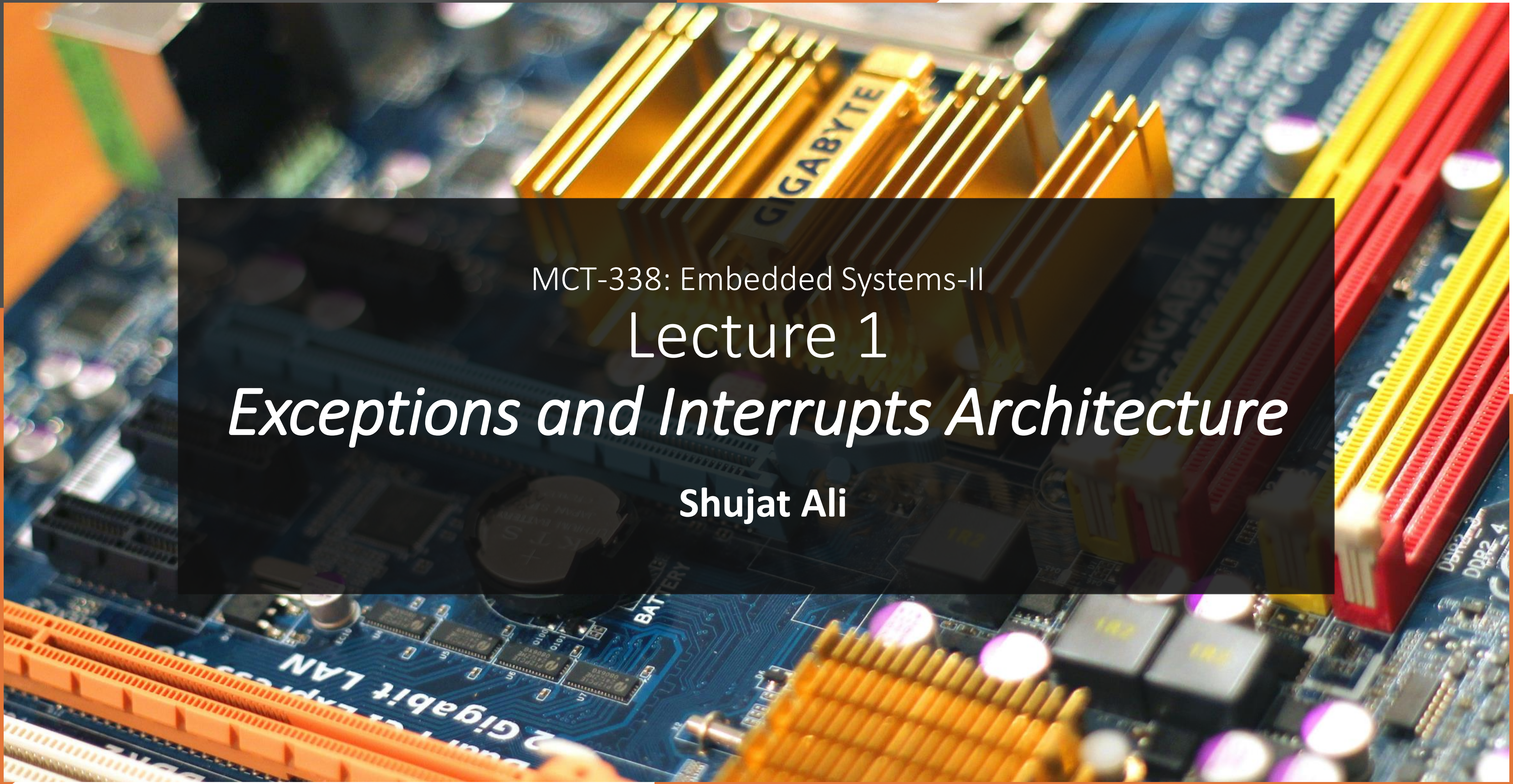
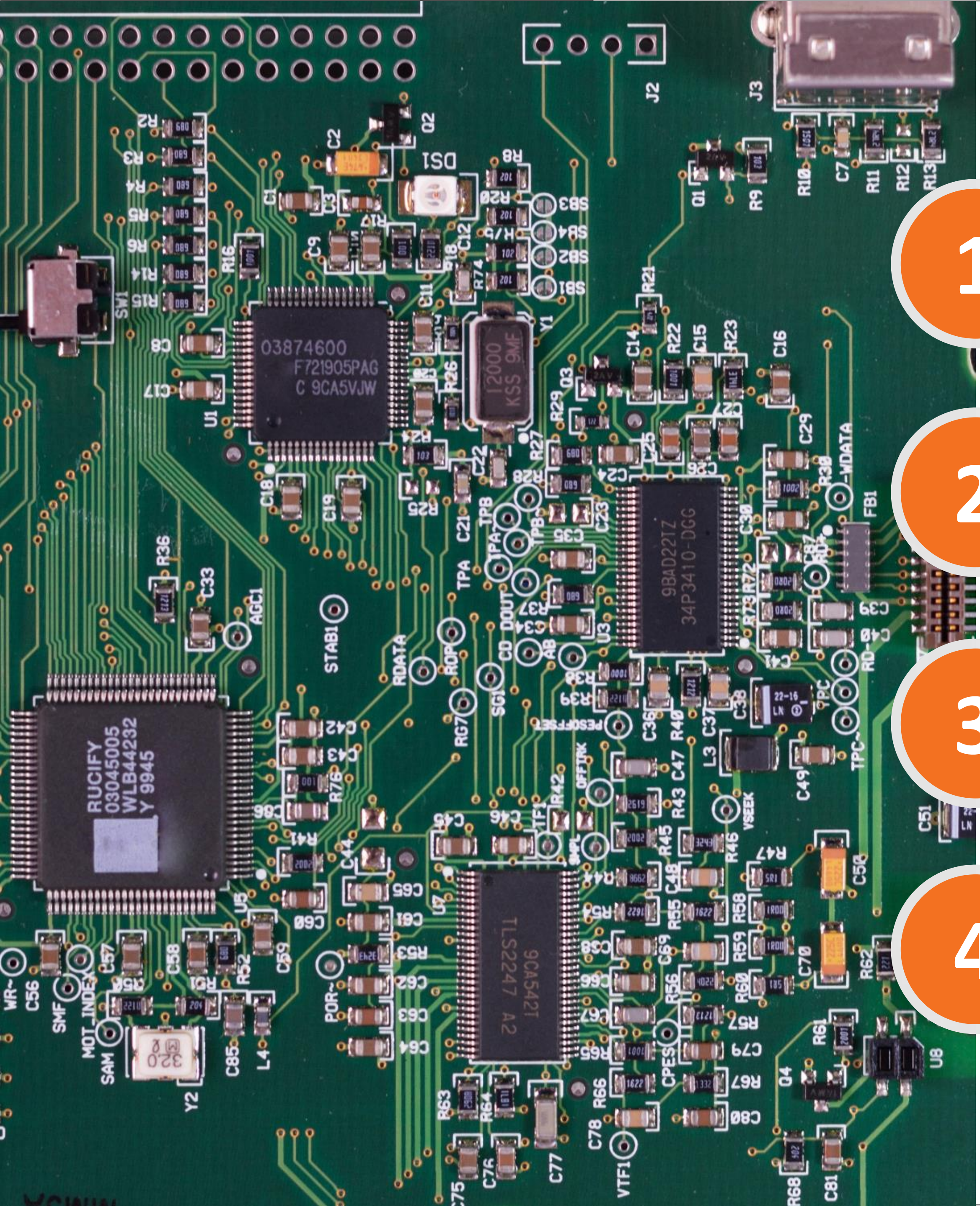


MCT-338: Embedded Systems-II

Lecture 1

Exceptions and Interrupts Architecture

Shujat Ali



1. WHAT IS AN INTERRUPT?
2. CORTEX-M EXCEPTIONS AND INTERRUPTS
3. INTERRUPT CONFIGURATION
4. EXCEPTION/INTERRUPT HANDLING

WHAT IS AN INTERRUPT?

Polling vs Interrupt, ISR, Interrupt Routine Handling

Polling vs. Interrupt

WHAT IS AN INTERRUPT?

Polling vs. Interrupt

WHAT IS AN INTERRUPT?

POLLING

- the process of periodically checking status of a device to see if it is time for the next I/O operation.
- It is basically a protocol in which the CPU services the I/O devices.
- It is an inefficient method, as most of the devices do not require continuous attention

INTERRUPT

- A signal to the CPU to take an immediate action is called an **interrupt**.
- interrupt is a process with the help of which the CPU is notified of requiring attention.
- The interrupt is considered as a hardware mechanism.
- Whenever an interrupt occurs, the CPU stops executing the current program and transfer its control to interrupt handler or **interrupt service routine**.

Interrupt Service Routine (ISR)

WHAT IS AN INTERRUPT?

- Today almost every microcontroller integrates interrupt capability, which can range from simple interrupts to multi-level prioritized interrupts.
- Both hardware (e.g., external inputs or peripherals) as well as software events can generate interrupts.
- Whenever an event or exception happens, the corresponding peripheral or hardware requires a response from the processor.
- The response from the processor is implemented as a function call and is also called **interrupt service routine** (ISR).

Interrupt Routine Handling

WHAT IS AN INTERRUPT?

Key steps involved during interrupt routine handling:

1. One of the interrupt or exception sources generates a request.
2. In response to the interrupt, the processor suspends the currently executing task.
3. The processor executes an interrupt service routine to generate the response and service the source of interrupt.
4. Finally, the processor resumes the execution of previously suspended task from the same state.

CORTEX-M EXCEPTIONS AND INTERRUPTS

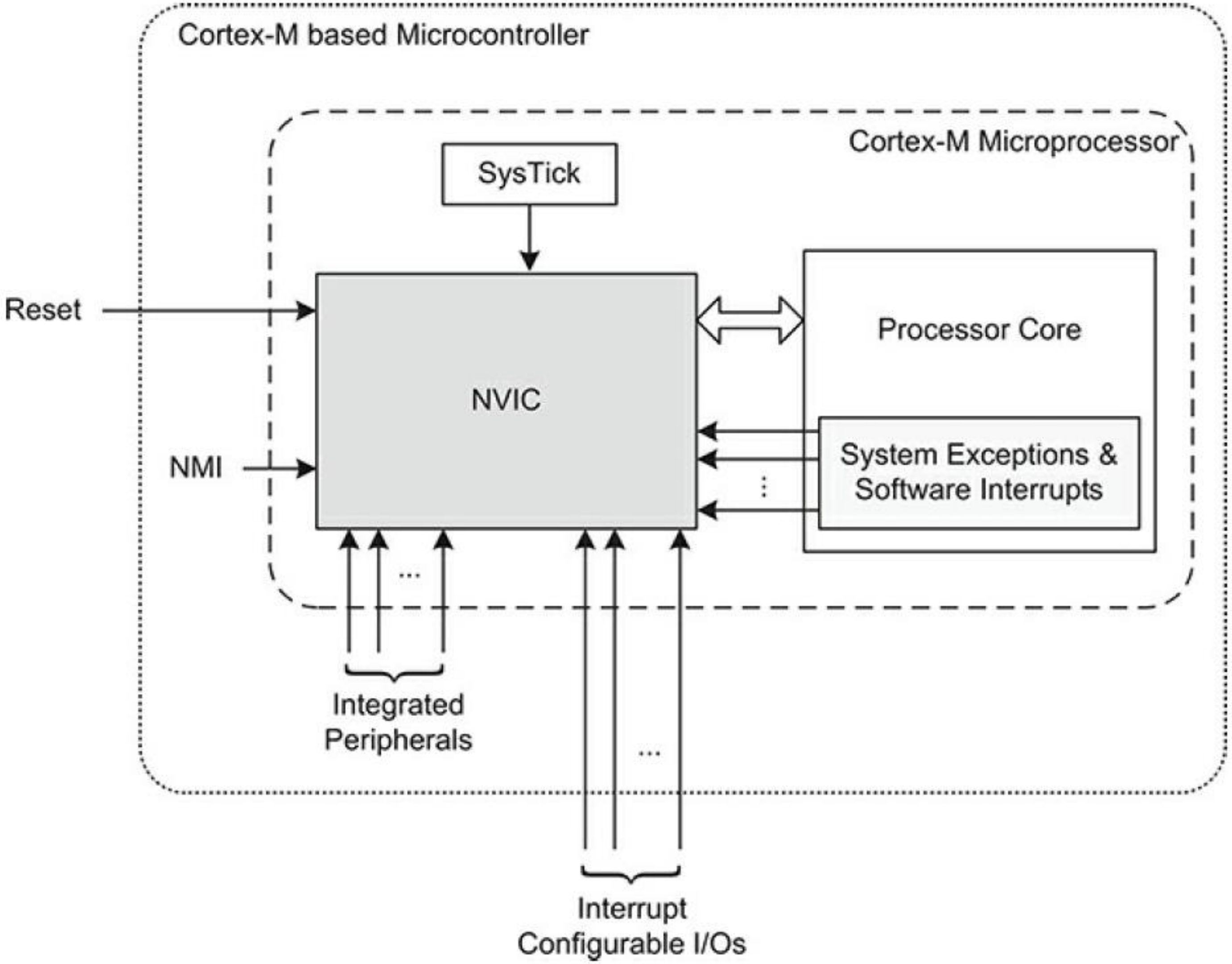
NVIC, System Exceptions and Interrupts and their Priorities & Sates

CORTEX-M EXCEPTIONS

CORTEX-M EXCEPTIONS AND INTERRUPTS

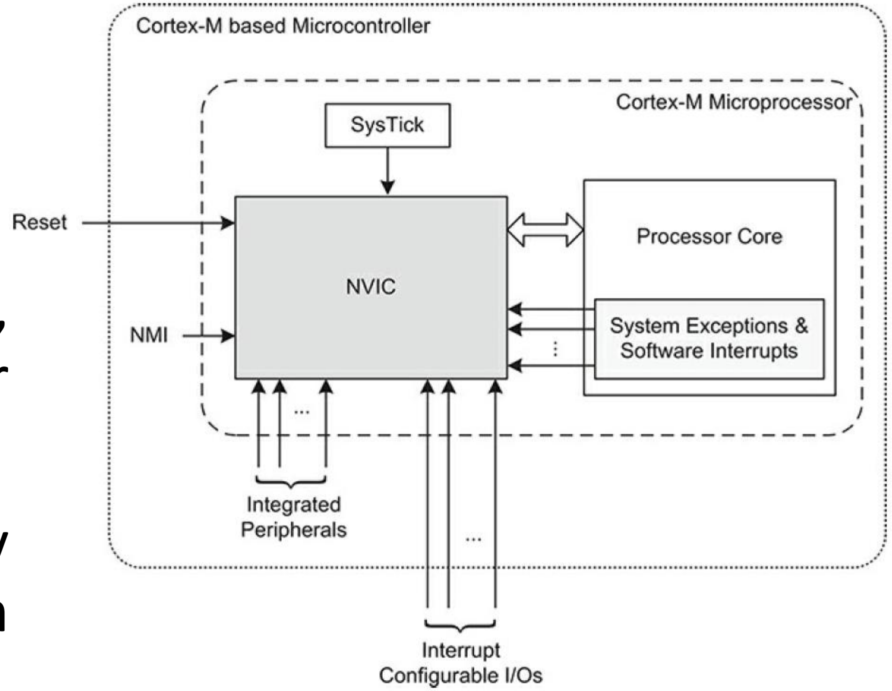
- All Cortex-M processors have a NVIC (Nested Vector Interrupt Controller) that is responsible for handling exceptions and interrupts
- Exceptions are numbered 1 to 255 and according to ARM nomenclature:
  - Exceptions numbered 1 to 15 are called **System Exceptions** or simply **Exceptions**
  - Exceptions numbered 16 to 255 are called **Interrupts**

Block diagram showing NVIC connectivity with ARM Core and different interrupt sources


Cortex-M based Microcontroller
Cortex-M Microprocessor
SysTick
Reset
NMI
NVIC
Processor Core
System Exceptions & Software Interrupts
...
Integrated Peripherals
...
Interrupt Configurable I/Os

Nested Vector interrupt Controller (NVIC)

CORTEX-M EXCEPTIONS AND INTERRUPTS


Cortex-M based Microcontroller
Cortex-M Microprocessor
SysTick
Reset
NVIC
Processor Core
NMI
System Exceptions & Software Interrupts
...
Integrated Peripherals
...
Interrupt Configurable I/Os

- In ARM Cortex-M based µProcessor Architecture, NVIC is tightly integrated with Cortex-M processor core
- NVIC can be configured for the desired functionality using its memory-mapped control registers, mostly in privileged mode
- NVIC also contains a timing module called SYSTICK timer that is responsible for generating a timing reference used by system software to manage and schedule its activities
- NVIC supports 1-240 peripheral interrupts (correspondingly exceptions 16-255), which are also known as **Interrupt Requests** (IRQs)
- Almost all the Cortex-M based microcontrollers support exceptions 1-15, however, the actual number of interrupts that are supported by the microcontroller is determined by the hardware manufacturers

System Exceptions and Interrupts

CORTEX-M EXCEPTIONS AND INTERRUPTS

Exception #, their Labels and Descriptions

#	Label	Description
1	Reset	System Reset Exception
2	NMI	Non-maskable Interrupt. The use of this system exception is defined by the user
3	Hard Fault	This exception is caused by the Bus Fault, Memory Management Fault, or Usage Fault. The Usage Fault occurs if the corresponding interrupt handler cannot be executed.
4	Memory Management Fault	This fault detects memory access violations to regions that are defined in the Memory Protection Unit (MPU). One possibility can be code execution from a memory region with read/write access only.
5	Bus Fault	This system exception occurs when memory access errors are detected when performing instruction fetch, data read or write, interrupt vector fetch or register stacking.
6	Usage Fault	Can occur due to execution of undefined instruction, unaligned memory access (in case of multiple data word load/store instructions). When this exception is enabled, it can detect, divide-by-zero as well as unaligned memory access.

System Exceptions and Interrupts

CORTEX-M EXCEPTIONS AND INTERRUPTS

Exception #, their Labels and Descriptions

#	Label	Description
7-10	Reserved	-
11	SVC	SuperVisor Call used by operating system.
12	Debug Monitor	Debug exception due to events including breakpoints, watchpoints, etc.
13	Reserved	-
14	PendSV	An OS based software exception for scenarios like context switching.
15	SysTick	Exception generates by System Tick Timer. This timer can be used by the OS for system timing reference generation.
16	Interrupt 0	Peripheral Interrupt 0 also called IRQ0. Can be connected to on-chip peripherals or interrupt I/O lines. This argument is valid for all IRQs.
17	Interrupt 1	Peripheral Interrupt 1 also called IRQ1.
…	…	…
255	Interrupt 239	Peripheral Interrupt 239 also called IRQ239.

Exception and Interrupt Priority

CORTEX-M EXCEPTIONS AND INTERRUPTS

- *What if multiple exceptions or interrupt occur at the same time?*
- *Since a microcontroller can perform a single task at a time, then how will it manage to perform these multiple tasks?*

- Exceptions or interrupts can be assigned priority level and microcontroller performs these tasks based on the assigned priority
- In ARM Cortex-M architecture, a higher priority corresponds to a smaller number assigned for priority level
- When the exception priorities are enabled, a higher priority exception can preempt a lower priority (correspondingly a larger value in priority level) exception.

Exception and Interrupt Priority

CORTEX-M EXCEPTIONS AND INTERRUPTS

- ARM Cortex-M based microcontroller support 3 fixed highest-priority levels and up to 128 programmable priorities levels
- Since higher priority corresponds to lower priority value, Reset interrupt is the highest priority interrupt
- Number of programmable priority levels depends on the microcontroller chip manufacturer
- For ARM Cortex-M4 microcontroller on TI TIVA LaunchPad, there are only 8 programmable priority levels for interrupts
- Interrupt-priority level configuration registers are used to assign the required priority level to a specific interrupt

Priority Assignment of different Exceptions

Exception #	Label	Priority
1	Reset	-3
2	NMI	-2
3	Hard Fault	-1
4	Memory Management Fault	Programmable
5	Bus Fault	Programmable
6	Usage Fault	Programmable
…	…	…
16	Interrupt 0	Programmable
17	Interrupt 1	Programmable
…	…	…
255	Interrupt 239	Programmable

Interrupt States

CORTEX-M EXCEPTIONS AND INTERRUPTS

- Due to multiple priority levels of different interrupts, an interrupt can have one of the following operating states.

1. **Active State**

- The interrupt is in active state when it is being serviced by the processor, but the servicing has not been completed yet
- An exception handler of higher priority can interrupt the execution of another lower priority exception handler. In this case, both exceptions are in the active state.

2. **Inactive State**

- An inactive state of an interrupt corresponds to the situation when no interrupt condition has been generated from the corresponding interrupt source
- The interrupt is neither active nor pending in this state

3. **Pending State**

- The interrupt is waiting to be serviced by the processor as it is busy in servicing a high priority interrupt
- The pending state changes to active state when the servicing corresponding to that interrupt starts.

4. **Active and Pending State**

- An interrupt is being serviced by the processor and there is a pending interrupt from the same source

INTERRUPT CONFIGURATION

Interrupt Masking, Interrupt Vector Table

Basics of Interrupt Configuration

INTERRUPT CONFIGURATION

There are two aspects related to interrupt configuration.

- **Global Configurations**: applies to all types of interrupts appearing on the device.
  1. Interrupt/exception masking registers configuration
  2. Setting up interrupt vector table
  3. Configuring interrupt priority groups
- **Local Configurations**: peripheral specific interrupt configuration of the source of interrupt
  1. Enabling and disabling of interrupts locally
  2. Interrupt pending control and status
  3. Priority level configuration
  4. Active status indication

Only the global configurations are discussed in this lecture.

1. Interrupt Masking

INTERRUPT CONFIGURATION

Interrupt/exception masking special registers in a Cortex- M processor:

- Priority Masking Register (PRIMASK)
- Fault Mask Register (FAULTMASK)
- Base Priority Masking Register (BASEPRI)

These registers are useful for interrupt enabling or disabling and mask the interrupts based on the assigned priority levels.

These registers can only be accessed when the processor is operating at privileged access level.

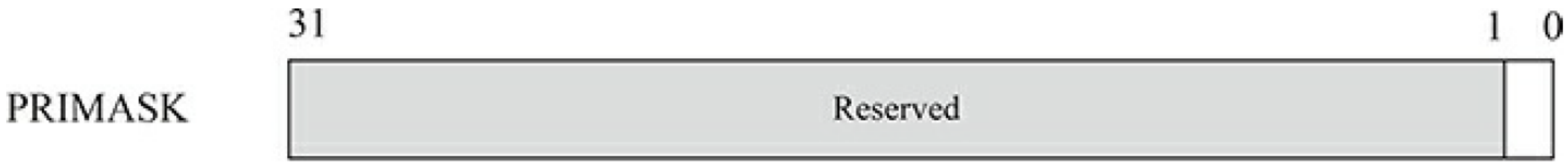
On reset these registers are cleared to zero resulting in no interrupt masking.

1. Interrupt Masking

INTERRUPT CONFIGURATION

Priority Masking Register (PRIMASK)

- Bit **0** of the special register **PRIMASK** is the interrupt mask bit.
  - If this bit is 1, interrupts and exceptions with programmable priority are not allowed.
  - If the bit is 0, then interrupts and exceptions are allowed.
- Reset, non-maskable interrupt (NMI), and hard fault are the only exceptions with fixed priority that are not masked by PRIMASK
- One of the common usages of PRIMASK is to disable all the interrupts when executing a critical code section that should not be interrupted once its execution starts

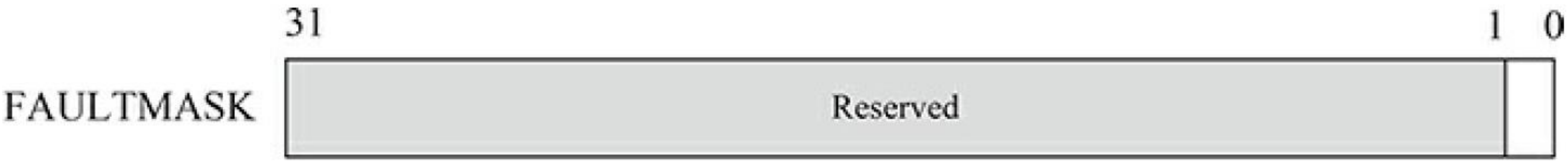


1. Interrupt Masking

INTERRUPT CONFIGURATION

Fault Mask Register (FAULTMASK)

- Bit **0** of the special register **FAULTMASK** is the fault mask bit.
    - If this bit is 1, interrupts and exceptions are not allowed except Reset and NMI
    - If the bit is 0, then interrupts and faults are allowed.
- The interrupt service routine corresponding to FAULTMASK can efficiently avoid any further triggering of fault interrupts.
- For instance, FAULTMASK can be used to suppress any bus faults.
- In contrast to PRIMASK, the FAULTMASK is cleared automatically when returning from an exception.

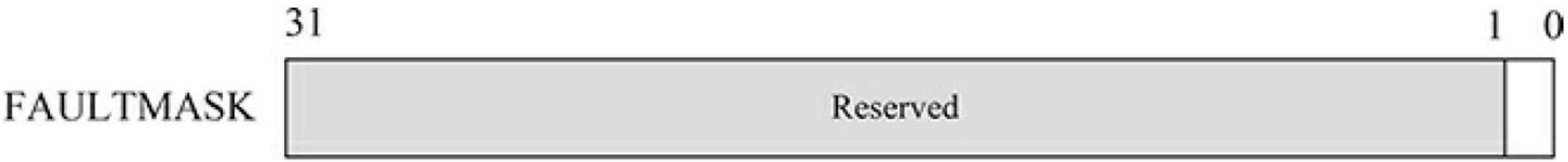


1. Interrupt Masking

INTERRUPT CONFIGURATION

Fault Mask Register (FAULTMASK)

- The FAULTMASK register can be used by the operating system for disabling temporarily the fault handling, in case a task has crashed.
- For this case, the crashing of a task might have been the result of different faults.
- When the operating system is busy in system recovery, it might be desirable to not allow some other faults to interrupt the system.
- Therefore, the FAULTMASK allows the operating system kernel to operate uninterrupted while dealing with fault condition.

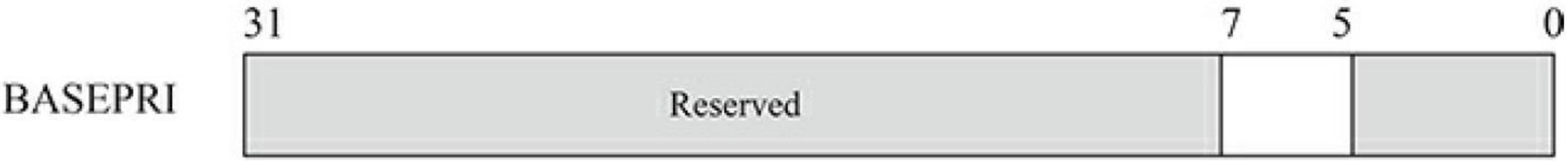


1. Interrupt Masking

INTERRUPT CONFIGURATION

Base Priority Masking Register (BASEPRI)

- It used for disabling interrupts, with flexibility, temporarily when dealing with time critical applications
- When BASEPRI is set to a nonzero value, it blocks all the interrupts of either the same or lower priority, while it allows the processor to accept the interrupts of higher priority for processing.
- For example, if **BASEPRI** bit 5-7 equals 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed.
- When BASEPRI is set to 0, it is disabled.



2. Interrupt Vector Table Setup

INTERRUPT CONFIGURATION

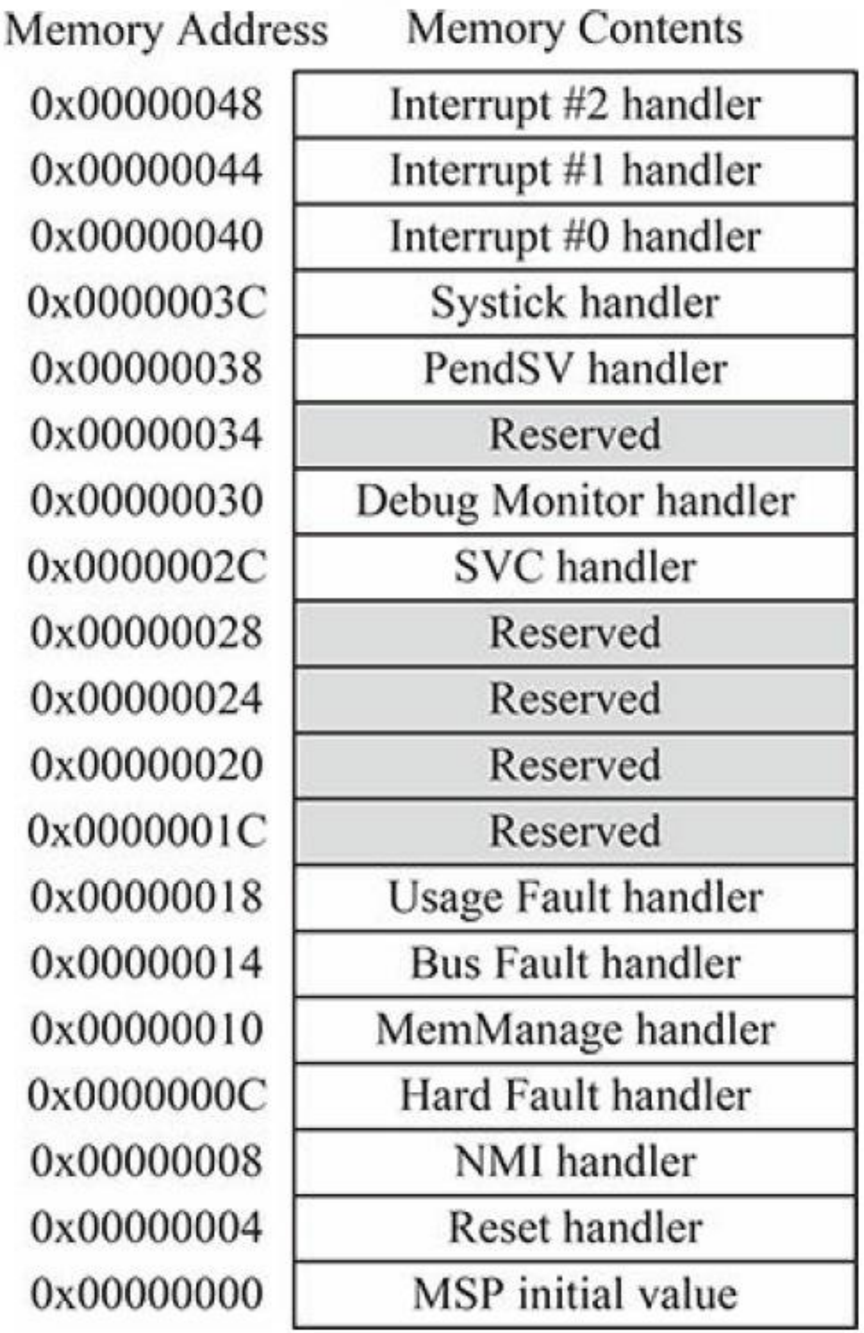
What is Interrupt Vector Table?

- The response to an interrupt by the Cortex-M processor is implemented in the form of a service routine called ISR.
- When an interrupt has occurred and is accepted for processing based on the masking registers configurations as well as priority settings
- The next step for the processor is to obtain the starting address of the corresponding interrupt service routine or the exception handler.
- The starting addresses of the interrupt service routines, for all the interrupts used by the application or the system, are stored in the memory in the form of an **interrupt vector table**.

2. Interrupt Vector Table Setup

INTERRUPT CONFIGURATION

- ISR vector addresses are stored sequentially in ascending order of exception numbers in IVT
- Since there are 256 maximum possible exceptions in ARM Cortex-M Architecture, so there are as many entries in IVT
- The first 16 entries of the interrupt vector table correspond to system exceptions and are always the same across all the Cortex-M based microcontrollers independent of the chip manufacturer.
- However, the vector table entries from 17 onward are used for peripheral interrupts and can be assigned to different peripheral modules in arbitrary order by the microcontroller chip manufacturer.
- The order as well as the assigned location to the entries in the vector table is fixed and they cannot be rearranged



Memory Address	Memory Contents
0x00000048	Interrupt #2 handler
0x00000044	Interrupt #1 handler
0x00000040	Interrupt #0 handler
0x0000003C	Systick handler
0x00000038	PendSV handler
0x00000034	Reserved
0x00000030	Debug Monitor handler
0x0000002C	SVC handler
0x00000028	Reserved
0x00000024	Reserved
0x00000020	Reserved
0x0000001C	Reserved
0x00000018	Usage Fault handler
0x00000014	Bus Fault handler
0x00000010	MemManage handler
0x0000000C	Hard Fault handler
0x00000008	NMI handler
0x00000004	Reset handler
0x00000000	MSP initial value

3. Interrupt Priority Groups Configuration

INTERRUPT CONFIGURATION

- This topic is beyond the scope of this class.

EXCEPTION/INTERRUPT HANDLING

Handlers and Service routines, Basic Steps to Handle Exception/Interrupt, Interrupt Latency, Role of NVIC in Interrupt Handling, Size of Stack Memory and Nested Interrupts

Handlers and Service routines

EXCEPTION/INTERRUPT HANDLING

When an exception or an interrupt occurs, the processor uses either handlers for exceptions or service routines for interrupts.

- **System Handlers**: System exceptions (i.e., exceptions 1 to 15) are handled by system handlers, and usually these handlers are implemented as part of the operating system.
- **Interrupt Service Routines**: The external or peripheral interrupts (i.e., exceptions 16 to 255) are handled by interrupt service routines implemented by user application programs

Basic Steps to Handle Exception/Interrupt

EXCEPTION/INTERRUPT HANDLING

When an exception takes place, a sequence of operations is performed for proper handling of the exception.

Step 1: Current Instruction Execution Status

Step 2: Register Stacking in Response to Interrupt Occurrence

Step 3: Fetch Exception Handler Starting Address

Step 4: Register Updating before Execution of ISR

Step 5: Execute Interrupt Service Routine

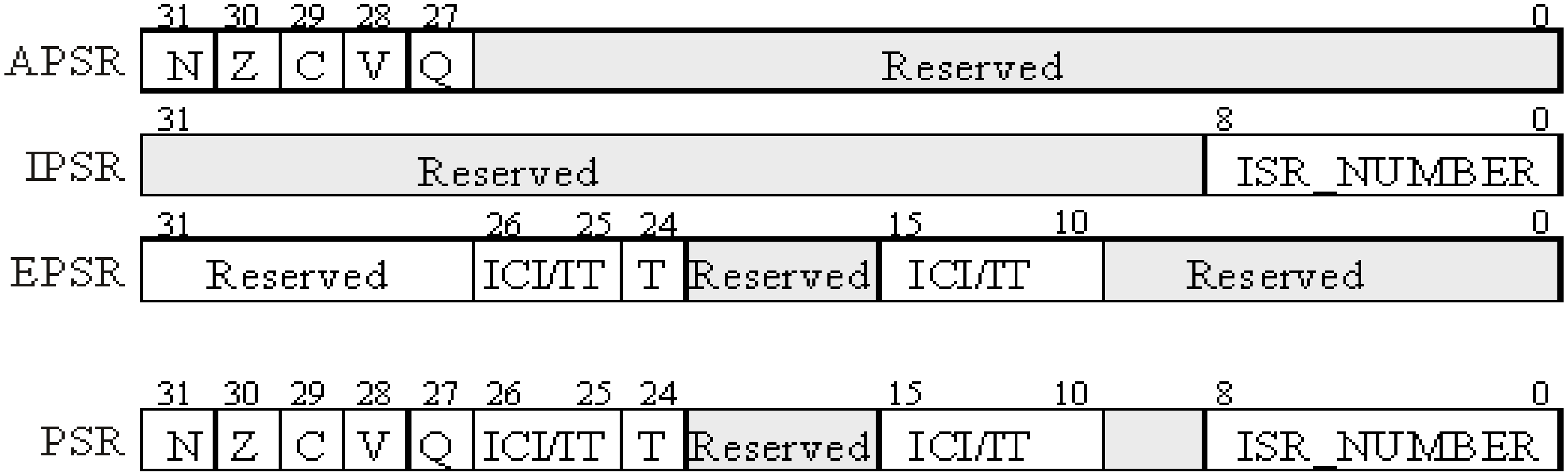
Step 6: Exception Exit or Return

Basic Steps to Handle Exception/Interrupt

EXCEPTION/INTERRUPT HANDLING

Step 1: Current Instruction Execution Status

When an exception takes place, the instruction currently being executed is either finished or terminated based on the value of ICI field in xPSR register.



Basic Steps to Handle Exception/Interrupt

EXCEPTION/INTERRUPT HANDLING

Step 2: Register Stacking in Response to Interrupt Occurrence

- Set of registers R0-R3, R12, LR, PC, and program status register (xPSR), called ***Stack Frame***, are pushed on the stack.

But onto which stack? MSP or PSP?

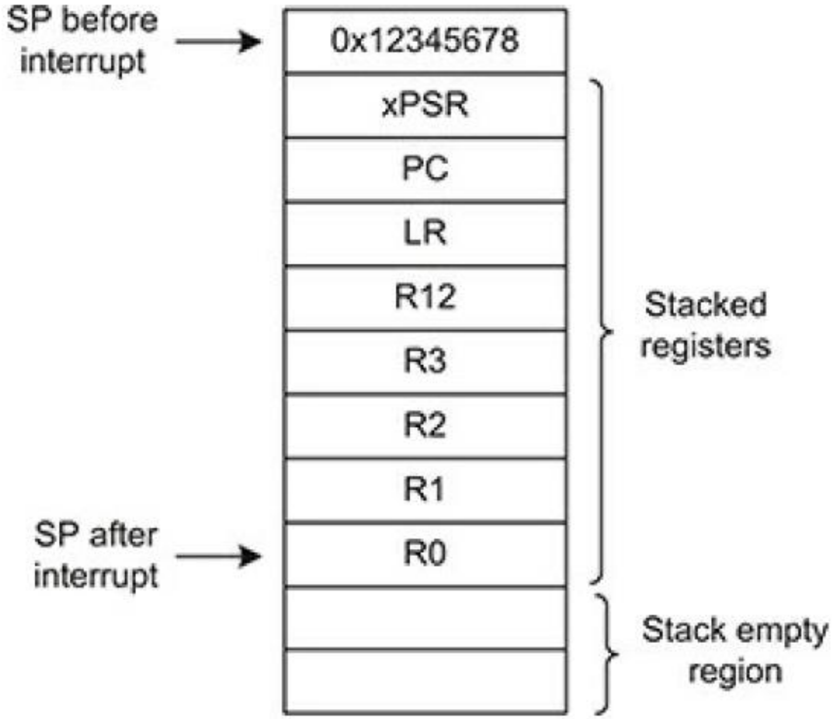
- If an interrupt or exception occurs, the status is saved to either the process or main stack, depending on the active stack pointer, but during handler mode (i.e., after stacking of the registers), only the main stack is used.
- For nested interrupts, where a higher priority interrupt occurs during the execution of a lower priority ISR, the main stack will be used during the execution of both ISRs.

Basic Steps to Handle Exception/Interrupt

EXCEPTION/INTERRUPT HANDLING

Step 2: Register Stacking in Response to Interrupt Occurrence

- The PC and xPSR registers are stacked first and this allows to fetch the ISR in parallel to stacking of other registers.
- The processor automatically stacks its state before entering the interrupt service routine and unstacks the system state while exiting from the ISR.
- This built in hardware capability of register stacking and unstacking
  - simplifies the interrupt programming task
  - makes the latency independent of user implementation.

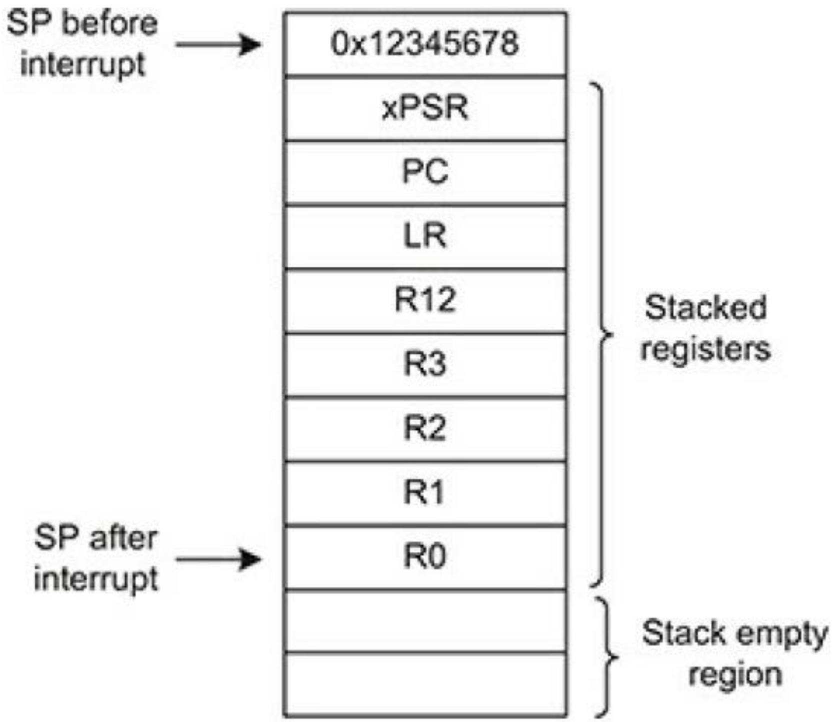


Basic Steps to Handle Exception/Interrupt

EXCEPTION/INTERRUPT HANDLING

Step 2: Register Stacking in Response to Interrupt Occurrence

- According to the C/C++ procedure call standard for the ARM architecture, the registers R0-R3, R12, LR, PC, and xPSR are caller save registers and are required to be stacked when making a function call [ARM(2012)].
- This arrangement allows the interrupt handler to be a normal C function.
- Any other general-purpose register that can be modified by the exception handler should be saved/retrieved on/from the stack by the user (by using PUSH/POP instructions)

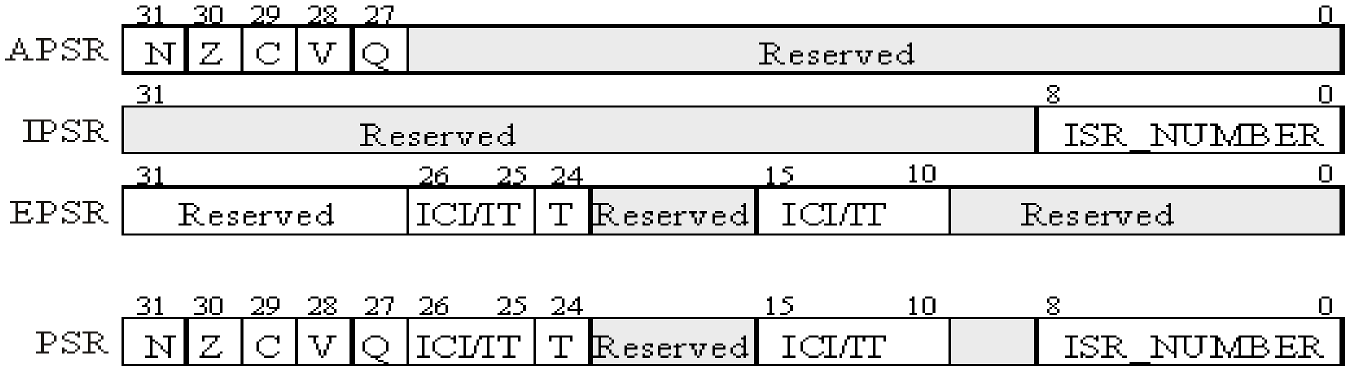


Basic Steps to Handle Exception/Interrupt

EXCEPTION/INTERRUPT HANDLING

Step 3: Fetch Exception Handler Starting Address

- The PC and xPSR registers are stacked first. This allows updating the exception number field (i.e., ISR_NUMBER) in xPSR register by the NVIC to let the processor know about the source of interrupt.
- Once the processor knows about the interrupt source and has stacked the PC register, it can start fetching the ISR, by reading the exception handler starting address from the interrupt vector table and update the PC
- This parallel operation is possible as register stacking is done using the data bus, while fetching of ISR is performed using the instruction bus.

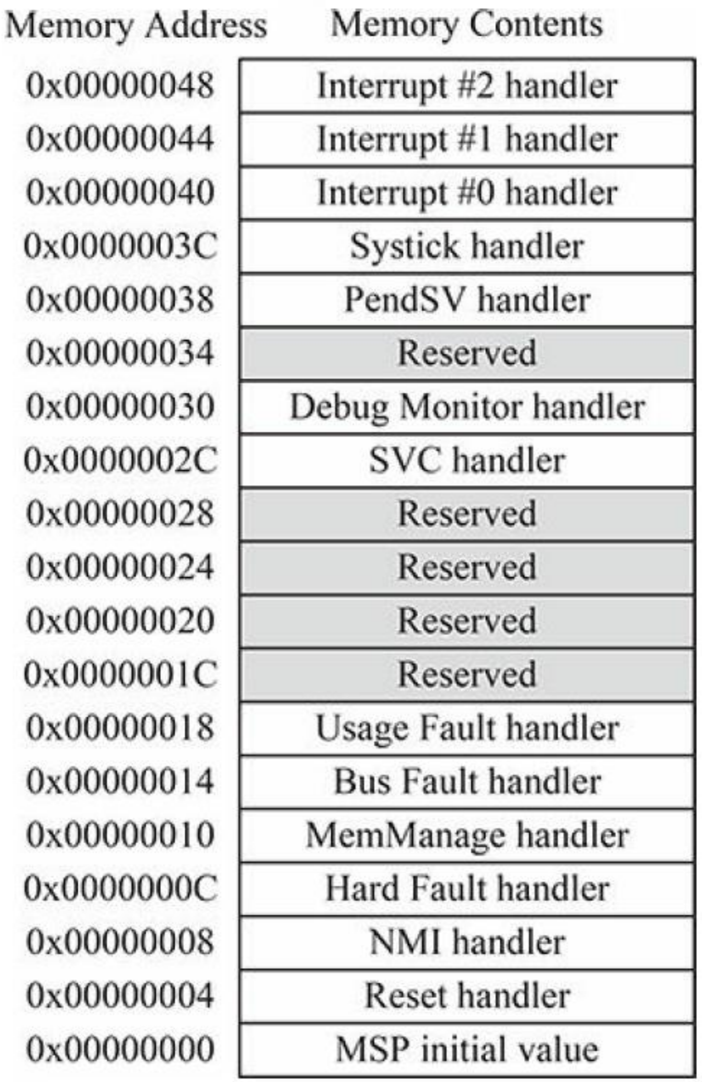


	31	30	29	28	27	0
APSR	N	Z	C	V	Q	Reserved

	31	8	0
IPSR	Reserved	ISR_NUMBER	

	31	26	25	24	15	10	0
EPSR	Reserved	ICI/IT	T	Reserved	ICI/IT	Reserved	

	31	30	29	28	27	26	25	24	15	10	8	0
PSR	N	Z	C	V	Q	ICI/IT	T	Reserved	ICI/IT		ISR_NUMBER	



Memory Address	Memory Contents
0x00000048	Interrupt #2 handler
0x00000044	Interrupt #1 handler
0x00000040	Interrupt #0 handler
0x0000003C	Systick handler
0x00000038	PendSV handler
0x00000034	Reserved
0x00000030	Debug Monitor handler
0x0000002C	SVC handler
0x00000028	Reserved
0x00000024	Reserved
0x00000020	Reserved
0x0000001C	Reserved
0x00000018	Usage Fault handler
0x00000014	Bus Fault handler
0x00000010	MemManage handler
0x0000000C	Hard Fault handler
0x00000008	NMI handler
0x00000004	Reset handler
0x00000000	MSP initial value

Basic Steps to Handle Exception/Interrupt

EXCEPTION/INTERRUPT HANDLING

Step 4: Register Updating before Execution of ISR

After stacking of the registers is completed, following registers are updated.

- ***SP***: The stack pointer will be updated to a new value during register stacking.
- ***xPSR***: The least significant nine bits updated to ISR_NUMBER of interrupt
- ***PC***: Updated to the exception handler and starts fetching the service routine instructions from the location of the exception handler.
- ***LR***: The LR will be updated to a special value called EXC-RETURN, which controls the exception or interrupt return type, when returning from the service routine.

Basic Steps to Handle Exception/Interrupt

EXCEPTION/INTERRUPT HANDLING

Step 5: Execute Interrupt Service Routine

- Specific interrupt service routine, written by the user, corresponding to the interrupt source is performed.

Basic Steps to Handle Exception/Interrupt

EXCEPTION/INTERRUPT HANDLING

Step 6: Exception Exit or Return

- After completing the execution of exception or interrupt handler, an exception or interrupt return is performed to restore the system status
- During the execution of interrupt return process, following steps take place:
- ***Unstacking***: All the registers that have been pushed to the stack are restored during this process. The stack pointer register is also updated during this process.
- ***NVIC register update***: The active bit corresponding to current exception is cleared. If an external interrupt remains active after clearing the current exception, the pending bit is set again, causing the processor to re-enter the interrupt service routine.

In the case of Cortex-M, return from interrupt can be performed using normal return instruction, allowing to implement the entire interrupt service routine as a normal C function.

Interrupt Latency

EXCEPTION/INTERRUPT HANDLING

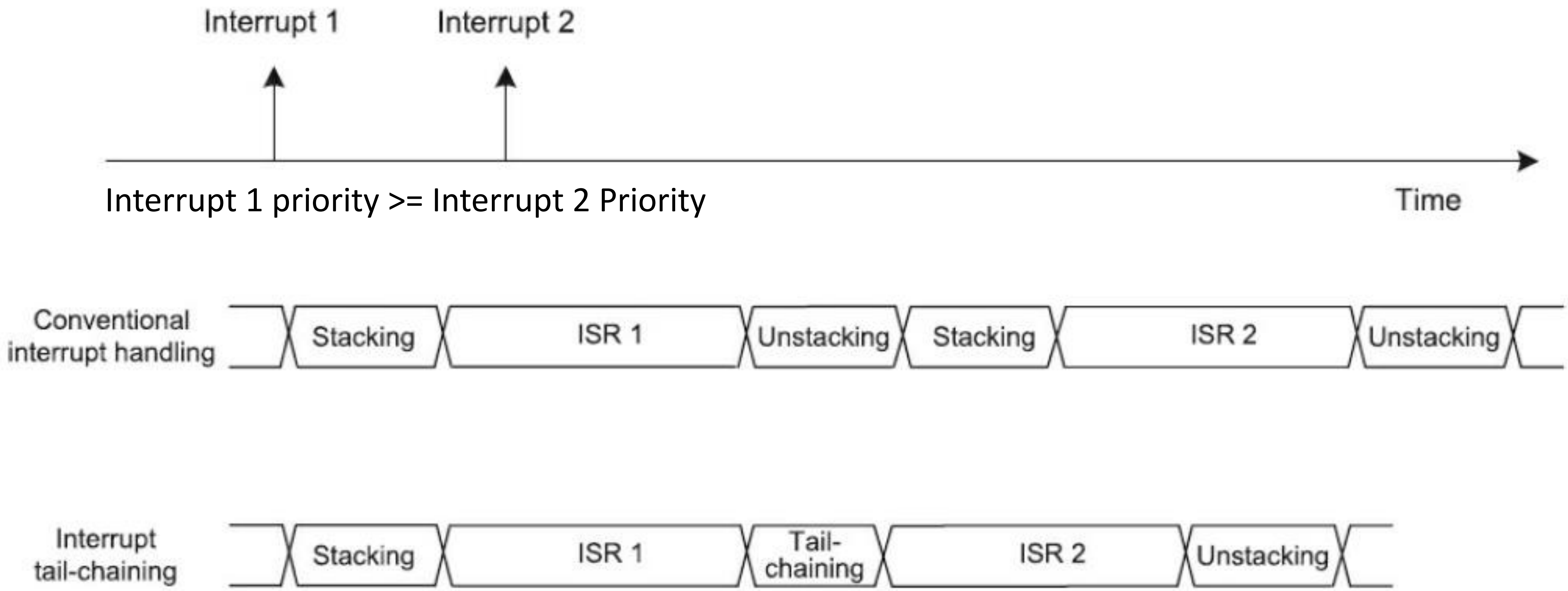
The interrupt latency is a crucial parameter for real time applications, it has two components:

- **Interrupt Entry Latency**: quantified as the number of processor clock cycles from the arrival of an interrupt signal at the processor until the execution of the first instruction in the interrupt service routine starts.
- **Interrupt Exit Latency:** equal to the number of clock cycles required from the execution of the interrupt return instruction until the execution of the next instruction of the interrupted task.

The Cortex-M3/M4 processor has the minimum interrupt entry latency of 12-cycles and interrupt exit latency of 10-cycles.

Interrupt Tail Chaining

EXCEPTION/INTERRUPT HANDLING



Interrupt 1 priority >= Interrupt 2 Priority

Interrupt Tail Chaining

EXCEPTION/INTERRUPT HANDLING

- ARM Cortex-M architecture avoids unnecessary register unstacking/stacking cycle, switching from pending interrupt to active interrupt, by implementing tail-chaining in the NVIC
- It improves both interrupt latency as well as its power efficiency (due to reduced number of memory read/write operations).
- Tail-chaining reduces the latency between two interrupt handlers to 6 cycles (to fetch interrupt handler corresponding to second ISR) compared to 22 (10+12) cycles.
- In the case of multiple nested interrupts, the processor will tail-chain a pending interrupt that has higher priority than all other interrupts in pending state.

Role of NVIC in Interrupt Handling

EXCEPTION/INTERRUPT HANDLING

- When multiple different priorities can be assigned to interrupts, the occurrence of nested interrupts is a natural consequence.
- The NVIC in ARM Cortex-M processor core is responsible for decoding interrupt priorities and assign appropriate interrupt status to each interrupt event
- During any active interrupt already being processed by the processor, and
  - a high priority interrupt occurs, then the state of current interrupt is changed to pending and newly arriving interrupt is processed immediately by the processor
  - a low priority interrupt occurs, then newly arriving interrupt is assigned pending state and the processor continues executing the ISR corresponding to currently active interrupt

Size of Stack Memory and Nested Interrupts

EXCEPTION/INTERRUPT HANDLING

- When many different priorities are assigned to multiple interrupts, then the size of the main stack should be assigned by considering the possibility of multiple nested interrupts
- At each interrupt nesting level, at least 8 registers (called *stack frame*) are pushed onto the stack assuming the ISR does not require extra stack space
- For *n* nested interrupt levels, the minimum stack size required for worst case sequence of nested interrupt occurrences will be *32n* bytes
- These *32n* bytes will be used just for register stacking by the hardware and any stack operations inside ISRs will require additional stack space

Thank You!

Any Questions?