Features of the Cortex M0/M3/M4 processor

* Operational mode of the processor
* the difference access levels of the processor
* the register set of the processor
* the banked stack design of the processor
* exceptions and exception handling
* interrupt handling
* bus interface and bus matrix
* memory architecture of the processor, bit banding, memory
* endianness
* aligned and unaligned data transfer
* bootloader and IAP

Operational mode of the processor

* Processor gives 2 operational modes
  + Thread mode
  + Handler mode
* All your application code will execute under "Thread mode" of the processor. This is

also called as "User mode"

* All the exception handlers or interrupt handlers will run under the "handler mode" of the

processor

* Processor always starts with "Thread mode"
* Whenever the core meets the system exception or any external interrupts then the core will

change its mode to handler mode to service the ISR associated with that system exception or the interrupt

Access level of the processor

* Processor offers 2 access levels
  + Privileged Access Level (PAL)
  + Non-privileged Access Level (NPAL)
* If your code is running with PAL, then your code has full access to all the processor specific resources and restricted registers
* If your code is running with NPAL, then your code may not have access to some of the restricted registers of the processor
* By default, your code will run in PAL
* When the processor is in "Thread mode", it’s possible to move processor into NPAL. Once you move out of the PAL to NPAL being in thread mode, then it’s not possible to come back to PAL unless you change the processor operational mode to "Handler Mode"
* "Handler Mode" code execution is always with PAL
* Use the CONTROL register of the processor if you want to switch between the access level

Cortex M Processor Core Registers

* R0 - R12 (total 13) registers are for general purpose
* All the core registers are 32 bits wide
* The register R13 is called as SP
* The register R14 is called LR and used to hold the return information during function call and exception handling
* The register R15 is called PC which holds the address of the next instruction to be executed
* Note:
  + bx: branch indirect
  + bl: branch with link

Memory mapped and non-memory mapped register of the MCU:

* Non memory mapped register:
  + The register does not have unique address to access them. Hence, they are not part of the processor memory map
  + You cannot access these registers in a ‘C’ program using address dereferencing
  + To access these registers, you must use assembly instructions
  + For instance, processor core registers
* Memory mapped register:
  + Register of the processor specific peripherals (NVICM, MPU, SCB, DEBUG, etc)
  + Registers of the microcontroller specific peripherals (RTC, I2C, TIMER, CAN, USB, etc)
  + Every register has its address in the processor memory map
  + You can access these registers in a ‘C’ program using address dereferencing

ARM GCC inline assembly code usage

* Inline assembly code is used to write pure assembly code inside a ‘C’ program
* GCC inline assembly code syntax as shown below
  + \_\_asm volatile(“MOV RO, R1”);
* \_\_asm volatile(code:output operand list:input operand list: clobber list);
  + Volatile attribute to the asm statement to instruct the compiler not to optimize the assembler code
  + A list of output/input operands separated by commas
  + Clobber list is mainly used to tell the compiler about modifications done by the assembler code

Input/output operands and Constraint string

* Each input and output operand is described by a constraint string followed by a c expression in parentheses
  + Format: “<Constraint string>” (<’C’ expression>)
  + Constraint string = constraint modifier + constraint character

Reset sequence of the Cortex M Processor

* When you reset the processor, the PC is loaded with the value 0x0000\_0000
* The processor reads the value at memory location 0x0000\_0000 into MSP
  + MSP = value at 0x0000\_0000
  + MSP is a Main Stack Pointer register, that means, processor first initialize the Stack Pointer
* After that processor reads the value at memory location 0x0000\_0004 into PC
  + That value is the address of the reset handler
* PC jumps to the reset handler
* A reset handler is just a C or assembly function written by you to carry out any initializations required
* From reset handler you call your main() function of the application

Discussion about T bit of the EPSR

* Various ARM processors support ARM-Thumb interworking, that means the ability to switch between ARM and Thumb state
* The processor must be in ARM state to execute instructions which are from ARM ISA (instruction set architecture) and the processor must be in Thumb state to execute instructions of Thumb ISA.
* If ‘T’ bit of the EPSR is set (1), processor thinks that the next instruction which it is about to execute is from Thumb ISA
* If ‘T’ bit of the EPSR is set (0), processor thinks that the next instruction which it is about to execute is from ARM ISA
* The cortex Mx processor does not support the “ARM” state. Hence, the value of ‘T’ bit must always be 1. Failing to maintain this is illegal and this will result in the “Usage fault” exception
* The lsb (bit 0) of the program counter (PC) is linked to this ‘T’ bit. When you load a value or an address into PC the bit [0] of the value is loaded into the T-bit
* Hence, any address you place in the PC must have its 0th bit as 1. This is usually taken care by the compiler and programmers need not to worry most of the time.
* This is the reason why you see all vector addresses are incremented by 1 in the vector table

Stack memory

* Stack memory is part of the main memory (internal RAM or external RAM) reserved for the temporary storage of data (transient data)
* Mainly used during function, interrupt/exception handling
* Stack memory is used in last in first out fashion (LIFO)
* The stack can be accessed using PUSH and POP instructions or using any memory manipulation instructions (LD, STR)
* The stack is traced using a stack pointer register (SP), PUSH and POP (increment or decrement) instructions affect stack pointer register (SP, R13)

Stack memory uses

* The temporary storage of processor registers values
* The temporary storage of local variables of the function
* During system exception or interrupt, stack memory will be used to save the context (some general-purpose register, processor status register, return address) of the currently executing code

Stack operation model

* In ARM Cortex Mx processor stack consumption model is Full Descending (FD)

Different stack operation model

* Full Ascending Stack (FA)
* Full Descending Stack (FD)
* Empty Ascending Stack (EA)
* Empty Descending Stack (ED)

Banked stack pointer

* Cortex Mx Processor physically has 3 stack pointers, SP(R13), MSP and PSP. MSP: main stack pointer, PSP: process stack pointer. SP is called as current stack pointer
* After processor reset, by default, MSP will be selected as current stack pointer. That means, SP copies the value of MSP
* Thread mode can change the current stack pointer to PSP by configuring the CONTROL register’s SPSEL bit
* Handler mode execution will always use MSP as the current stack pointer. That also means that, changing the value of SPSEL bit being in the handler mode does not make sense. The write will be ignored
* MSP will be initialized automatically by the processor after reset by reading the content of the address 0x00000000
* If you want to use the PSP then make sure that you initialize the PSP to valid stack address in your code

Cortex M processor exception model

* What is exception?
  + Anything which disturb the normal operation of the program by changing the operation mode of the processor
* There are two types of exception
  + System exception
  + Interrupts
* What is system exception
  + Generated by the processor itself internally
* Interrupt coms from the external world to the processor
* Whenever the processor core meets with an exception, it changes the operation mode to handler mode
* There are in total 15 system exceptions supported by the Cortex Mx processors, and 240 interrupts. So, in total Cortex Mx processors support 255 exceptions

ARM Cortex Mx: Different System Exceptions

* There is room for 15 system exceptions
* Exception number one is Reset Exception (or system exception)
* Only 9 implemented system exceptions, 6 are reserved for future implementation
* Exception number 16 is interrupt 1 (IRQ 1)

NVIC (Nested Vector Interrupt Controller) discussion

* NVIC is one of the peripheral of the Cortex M processor core
* It is used to configure the 240 interrupts
* Using NVIC register you can enable/disable/pend various interrupts and read the status of the active and pending interrupts
* You can configure the priority and priority grouping of various interrupts
* It is called as nested because, it supports pre-empting a lower priority interrupt handler when higher priority interrupt arrives

ARM Cortex M3/M4 Fault Handlers

* Faults happen because of programmers handling processor by violating the design rules oy may be due to interfaces with which the processor deals
* Whenever a fault happens, internal processor registers will be updated to record the type of fault, the address of instruction at which the fault happened, and if an associated exception is enabled, the exception handler will be called by the processor
* In the exception handler programmers may implement the code to report, resolve, or recover from the fault
* For example, if your code tries to divide a number by zero, then divide by 0 fault will be raised from the hardware, which will invoke usage fault exception handler (if enabled). In the exception handler, you may make certain decisions to get rid of the problem, like closing the task, etc.
* Most of the time, fault happens by programmer’s code not adhering to processor programming guidelines.

Cause of Fault

* Divide by zero (if enabled)
* Undefined instruction
* Attempt to execute code from memory region which is marked as execute never (XN) to prevent code injection
* MPU guarded memory region access violation by the code
* Unaligned data access (if enabled)
* Returning to thread mode keeping active interrupt alive
* Bus error (Example, no response from memory device)
* Executing SVC instruction inside SVC handler or calling a function in SVC handler which eventually execute hidden SVC instruction
* Debug monitor settings and related exceptions

Hard fault Exception

* A hard fault is an exception that occurs because of an error during exception procession, or because an exception cannot be managed by any other exception mechanism. It has 3rd highest fixed priority (-1) after reset and NMI meaning it has higher priority than any exception with configurable priority
* Causes
  + Escalation of configurable fault exceptions
  + Bus error returned during a vector fetch
  + Execution of break point instruction when both halt mode and debug monitor is disabled
  + Executing SVC instruction inside SVC handler

Mem Manage Fault exception

* This is a configurable fault exception. Disabled by default
* You can enable this exception by configuring the processor register “System Hnadler Control and State Register (SHCSR)”
* When mem manage fault happens, mem manage fault exception handler will be executed by the processor
* Priority of this fault exception is configurable
* Causes
  + As its name indicates, this fault exception triggers when memory access violation is detected (access permission by the processor or MPU)
  + Unprivileged thread mode code (such as user application or RTOS task) tries to access a memory region which is marked as “privileged access only” by the MPU
  + Writing to memory regions which are marked as read-only by the MPU
  + This fault can also be triggered when trying to execute program code from “peripheral” memory regions. Peripheral memory regions are marked as XN (execute never) regions by the process design to avoid code injection attacks through peripherals

Bus Fault exception

* Due to error response returned by the processor bus interfaces during access to memory devices
  + During instruction fetch
  + During data read or write to memory devices
* If bus error happens during vector fetch, it will be escalated to a hard fault even if bus fault exception is enabled
* Memory device sends error response when the processor bus interface tries to access invalid or restricted memory locations which could generate a bus fault
* When the device is not ready to accept memory transfer
* You may encounter such issues when you play with external memories such as SDRAM connected via DRAM controllers
* Unprivileged access to the private peripheral bus

Usage fault exception

* Execution of undefined instruction (Cortex M4 supports only thumb ISA, so executing any instruction outside this ISA (like ARM ISA) would result in a fault)
* Executing floating point instruction keeping floating point unit disabled
* Trying to switch to ARM state to execute ARM ISA instructions. The T bit of the processor decides ARM state or Thumb state. For cortex M it should be maintained at 1. Making T bit 0 (may happen during function call using function pointers whose 0th bit is not maintained as 1) would result in fault
* Trying to return to thread mode when an exception/interrupt is still active
* Unaligned memory access with multiple load or multiple store instructions
* Attempt to divide by zero (if enabled, by default divide by zero results in zero)
* For all unaligned data access from memory (only if enabled, otherwise cortex m supports unaligned data access)

\_\_attribute\_\_((naked)) functions

* This attribute tells the compiler that the function is an embedded assembly function. You can write the body of the function entirely in assembly code using \_\_asm statements
* The compiler does not generate prologue and epilogue sequences for functions with \_\_attribute\_\_((naked))
* Use naked functions only to write some assembly instructions (\_\_asm statements). Mixing ‘C’ code might not work properly

Exceptions for system-level services

* ARM cortex Mx processor supports 2 important system-level service exceptions. SVC (Supervisory call) and PendSV (Pendable Service)
* Supervisory calls are typically used to request privileged operations for access to system resources from an operation system
* SVC exception is mainly used in an OS environment. For example, a less privileged user task can trigger SVC exception to get system-level services (like accessing device drivers, peripheral) from the kernel of the OS
* PendSV is mainly used in an OS environment to carry out context switching between 2 or more tasks when no other exceptions are active in the system

SVC (Supervisor Call) instruction

* SVC is a thumb ISA instruction which cause SVC exception
* In a RTOS scenario, user tasks can execute SVC instructions with an associated argument to make supervisory calls to seek privileged resources from the kernel code
* Unprivileged user tasks use the SVC instruction to change the processor mode to privileged mode to access privileged resources like peripherals
* SVC instruction is always used along with a number, which can be used to identify the request type by the kernel code
* The SVC handler executes right after the SVC instruction (no delay. Unless a higher priority exception arrives at the same time)

Methods to trigger SVC exception

* Direct execution of SVC instruction with an immediate value. Example: ‘SVC #0X04’ in assembly (Using SVC instruction is very efficient in terms of latency)
* Setting the exception pending bit in ‘System handler control and state register’ (uncommon method)

How to extract the SVC number

* The SVC instruction has a number embedded within it, often referred to as the SVC number
* In the SVC handler, you should fetch the opcode of the SVC instruction and then extract the SVC number
* To fetch the opcode of SVC instruction from program memory, we should have the value of PC (return address) where the user code had interrupted while triggering the SVC exception
* The value of the PC where the user code had interrupted is stored in the stack as a part of exception entry sequence by the processor

PendSV Exception

* It is an exception type 14 and has a programmable priority level
* This exception is triggered by setting its pending status by writing to the ‘Interrupt Control and State Register’ of processor
* Triggering a pendSV system exception is a way of invoking the pre-emptive kernel to carry out the context switch in an os environment
* In an os environment, PendSV handler is set to the lowest priority level, and the PendSV handler carries out the context switch operation

Typical use of PendSV

* Typically, this exception is triggered inside a higher priority exception handler, and it gets executed when the higher priority handler finishes
* Using this characteristic, we can schedule the PendSV exception handler to be executed afer all the other interrupt processing tasks are done
* This is very useful for a context switching operation, which is a crucial operation in various os design
* Using PendSV in context switching will be more efficient in an interrupt noisy environment
* In an interrupt noisy environment, and we need to delay the context switching until all IRQ are executed
* Offloading interrupt processing
* If a higher priority handler is doing time-consuming work, then the other lower priority interrupts will suffer, and systems responsiveness may reduce. This can be solved using a combination of ISR and PendSV handler

What is a task?

* A task is nothing but a piece of code, or you can call it a c function, which does a specific job when it is allowed to run on the CPU
* A task has its own stack to create its local variables when it runs on the CPU. Also, when scheduler decides to remove a task from CPU, scheduler first saves the context (state) of the task in task’s private stack
* In sum, a piece of code or a function is called a stack when it is schedulable and never loses its state unless it is deleted permanently
* Round robin scheduling method is time slices are assigned to each task in equal portions and in circular order

What is context switching?

* Context switching is the procedure of switching out of the currently running task from the CPU after saving the task’s execution context or state and switching in the next task to run on the CPU by retrieving the past execution context or state of the task

Importance of start-up File

* The startup file is responsible for setting up the right environment for the main user code to run
* Code written in startup file runs before main(). So, you can ay startup file calls mains()
* Some part of the startup code file is target dependent
* Startup code takes care of vector table replacement in code memory as required by the ARM cortex Mx processor
* Startup code may also take care of stack reinitialization
* Startup code is responsible for .data, .bss section initialization in main memory

Linker Scripts

* Linker scripts is a text file which explains how different sections of the object files should be merged to create an output file
* Linker and locator combination assigns unique absolute addresses to different sections of the output file by referring to address information mentioned in the linker scripts
* Linker scripts also includes the code and data memory address and size information
* Linker scripts are written using the GNU linker command language
* GNU linker script has the file extension of .ld
* Diagram

  Description automatically generated with medium confidenceYou must supply linker script at the linking phase to the linker using -T option