras cosas como indicar si se usan instrucciones Thumb o no, entre otros.

La línea Cortex define una serie de excepciones. ARM define las primeras 15. Siempre la primera es el reset y la que tiene mayor prioridad.

Las excepciones son eventos que no tienen que ver con la ejecución del programa. Tiene que ver con cambios de estado en un pin, alertas de tensión baja, etc. Estos pueden suceder en cualquier momento. Y cuando suceden, alguna parte del código se tiene que encargar de manejar eso.

Usage fault, es una instrucción inválida. Cuando el procesador hace fetch de una instrucción, pero es una combinación de bits que no entiende. Se produce entonces una excepción. Entonces la NVIC toma control del micro y le dice que vaya a buscar en el vector de interrupciones la dirección que maneje esa excepción. El PC toma esa función y la ejecuta.

Las excepciones son una generalización de las interrupciones clásicas. Pero no todo se da porque cambia el estado en un pin. A veces también hay errores que deben ser manejados. Pueden haber varios motivos, por software también.

Un RTOS brinda al usuario servicios para periféricos, memoria, etc.

Todo lo que se ejecuta desde el vector de interrupciones, se hace en modo privilegiado.

Las excepciones a partir de la 16, las define el fabricante. Generalmente cambian de nombre dependiendo del fabricante.

En Cortex se puede poner máscaras para no escuchar a ciertas excepciones. El NVIC no va a reaccionar cuando suceda una excepción. Las 3 primeras no se pueden enmascarar.

El micro tiene mecanismos con los que detecta que la tensión de alimentación es peligrosamente baja y con ello puede generar una excepción. Esta sería importante y no se puede enmascarar para ignorar.

Las excepciones pueden suceder en cualquier momento y en cualquier parte de un programa.

Existe una latencia cada vez que se produce un evento, por todo el proceso que sucede.

Muchas veces hay aplicaciones donde se necesita un tiempo de respuesta que tiene que ser menor a un cierto valor, en base a eso se determina el micro a utilizar.

Una llamada a función es cambiar el PC, pero cuando salgo de la función, ¿a dónde vuelvo? Para eso se utiliza LR que guarda la dirección de memoria de la instrucción que sigue después de la función. Para eso se utiliza BL que guarda esta dirección automáticamente.

El registro de control tiene el bit 1 para elegir si SP apunta a MSP o PSP. El bit 0 es para definir el modo: privilegiado o no privilegiado.

Ortogonal quiere decir que tocar 1 no modifica la actividad del otro? No dependen uno del otro. Por ejemplo, el seteo de una opción no modifica a otras?

Ortogonalidad del set de instrucciones. Hay muchas instrucciones, pero en general todas funcionan tomando como operando cualquier registro. Cualquier instrucción opera con cualquier registro.

Antes en otras arquitecturas viejas, había registros con nombres específicos y que se podían solo utilizar para operaciones/situaciones específicas.

Si quiero pasar de no privilegiado a privilegiado, el micro lo ignora. La forma de hacerlo es pedírselo al SO con el llamado de un manejador de excepción. No lo podré hacer cambiando el bit 0 en el registro de control. Ver filmina 20.

El micro tiene dos modos de operación: Thread (durante código de usuario = privilegiado o no privilegiado) o Handler (durante manejador de excepciones = siempre privilegiado).

La instrucción svc en assembly lo que hace es llamar a una excepción.

Los parámetros a funciones se pasan a los registros más bajos en orden. R0 contiene siempre el valor de retorno de la función.

**Clase 27/05/23**

Ver el siguiente ejemplo:

**for(uint8\_t i = longitud - 1; i >= 0; i- -) {**

**/\* Bloque \*/**

**}**

Como **i** es una variable sin signo, nunca será negativa, pasará de 0 a 263 y el búcle nunca terminará.

Mientras más variables se usan, más registros se usan. Por lo tanto, aumenta el procesamiento. ¿Por qué no hacerlo de esta manera? Se ahorra una variable = se ahorra un registro = se ahorra tiempo.

**for(; longitud > 0; i- -) {**

**/\* Bloque \*/**

**}**

Por convención los parámetros se envían en R0, R1, R2 y R3. El valor de retorno siempre se hace en R0.

Hacer decremento en vez de incremento en un bucle, se ahorra una instrucción CMP.

Hay registros de control como el PSR, para guardar el estado del micro.

Pasar de modo privilegiado a no privilegiado se puede hacer fácilmente, pero no al revés.

**Add R0, R1, R2 @R0 = R1 + R2**

**Add R0, R1 @R0 = R0 + R1**

Cada vez que hace un llamado a función, lo que se hace es un salto o bifurcación. El PC cambia a donde está la función definida.

Cuando se llama la función se hace un BL (branch with link), se hace el salto y se actualiza el registro LR (link register).

**BX LR @= MOV PC, LR**

Puede ser que el compilador esté usando R0, R1, R2 y R3, entonces es necesario que en el cambio de contexto se guarden en el stack. Cuando se vuelve de la función, se hace un POP y se reestablecerán los valores anteriores.

Si se utiliza de R4 para arriba, el compilador no guarda estos registros en el stack, yo lo tengo que hacer.

Tener cuidado que los registros pueden tener basura.

Cuando uno escribe en Assembly, no escribe programas enteros, sino optimiza funciones específicas.

Se comprobó estadísticamente que como mucho las funciones toman 4 parámetros, es por eso que los parámetros van de R0 a R3. Si hay un 5° (caso raro) se utilizará el stack.

Tener cuidado con el stack cuando se utiliza RTOS, porque una tarea la puede ocupar toda.

Corrimiento a la izquierda 2 bits = multiplicar x 4.

Corrimiento a la derecha 2 bits = dividir x 4.

NOP: instrucción que no hace nada (No OPeration).

Instrucción IT = if true, toma como mínimo una condición verdadera.

Las etiquetas representan la dirección de memoria de la siguiente instrucción de donde está puesta la etiqueta.

**Clase 03/06/23**

Se puede medir cuánto demora la ejecución de un programa, llamando a funciones que dan el tiempo (o cantidad de ciclos).

Heap y stack = memoria dinámica.

Data = variables **globales** estáticas.

Si una variable es local de una función, se creará en tiempo de ejecución.

Siempre que se pida memoria dinámica, hay que devolverla en algún momento. Este es un bug muy común y se llama **memory leak.** Pasa con el heap.

Pasarse con el stack se llama **stack overflow.**

En las arquitecturas más nuevas, ocurre una excepción cuando se llega al límite en el stack.

Se recomienda no usar memoria dinámica en sistemas embebidos.

Se puede codear Assembly inline (junto a código de C). Pero en general se utilizan archivos de Assembly separados (“.S”).

Generalmente el inline se utiliza para cosas muy específicas.

**Window -> Show view -> Memory -> Add Memory Monitor** (se agrega una memoria inicial, para no mostrar toda la memoria).

Cortex solo permite instrucciones Thumb (Thumb1 = 16 bits y Thumb2 = 32 bits).

**Es importante chequear que el arreglo sea mayor a 0!**

**Es importante chequear que el puntero al arreglo no sea NULL!**

Es mucho más rápido hacer corrimientos lógicos a la derecha o izquierda, en vez de utilizar instrucciones específicas de multiplicación o división.

Los códigos de filtros digitales se suelen escribir en Assembly porque es más eficiente.

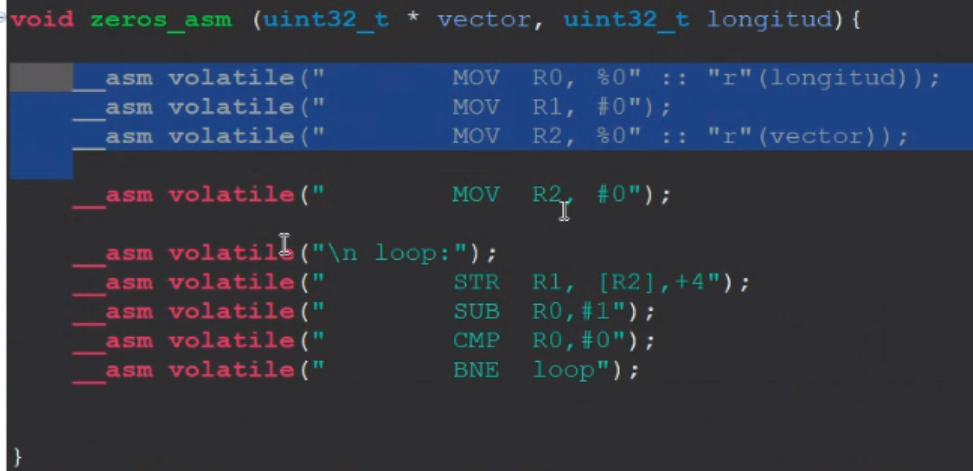
En C, para escribir Assembly inline:

**\_\_asm volatile(“ /\* assembly instruction 1 \*/ “);**

**\_\_asm volatile(“ /\* assembly instruction 2 \*/ “);**

**\_\_asm volatile(“ /\* assembly instruction 3 \*/ “);**

**\_\_asm volatile(“ MOV R0, %0“ :: “r”(variable));** // se está re-asignando los parámetros a los registros. Esta es una desventaja en eficiencia.



Los intrinsics nos permiten llamar funciones de Assembly en C.



Por ejemplo, estaría muy bueno poder utilizar USAT en C.

La forma de saber si un código es más rápido que otro es mediante un Timer. El microcontrolador tiene un timer para contar ciclos de reloj. Esto es muy común en los micros modernos.

El filtro de ventana móvil funciona así: por cada iteración, se tomará una determinada cantidad de elementos antes y después del elemento sobre el cual estoy parado (índice actual). Se toman todos los elementos (incluido el actual) y se puede hacer por ejemplo un promedio.

¿Qué pasa cuando la ventana cae fuera del arreglo? A veces se toman los datos como una cola circular (se comienza desde el otro extremo), a veces se pone 0, a veces se copia el último o primer valor.

Se acuerda para el ejercicio colocar 0s. La ventana deberá ser 5 elementos hacia un lado y 5 hacia el otro desde el índice en cuestión. Luego, se hace un promedio.

Luego para el próximo ejercicio, en la de empaquetar un arreglo de 32 bits en otro de 16 bits se debe utilizar la parte más significativa de los 32 bits porque es lo que más se acercará al número original.

**Clase 10/06/23**

Archivo .S (mayúscula) primero el ensamblador los pasa por el preprocesador de C, esto permite incluir un archivo .h y utilizar las constantes definidas ahí. Para el archivo .s (minúscula) eso no pasa.

Hay instrucciones específicas para punto flotante. Se utilizan 64 bits para manejar punto flotante.

Las configuraciones del micro en bajo nivel se hace modificando bits en los registros de configuración.

El oscilador interno RC sirve, pero es muy impreciso porque no se está utilizando un cristal. Es una alternativa de bajo costo.

M4 está orientado para el procesamiento digital de señales.

**Clase 17/06/23**

GCC utiliza el llamado a funciones inline para optimización.

Se puede optimizar evitando los bucle for que rompen el pipeline, se evitan saltos y comparaciones.

GCC tiene niveles de optimización. Por defecto está en modo Debug. Se busca ser literal con el código hecho en C. Esto es para que se pueda hacer el paso a paso.

Si queremos performance, saca el compilador en modo Debug.

Project - Properties - C/C++ Build - Settings - Configuration: Debug/Release

En Debugging se puede elegir el nivel para indicar que cantidad de información incluir en el ejecutable (símbolos, etc. para el programador).

En MCU GCC Compiler - Optimization se puede elegir el nivel de optimización.

Optimización máxima -O3 puede traer problemas. Probar y si algo anda mal bajarle el nivel de optimización.

En modo Release se eliminan los símbolos: nombres de variables, de funciones (lo que necesita el ser humano). No se pierde tiempo en eso. Hay una optimización.

En modo Release, no se hace un Debugging, sino que se hacen pruebas funcionales para ver si hace lo que tiene que hacer -> resultados.

A veces cuando se quiere hacer un Debugging de un programa optimizado, no podemos ver las variables porque no las vamos a encontrar. Por ejemplo, la variable que tiene las cuentas de ciclos no va a aparecer. Se puede entonces mandar el valor por consola.

**Optimizar para velocidad** -> para que sea más rápido.

**Optimizar para tamaño** -> reducir la cantidad de instrucciones -> optimizar para que se reduzca la cantidad de FLASH utilizada.

En modo Debug respeta todo, en modo Release hace lo que le parece con tal de cumplir el objetivo de optimización.

Cuanto más se optimiza, más bugs aparecen.

No basta con optimizar en Aseembly. El compilador en optimización máxima (utilizando solo C) le va a ganar al Aseembly. Es necesario utilizar SIMD para sacar ventaja. Es necesario conocer la arquitectura para aprovecharlo al máximo.

Siempre optimizando loops, es donde se va a ver mayor impacto. La cantidad de instrucciones multiplicado por la cantidad de iteraciones da la cantidad total de ciclos. Sacando 2 o 3 instrucciones se puede ver una mejora muy grande.

En Assembly, además, tenemos total control del tiempo del procesador y todo lo que ejecuta. En cambio, no se sabe lo que hizo el compilador en C.

Los manejadores de excepciones no toman parámetros. En realidad no son funciones, son manejadores. Por lo tanto, no se respeta enviar parámetros en R0, R1, R2 y R3. Por lo tanto, tampoco se guardó el LR (no es llamado a función). NVIC utiliza el LR para otra cosa. La dirección de retorno se guarda en el STACK.

LR tiene un valor codificado y tiene un valor particular. Indica no a dónde se debe volver, sino a cómo se vuelve, a que estado, en qué modo por ejemplo: privilegiado o no privilegiado. Ver en filmina 66, la codificación.

Si se está ejecutando el manejador de una excepción y aparece otra de mayor prioridad (recordar que las tres primeras no son configurables, las demás sí) se pasa a ejecutar la de mayor prioridad. Si aparece una de menor prioridad, se va a seguir ejecutando la actual hasta el final y después se sigue con la de menor prioridad. Siempre se mantiene en modo Handler y no pasa a modo Thread en el medio. Por eso en el LR, el bit 3 indica el modo de retorno.

En FreeRTOS no se usa el PCP y el MCP.

Cuando se busca hacer el cambio de contexto, se hace lo que se llama “Stacking”, donde se guardan en el Stack todos los registros. Esto es cuando se pasa de Thread a Handler.

Fetch es obtener las instrucciones de memoria.

Todos los registros que empiezan con “S” son de la FPU.

Si en el medio del Stacking aparece una excepción de mayor prioridad, si no llegó todavía al Vector Fetch, se pasa al de mayor prioridad. Esto se llama “Late arrival”.

PendSV es una excepción que tiene la prioridad más baja. Es ahí donde se produce el context switching. Generalmente el pedido del context switching ocurre en el systick. Como PendSv tiene prioridad baja, solo va a comenzar sólo cuando termine de ejecutarse lo otro. Entonces esto garantiza que el context switching no se produzca durante el systick o en el medio de una interrupción importante.

La prioridad 0 es la más alta. La subprioridad sirve por ejemplo cuando se tiene un “Late arrival” para ver si una excepción de misma prioridad que llegó se puede poner por encima o no.