ALP Lab Evaluation

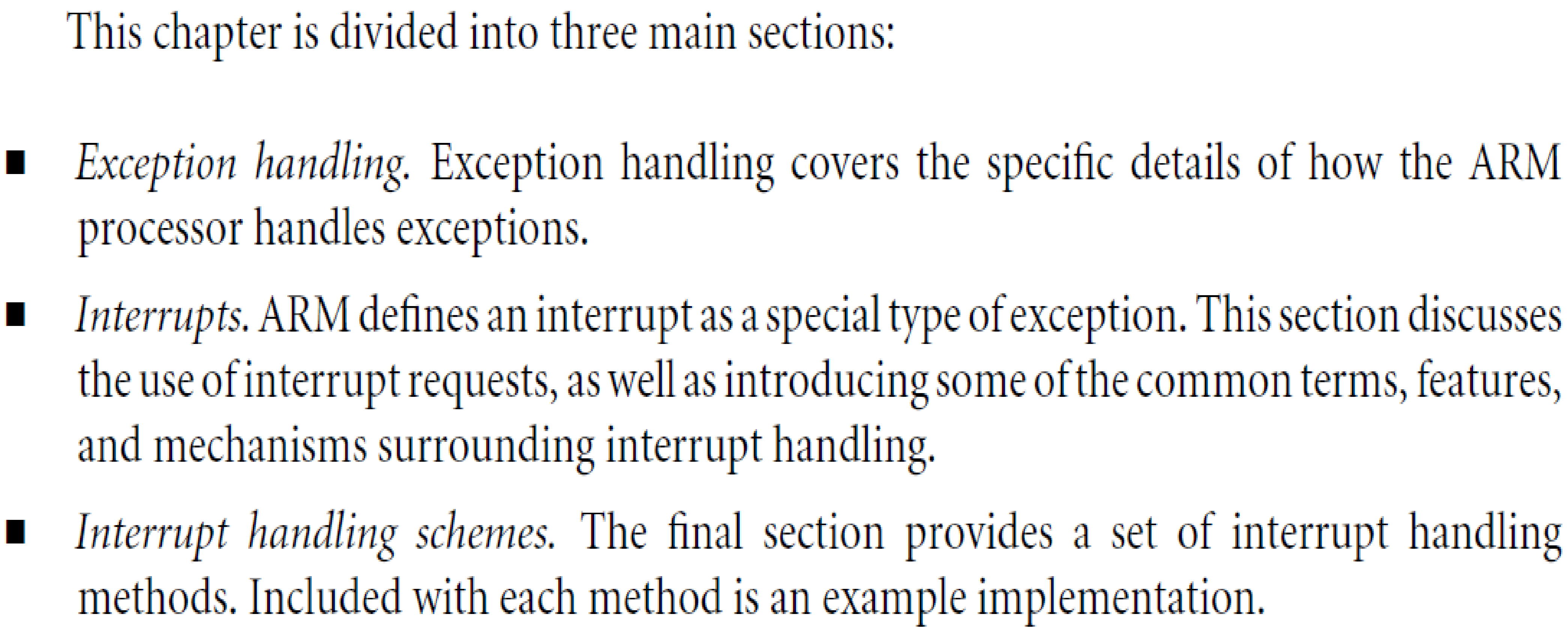
Total Marks: 30M

- Code with appropriate comments: 5M
- Explanation of the code:5M
- Testing program with different set of input values:5M
- Viva Questions: theory and Lab: 5M
- All programs (18) complete execution in system: 10M

Exception and Interrupt Handling

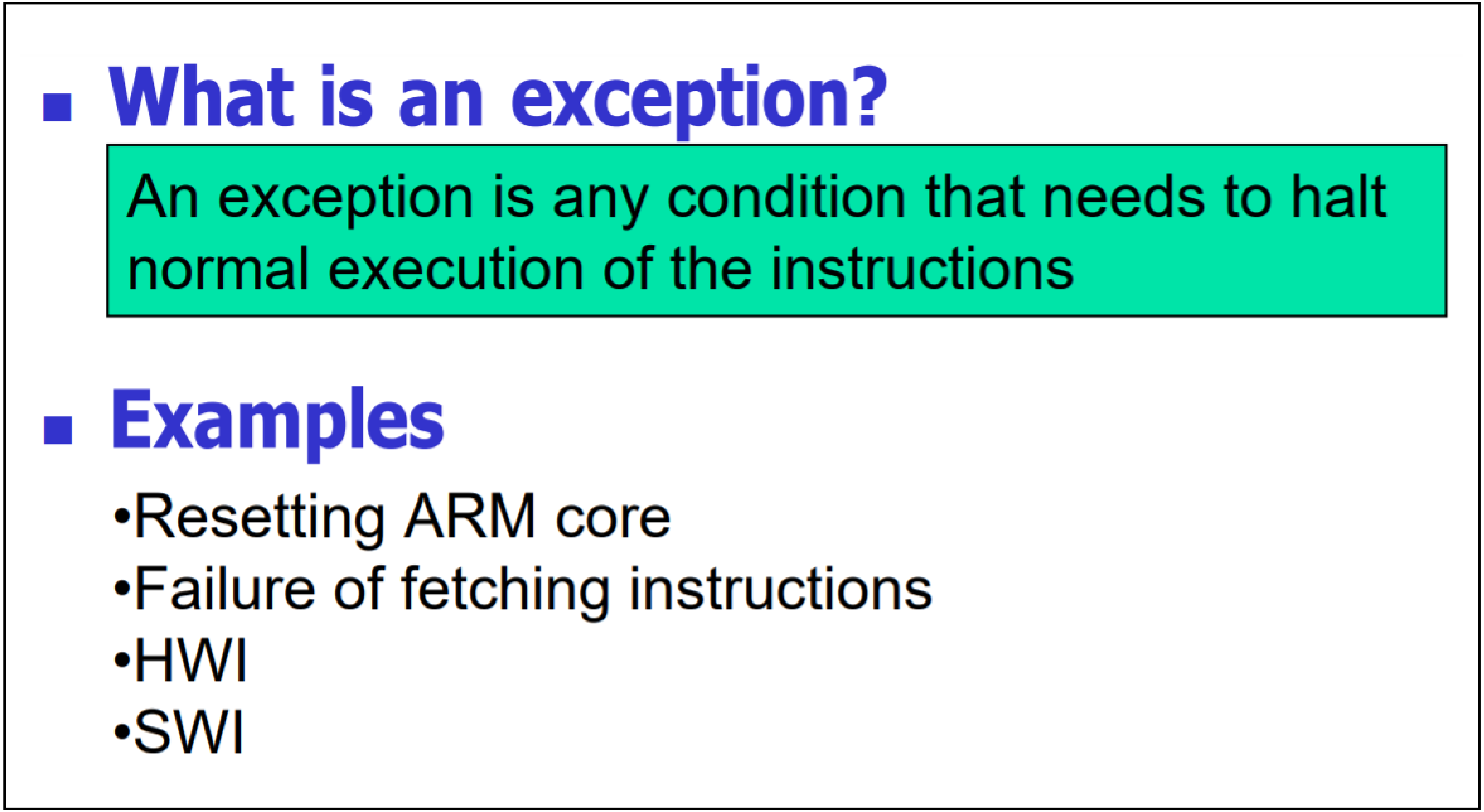
```
mov r1,r2
mov r3,#3
div r3,r1
          IVT: mem addr. Of 1st instr.    ISR1
          mem. Addr of 1st instr.          ISR2

          ISR1:
                                   --- IRET
```

Exception and Interrupt Handling



This chapter is divided into three main sections:

- *Exception handling.* Exception handling covers the specific details of how the ARM processor handles exceptions.
- *Interrupts.* ARM defines an interrupt as a special type of exception. This section discusses the use of interrupt requests, as well as introducing some of the common terms, features, and mechanisms surrounding interrupt handling.
- *Interrupt handling schemes.* The final section provides a set of interrupt handling methods. Included with each method is an example implementation.

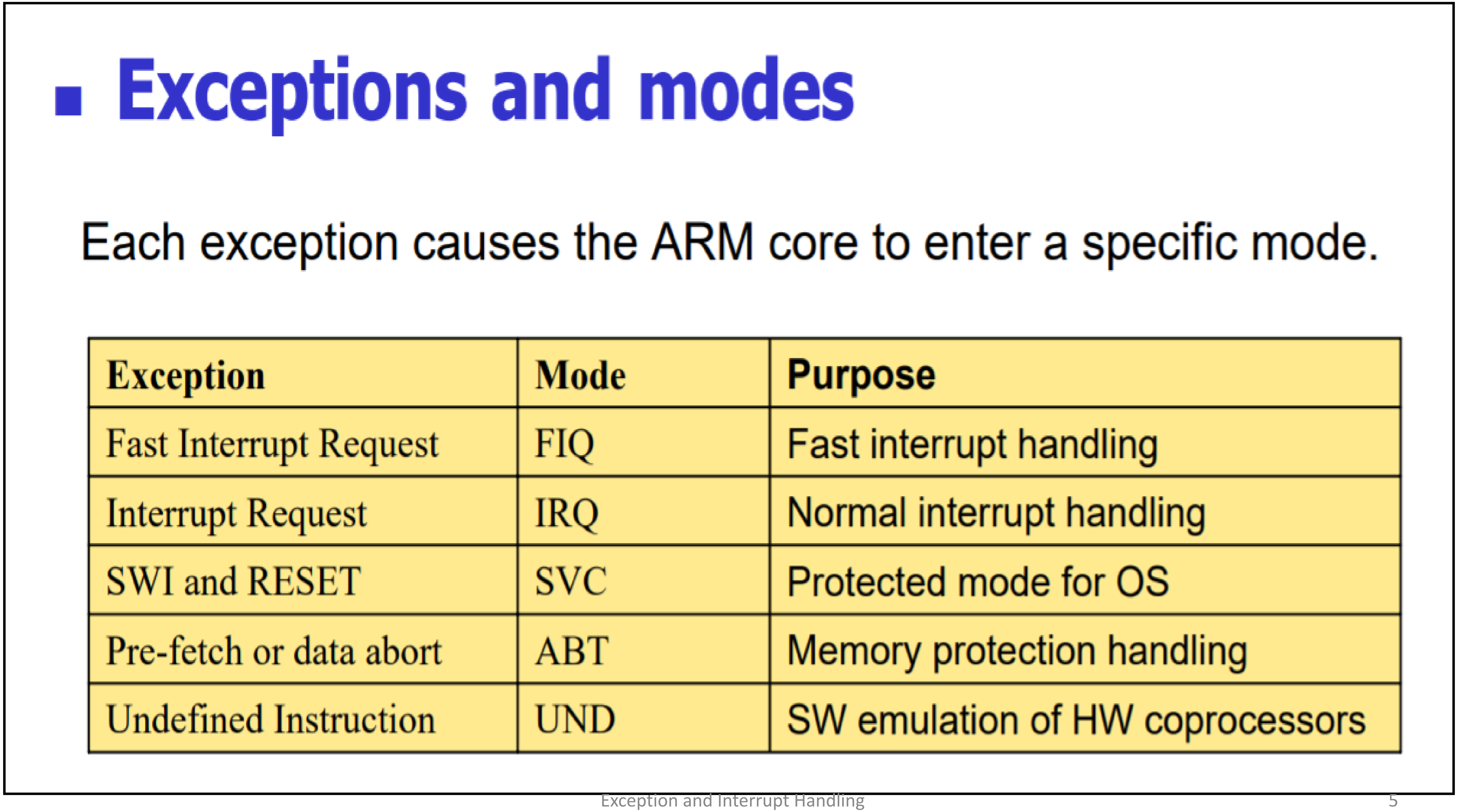


What is an exception?

An exception is any condition that needs to halt normal execution of the instructions

Examples

- Resetting ARM core
- Failure of fetching instructions
- HWI
- SWI



Exceptions and modes

Each exception causes the ARM core to enter a specific mode.

Exception	Mode	Purpose
Fast Interrupt Request	FIQ	Fast interrupt handling
Interrupt Request	IRQ	Normal interrupt handling
SWI and RESET	SVC	Protected mode for OS
Pre-fetch or data abort	ABT	Memory protection handling
Undefined Instruction	UND	SW emulation of HW coprocessors

Exceptions and Modes

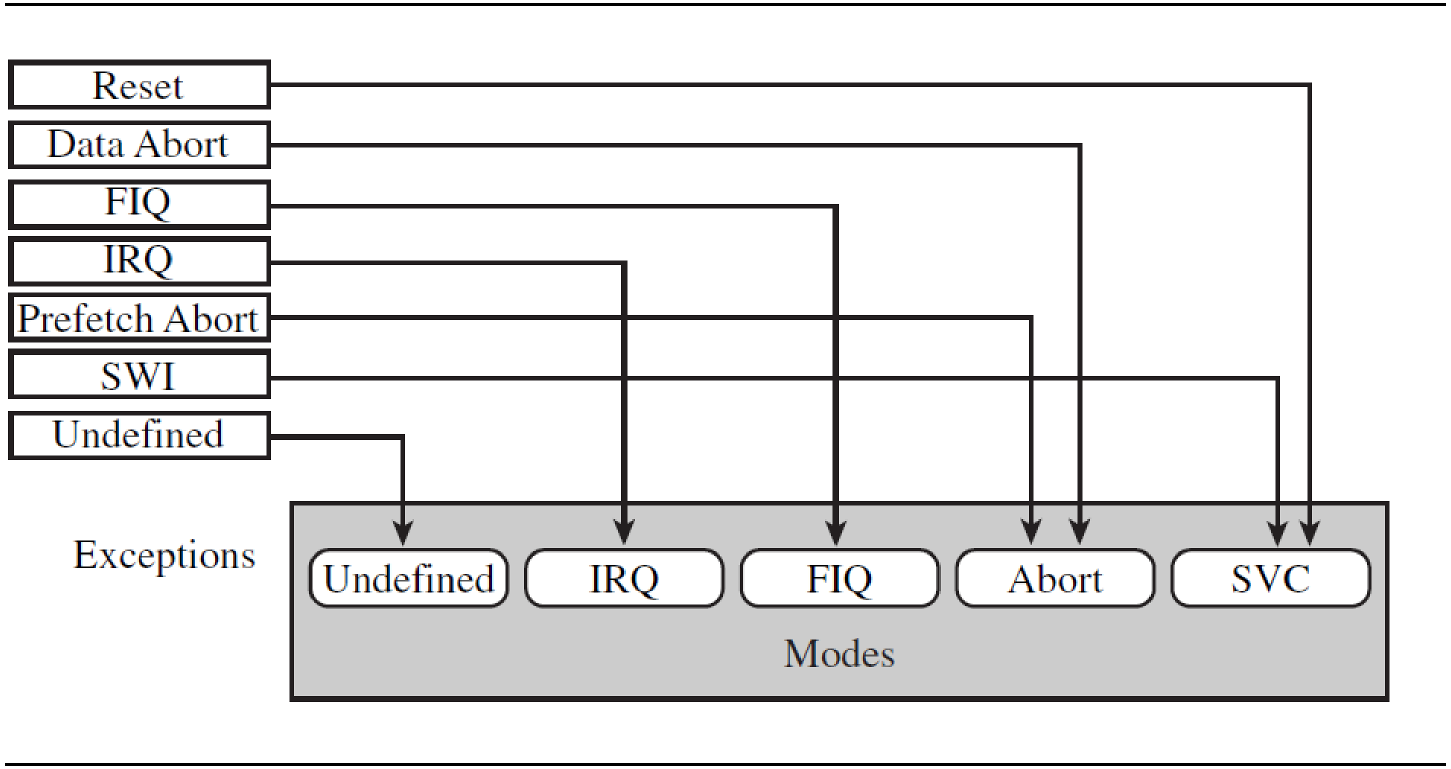
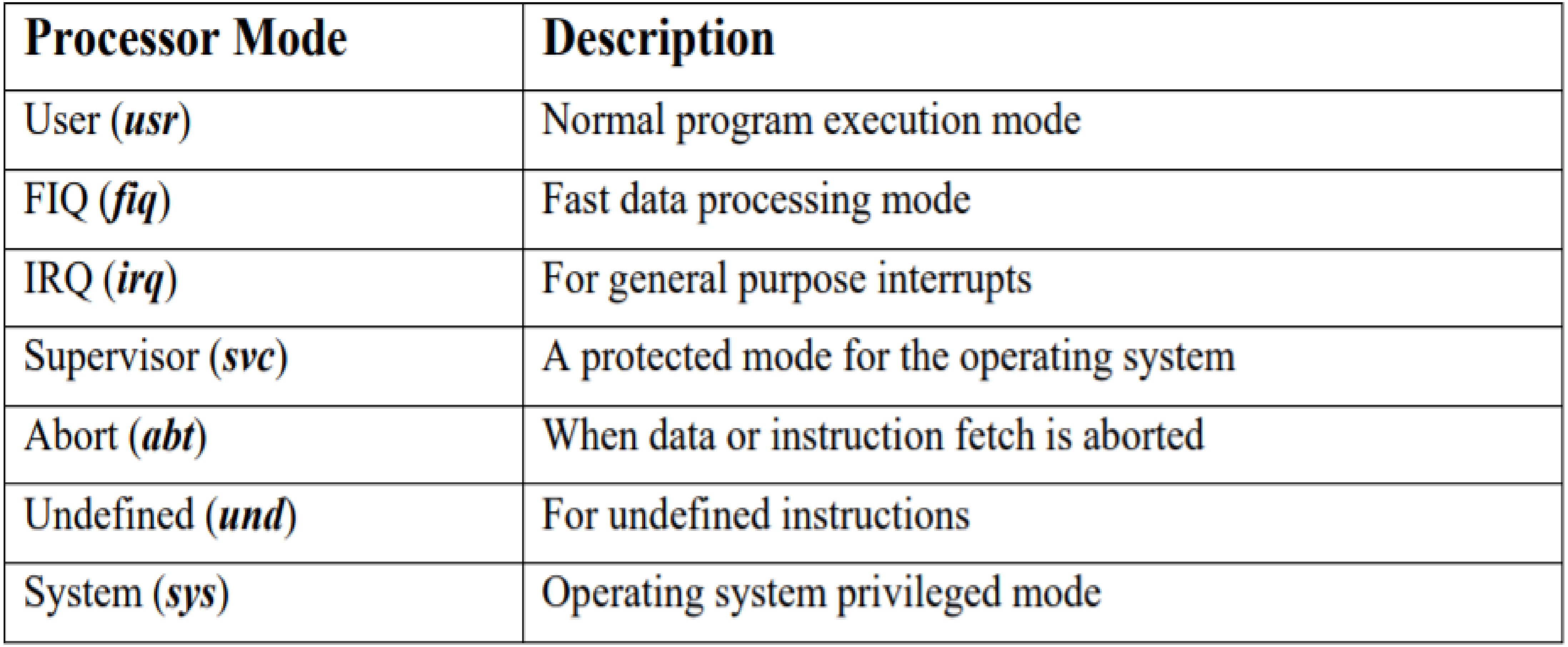


Figure 9.1 Exceptions and associated modes.

Exceptions and Modes



Processor Mode	**Description**
User (***usr***)	Normal program execution mode
FIQ (***fiq***)	Fast data processing mode
IRQ (***irq***)	For general purpose interrupts
Supervisor (***svc***)	A protected mode for the operating system
Abort (***abt***)	When data or instruction fetch is aborted
Undefined (***und***)	For undefined instructions
System (***sys***)	Operating system privileged mode

Exceptions and Modes

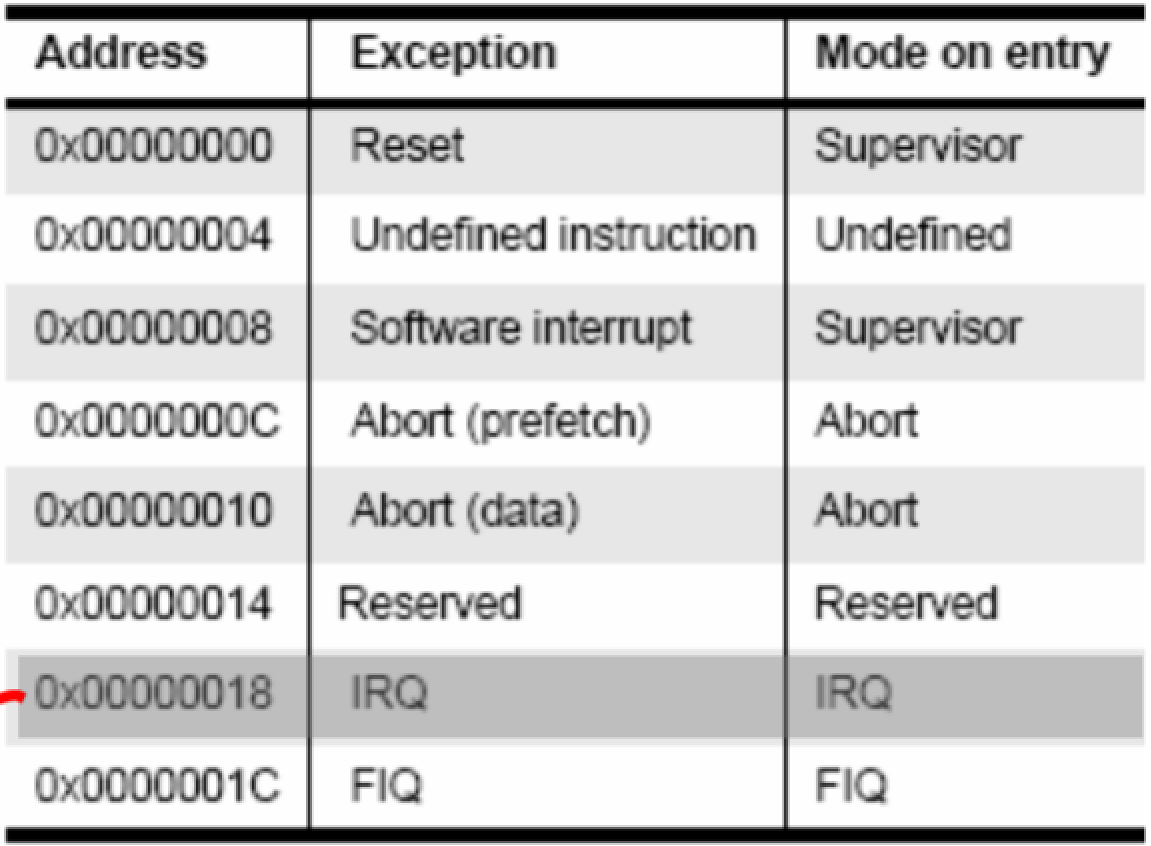
- When exception causes a mode change, the core automatically
  - Saves ***cpsr*** to ***spsr*** of the exception mode.
  - Saves ***pc*** to ***lr*** of the exception mode.
  - Sets ***cpsr*** to the exception mode.
  - Sets ***pc*** to the address of the exception handler.

▪ Vector table

It is a table of addresses that the ARM core branches to when an exception is raised and there is always branching instructions that direct the core to the ISR.

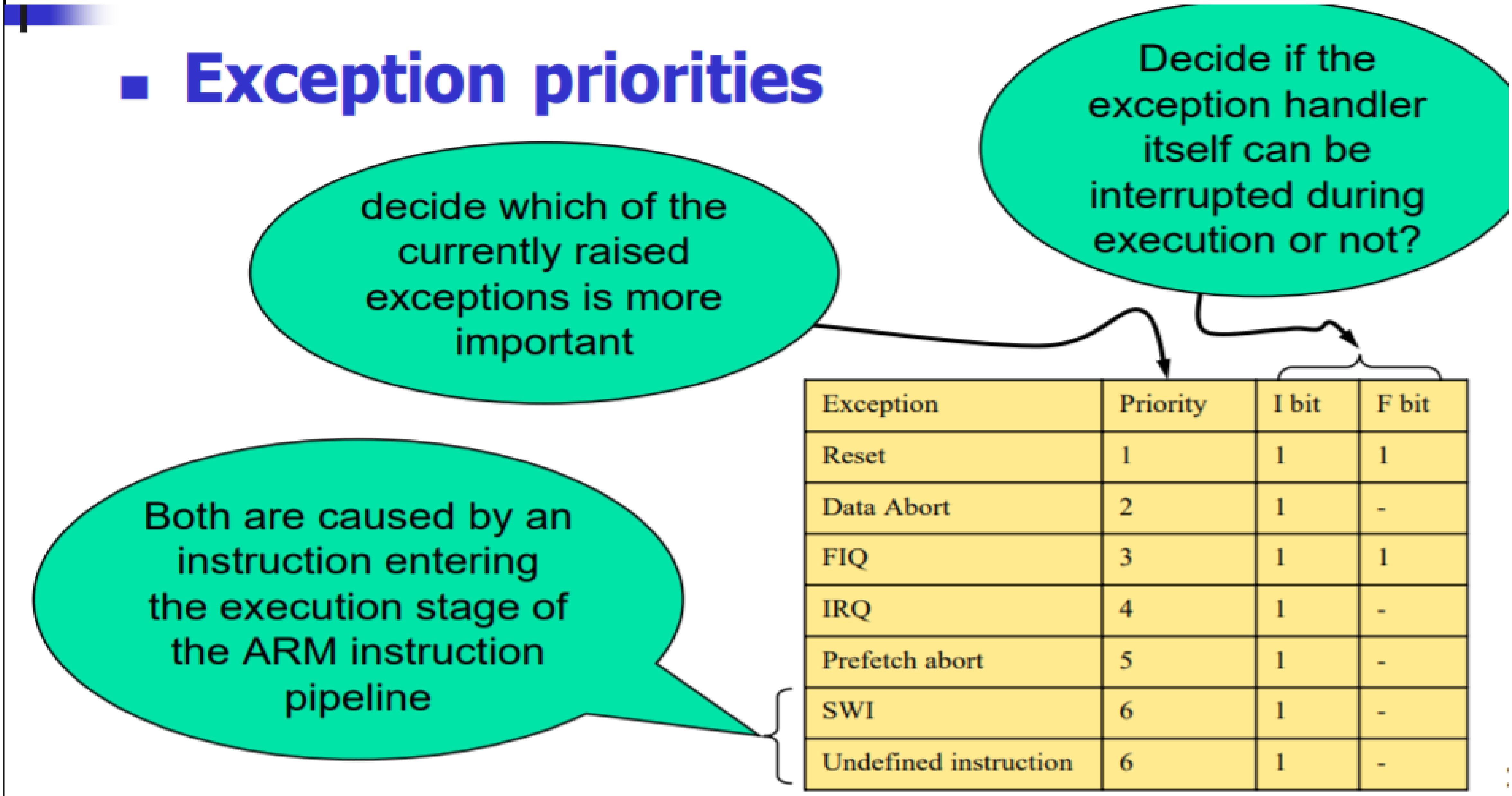
At this place in memory, we find a branching instruction

ldr pc, [pc, #_IRQ_handler_offset]



Address	Exception	Mode on entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

▪ Exception priorities

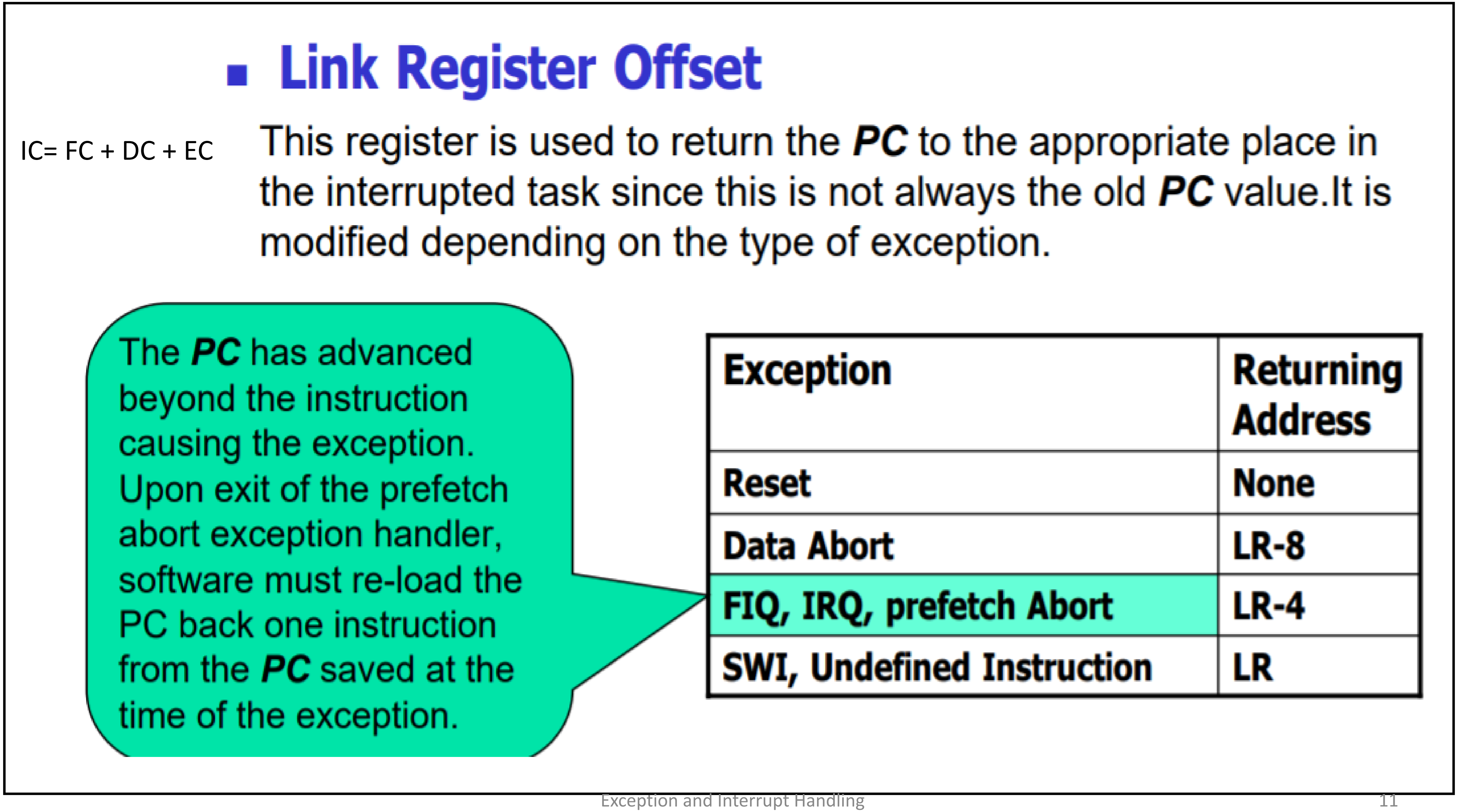


decide which of the currently raised exceptions is more important

Decide if the exception handler itself can be interrupted during execution or not?

Both are caused by an instruction entering the execution stage of the ARM instruction pipeline

Exception	Priority	I bit	F bit
Reset	1	1	1
Data Abort	2	1	-
FIQ	3	1	1
IRQ	4	1	-
Prefetch abort	5	1	-
SWI	6	1	-
Undefined instruction	6	1	-



■ Link Register Offset

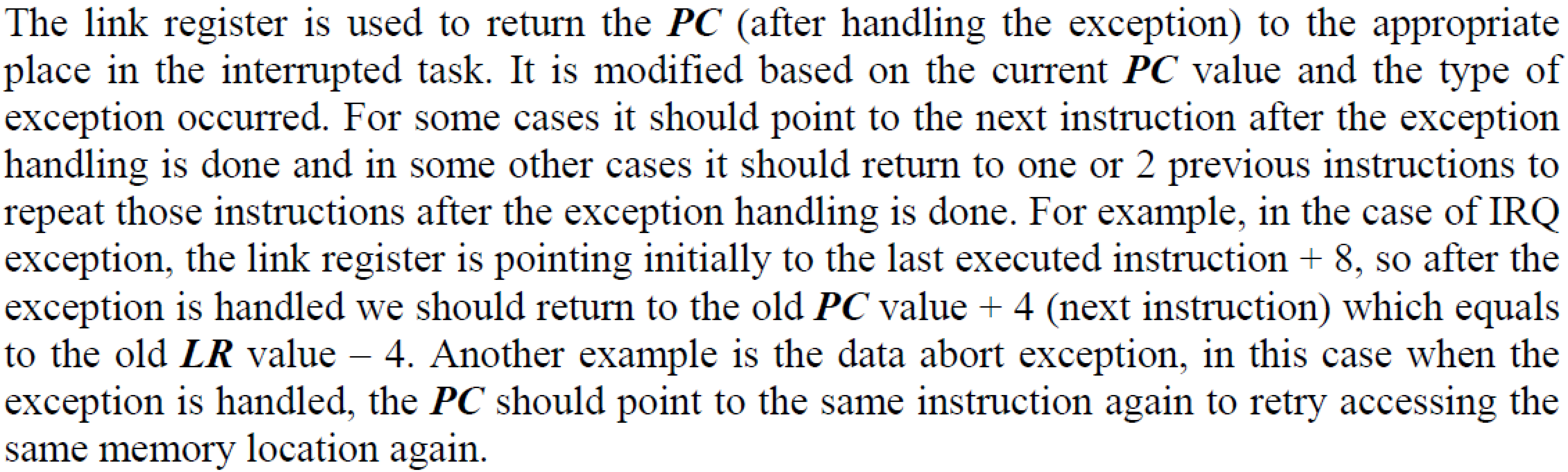
IC= FC + DC + EC

This register is used to return the ***PC*** to the appropriate place in the interrupted task since this is not always the old ***PC*** value.It is modified depending on the type of exception.

The ***PC*** has advanced beyond the instruction causing the exception. Upon exit of the prefetch abort exception handler, software must re-load the PC back one instruction from the ***PC*** saved at the time of the exception.

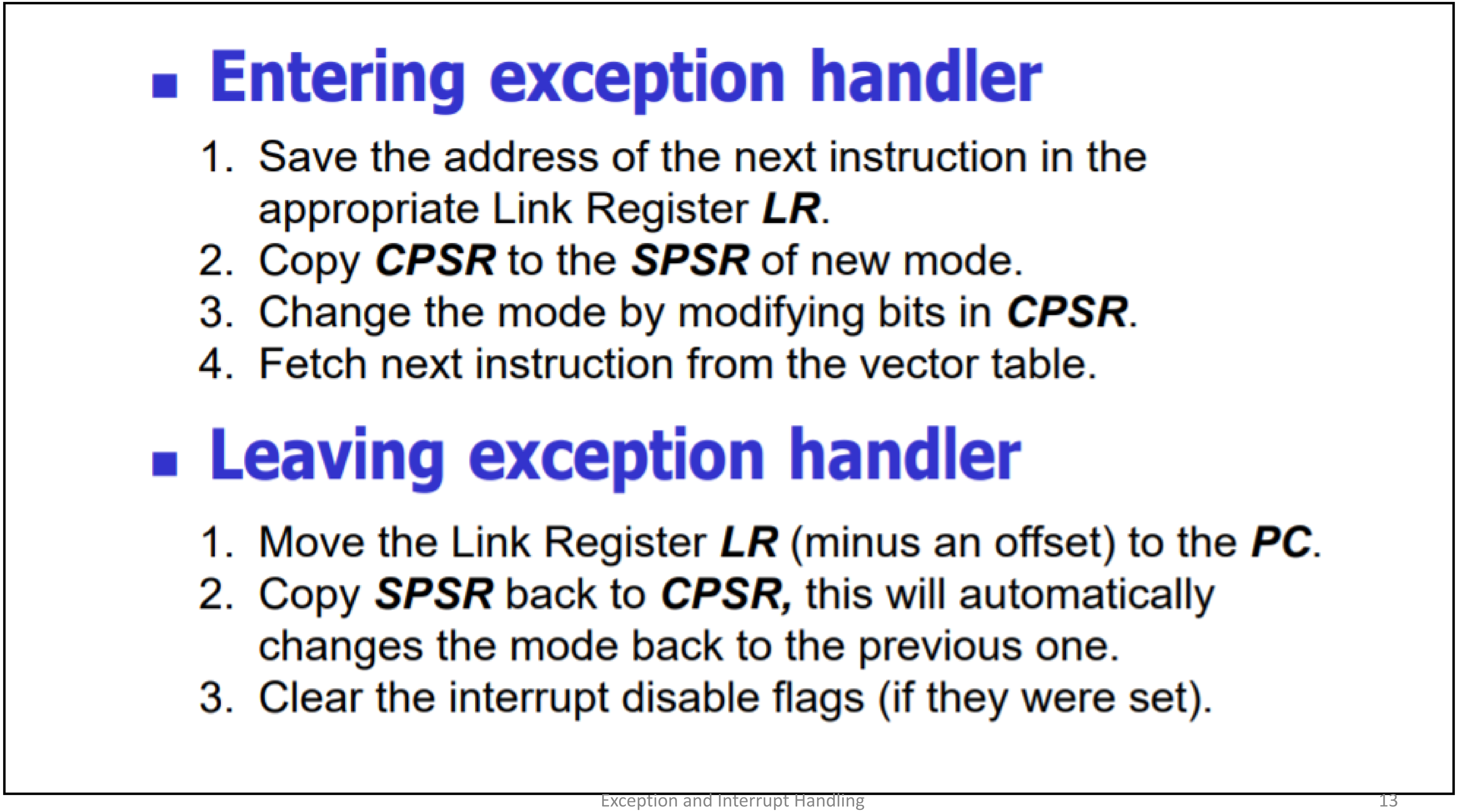
Exception	Returning Address
Reset	None
Data Abort	LR-8
FIQ, IRQ, prefetch Abort	LR-4
SWI, Undefined Instruction	LR

Link Register Offset



The link register is used to return the ***PC*** (after handling the exception) to the appropriate place in the interrupted task. It is modified based on the current ***PC*** value and the type of exception occurred. For some cases it should point to the next instruction after the exception handling is done and in some other cases it should return to one or 2 previous instructions to repeat those instructions after the exception handling is done. For example, in the case of IRQ exception, the link register is pointing initially to the last executed instruction + 8, so after the exception is handled we should return to the old ***PC*** value + 4 (next instruction) which equals to the old ***LR*** value – 4. Another example is the data abort exception, in this case when the exception is handled, the ***PC*** should point to the same instruction again to retry accessing the same memory location again.

```
0X00000000   LDR R0, =VALUE1          IC = FC + DC + EC
0X00000004   MOV R1, #0X0006          pc ⍰mem addr. of next instr.
0X00000008   ADDS R3, [R0], R1
```

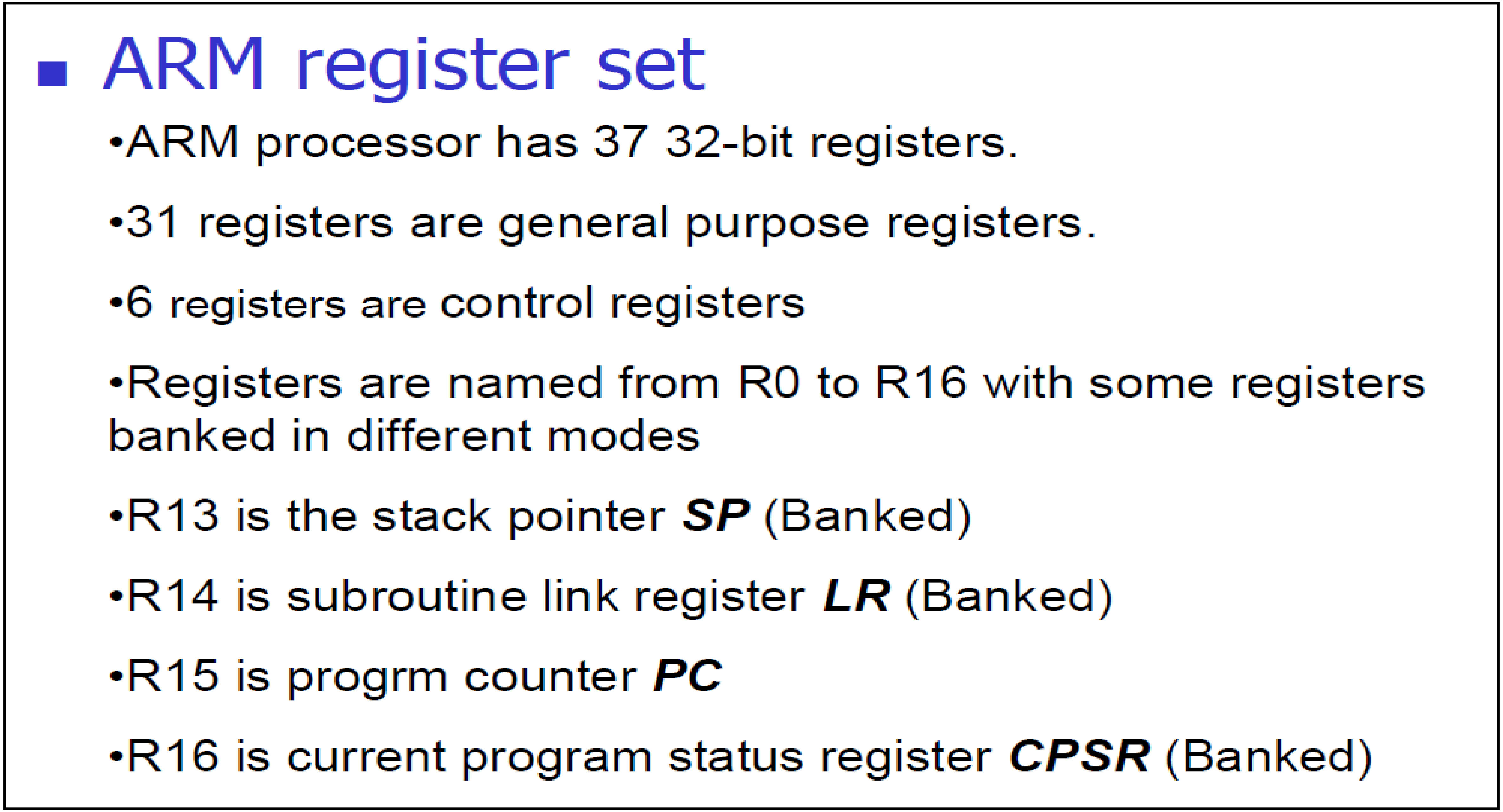


Entering exception handler

1. Save the address of the next instruction in the appropriate Link Register ***LR***.
2. Copy ***CPSR*** to the ***SPSR*** of new mode.
3. Change the mode by modifying bits in ***CPSR***.
4. Fetch next instruction from the vector table.

Leaving exception handler

1. Move the Link Register ***LR*** (minus an offset) to the ***PC***.
2. Copy ***SPSR*** back to ***CPSR,*** this will automatically changes the mode back to the previous one.
3. Clear the interrupt disable flags (if they were set).

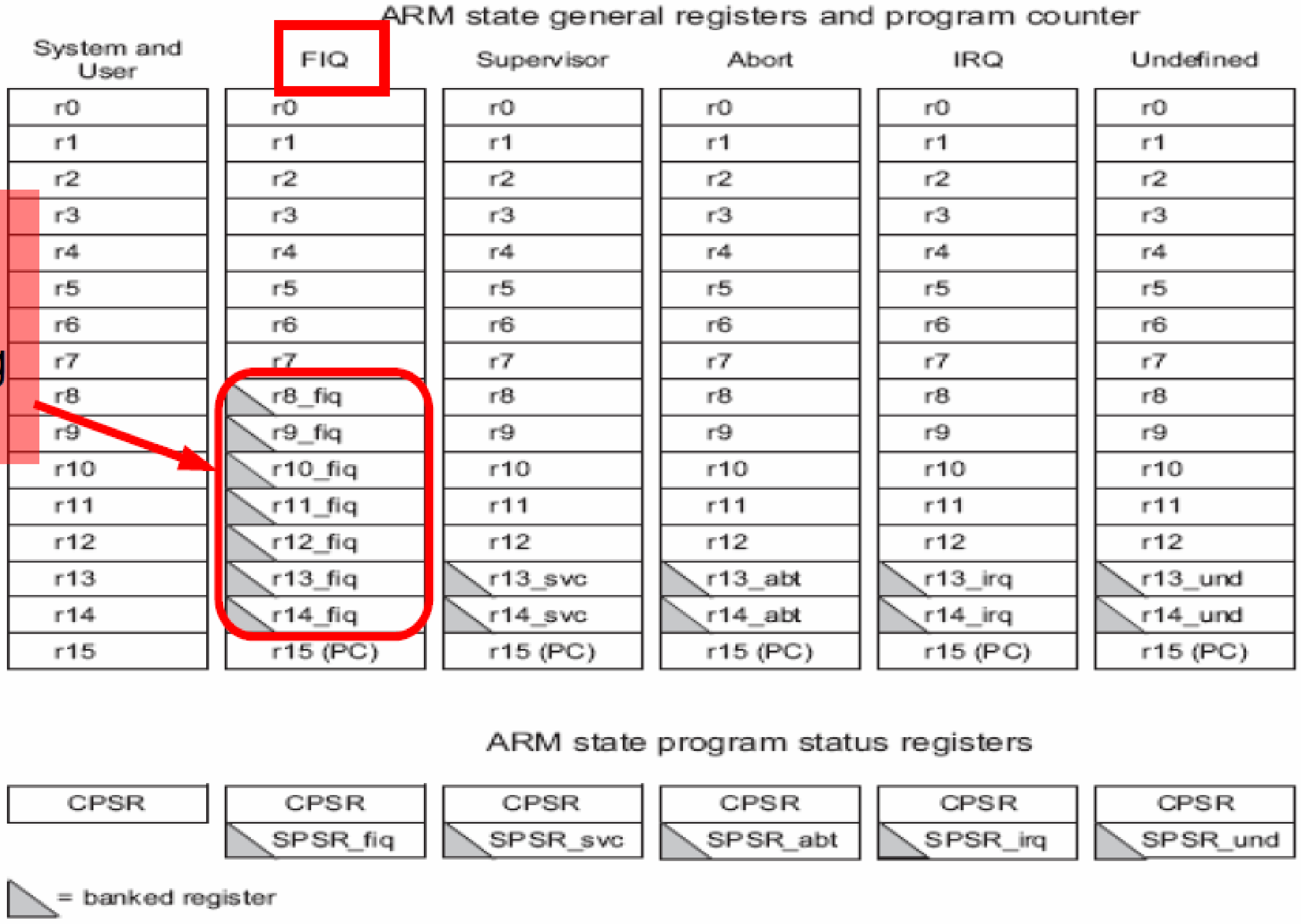


ARM register set

- ARM processor has 37 32-bit registers.
- 31 registers are general purpose registers.
- 6 registers are control registers
- Registers are named from R0 to R16 with some registers banked in different modes
- R13 is the stack pointer ***SP*** (Banked)
- R14 is subroutine link register ***LR*** (Banked)
- R15 is progrm counter ***PC***
- R16 is current program status register ***CPSR*** (Banked)

ARM register set

More banked registers, so context switching is faster



ARM state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	◺ r8_fiq	r8	r8	r8	r8
r9	◺ r9_fiq	r9	r9	r9	r9
r10	◺ r10_fiq	r10	r10	r10	r10
r11	◺ r11_fiq	r11	r11	r11	r11
r12	◺ r12_fiq	r12	r12	r12	r12
r13	◺ r13_fiq	◺ r13_svc	◺ r13_abt	◺ r13_irq	◺ r13_und
r14	◺ r14_fiq	◺ r14_svc	◺ r14_abt	◺ r14_irq	◺ r14_und
r15	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	◺ SPSR_fiq	◺ SPSR_svc	◺ SPSR_abt	◺ SPSR_irq	◺ SPSR_und

◺ = banked register

Different methods of returning from IRQ and FIQ exception handler

1. **use SUBS instr.**

```
 Handler
     <handler code>
        ......
     SUBS  pc, r14, #4 ; pc=r14-4
```

restore cpsr from spsr register

2. Subtracts offset from link register r14 at beginning of handler

```
handler
            SUB r14, r14, #4      ;r14=r14 – 4
             ......
             <handler code>
              ......
             MOVS  pc, r14
```

3. Use of interrupt to store the link register

i. subtract offset from link register

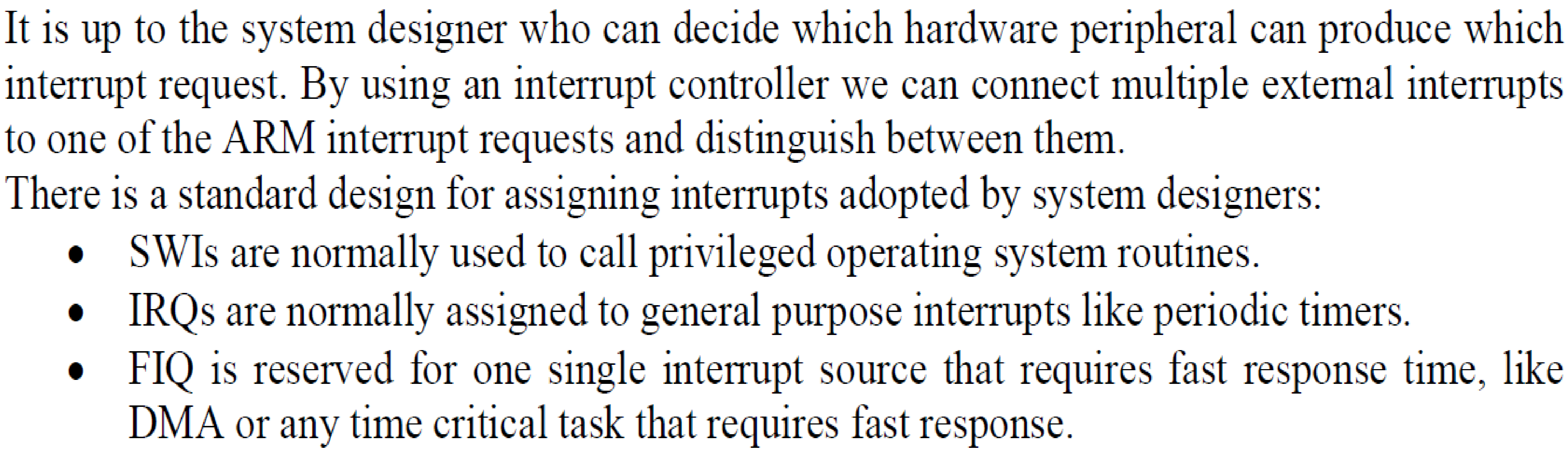
ii. Store it onto interrupt stack

```
handler
          SUB  r14, r14, #4   ; r14= r14 – 1
          STMFD  r13!, {r0-r3, r14}    ; store
context
          ....
          <handler code>
          ......
          LDMFD r13!, {r0-r3, pc}^    ;return
```

Interrupts

- Two types of Interrupts on ARM processor
  - Exception raised by external peripheral –namely IRQ ,FIQ
  - SWI – specific instruction that causes exceptions

- ## Assigning Interrupts



It is up to the system designer who can decide which hardware peripheral can produce which interrupt request. By using an interrupt controller we can connect multiple external interrupts to one of the ARM interrupt requests and distinguish between them.

There is a standard design for assigning interrupts adopted by system designers:

- SWIs are normally used to call privileged operating system routines.
- IRQs are normally assigned to general purpose interrupts like periodic timers.
- FIQ is reserved for one single interrupt source that requires fast response time, like DMA or any time critical task that requires fast response.

Interrupt latency

It is the interval of time from an external interrupt signal being raised to the first fetch of an instruction of the ISR of the raised interrupt signal.

System architects try to achieve two main goals:

- To handle multiple interrupts simultaneously.
- To minimize the interrupt latency.

And this can be done by 2 methods:

- allow nested interrupt handling
- give priorities to different interrupt sources

Nested Interrupts

IC=FC+ DC+ EC

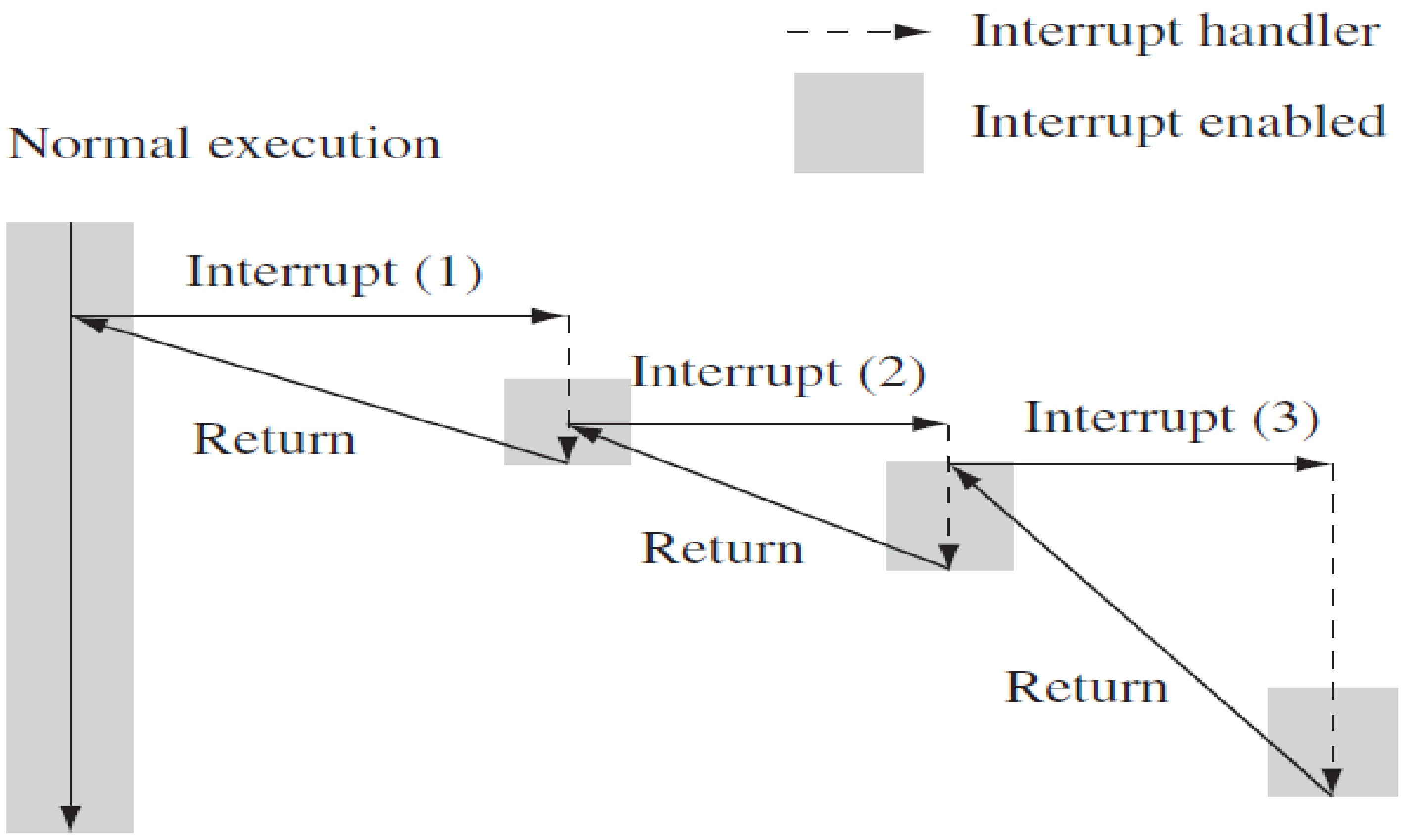
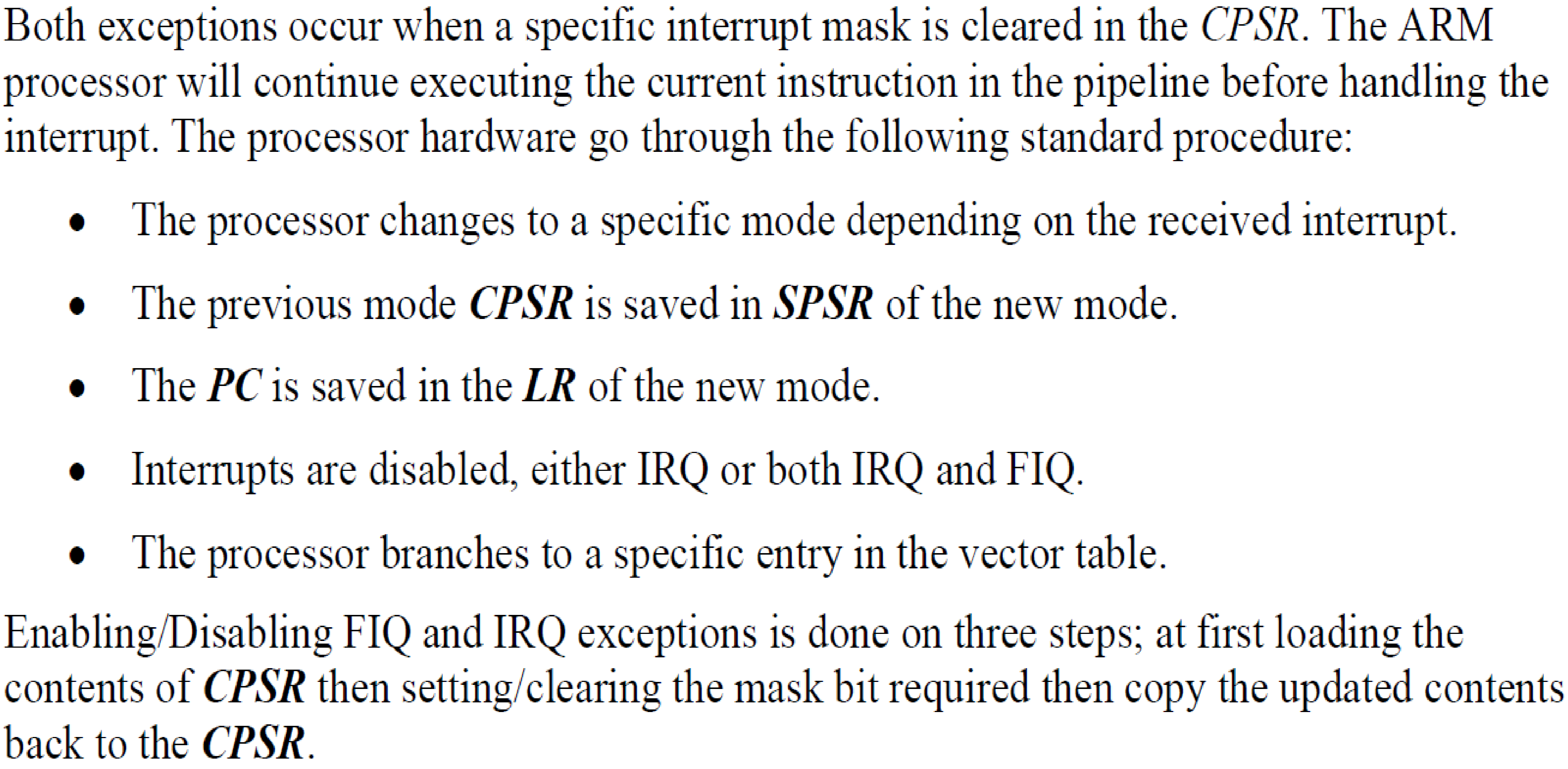


Figure 9.3 A three-level nested interrupt.

IRQ and FIQ Exceptions



Both exceptions occur when a specific interrupt mask is cleared in the *CPSR*. The ARM processor will continue executing the current instruction in the pipeline before handling the interrupt. The processor hardware go through the following standard procedure:

- The processor changes to a specific mode depending on the received interrupt.
- The previous mode ***CPSR*** is saved in ***SPSR*** of the new mode.
- The ***PC*** is saved in the ***LR*** of the new mode.
- Interrupts are disabled, either IRQ or both IRQ and FIQ.
- The processor branches to a specific entry in the vector table.

Enabling/Disabling FIQ and IRQ exceptions is done on three steps; at first loading the contents of ***CPSR*** then setting/clearing the mask bit required then copy the updated contents back to the ***CPSR***.

EXAMPLE 9.5

Figure 9.4 shows what happens when an IRQ exception is raised when the processor is in *user* mode. The processor starts in state 1. In this example both the IRQ and FIQ exception bits in the *cpsr* are enabled.

When an IRQ occurs the processor moves into state 2. This transition automatically sets the IRQ bit to one, disabling any further IRQ exceptions. The FIQ exception, however, remains enabled because FIQ has a higher priority and therefore does not get disabled when a low-priority IRQ exception is raised. The *cpsr* processor mode changes to *IRQ* mode. The *user* mode *cpsr* is automatically copied into *spsr_irq*.

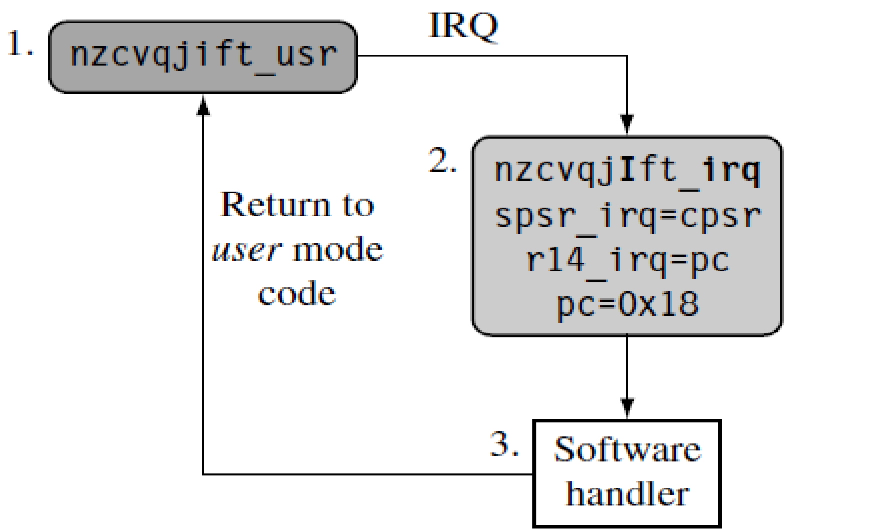


Figure 9.4 Interrupt Request (IRQ).

EXAMPLE 9.6

Figure 9.5 shows an example of an FIQ exception. The processor goes through a similar procedure as with the IRQ exception, but instead of just masking further IRQ exceptions from occurring, the processor also masks out further FIQ exceptions. This means that both interrupts are disabled when entering the software handler in state 3.

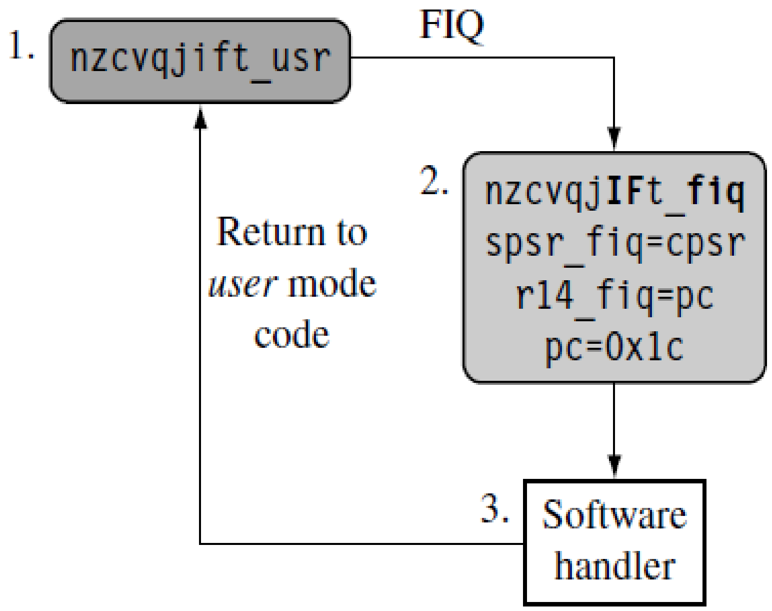
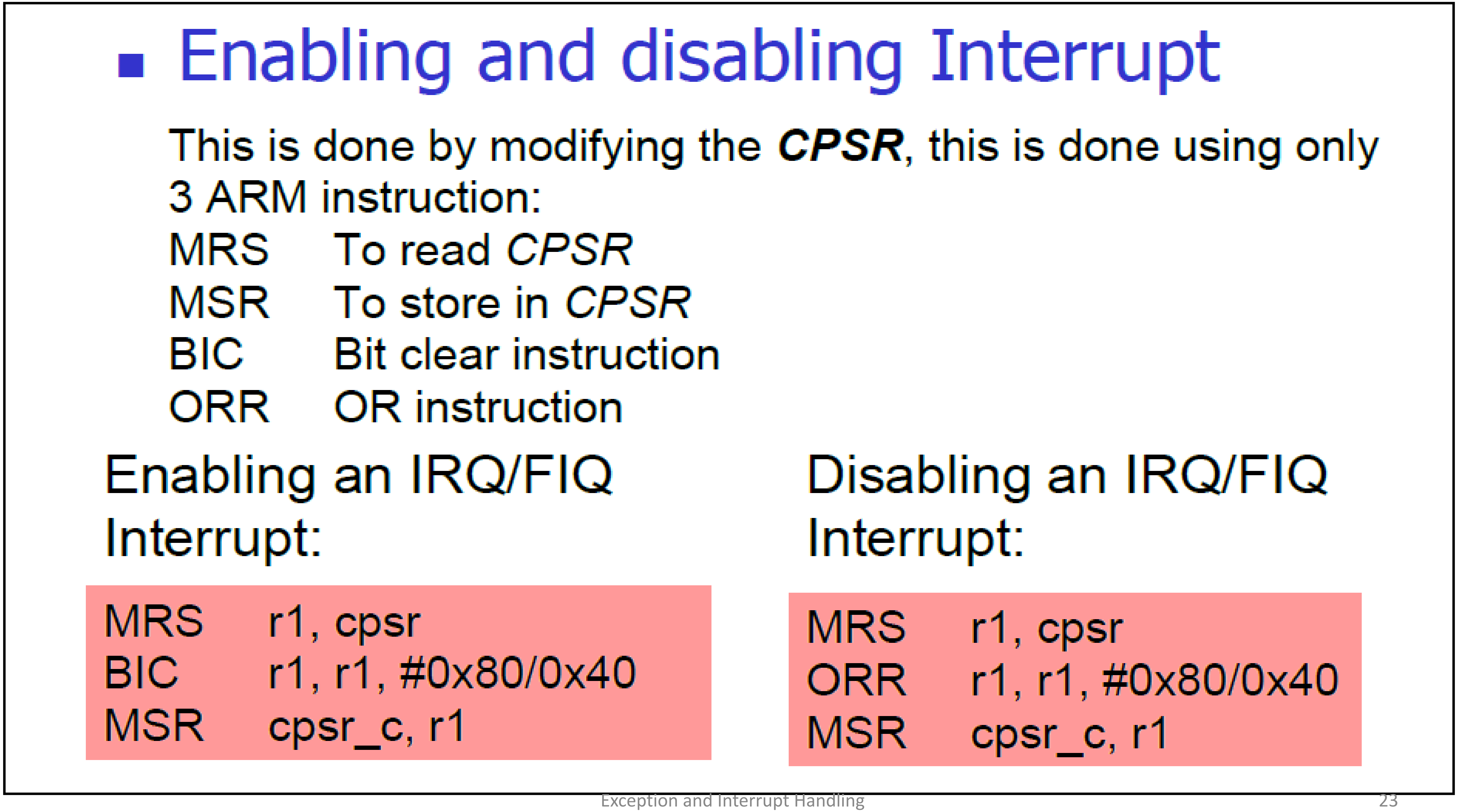


Figure 9.5 Fast Interrupt Request (FIQ).



Enabling and disabling Interrupt

This is done by modifying the ***CPSR***, this is done using only 3 ARM instruction:

MRS	To read *CPSR*
MSR	To store in *CPSR*
BIC	Bit clear instruction
ORR	OR instruction

Enabling an IRQ/FIQ Interrupt:

```
MRS     r1, cpsr
BIC     r1, r1, #0x80/0x40
MSR     cpsr_c, r1
```

Disabling an IRQ/FIQ Interrupt:

```
MRS     r1, cpsr
ORR     r1, r1, #0x80/0x40
MSR     cpsr_c, r1
```

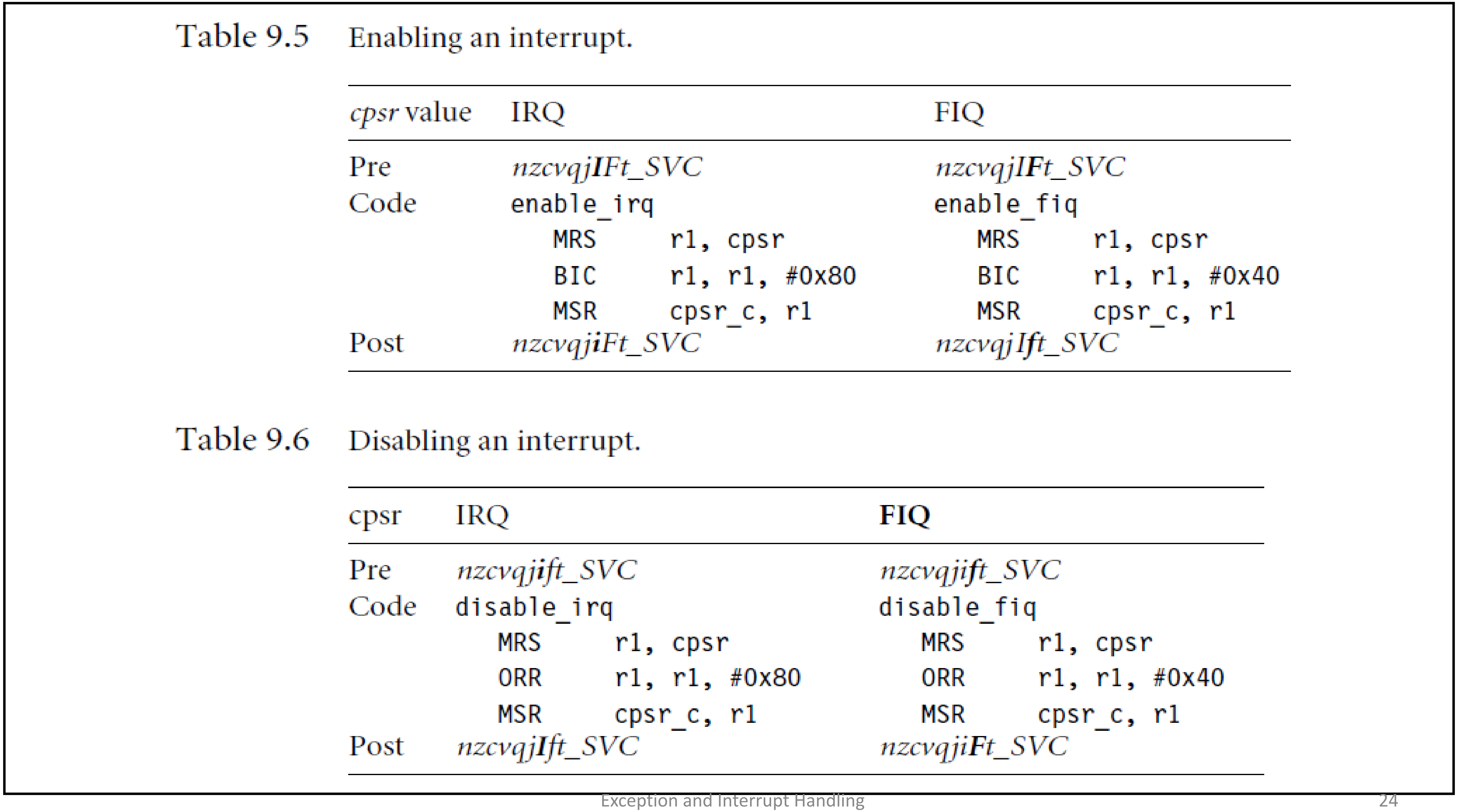


Table 9.5 Enabling an interrupt.

cpsr value	IRQ	FIQ
Pre	*nzcvqj**I**Ft_SVC*	*nzcvqjI**F**t_SVC*

Code	`enable_irq`
	`MRS r1, cpsr`
	`BIC r1, r1, #0x80`
	`MSR cpsr_c, r1`
		`enable_fiq`
		`MRS r1, cpsr`
		`BIC r1, r1, #0x40`
		`MSR cpsr_c, r1`

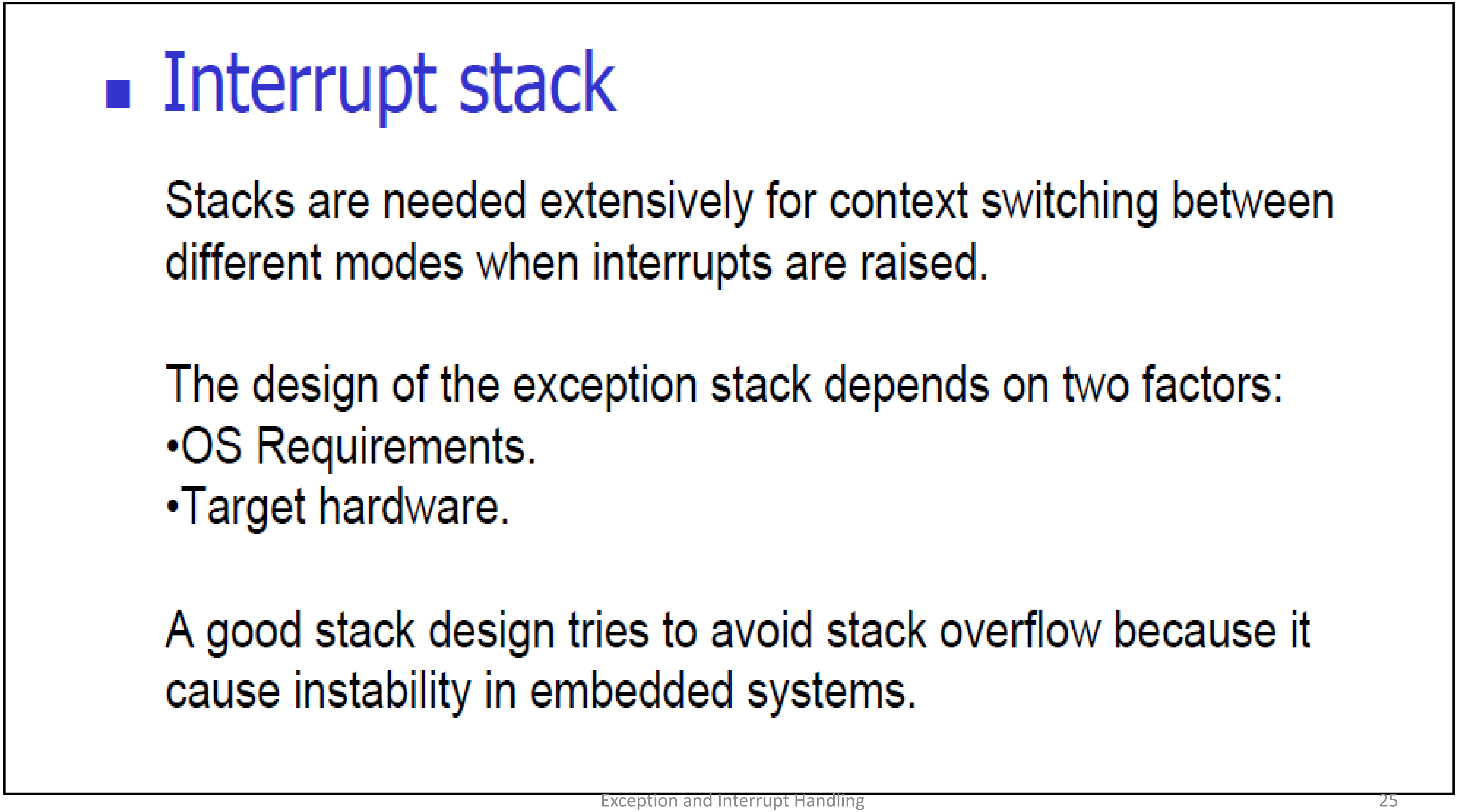
Post	*nzcvqj**i**Ft_SVC*	*nzcvqjI**f**t_SVC*

Table 9.6 Disabling an interrupt.

cpsr	IRQ	**FIQ**
Pre	*nzcvqj**i**ft_SVC*	*nzcvqji**f**t_SVC*

Code	`disable_irq`
	`MRS r1, cpsr`
	`ORR r1, r1, #0x80`
	`MSR cpsr_c, r1`
		`disable_fiq`
		`MRS r1, cpsr`
		`ORR r1, r1, #0x40`
		`MSR cpsr_c, r1`

Post	*nzcvqj**I**ft_SVC*	*nzcvqji**F**t_SVC*



Interrupt stack

Stacks are needed extensively for context switching between different modes when interrupts are raised.

The design of the exception stack depends on two factors:

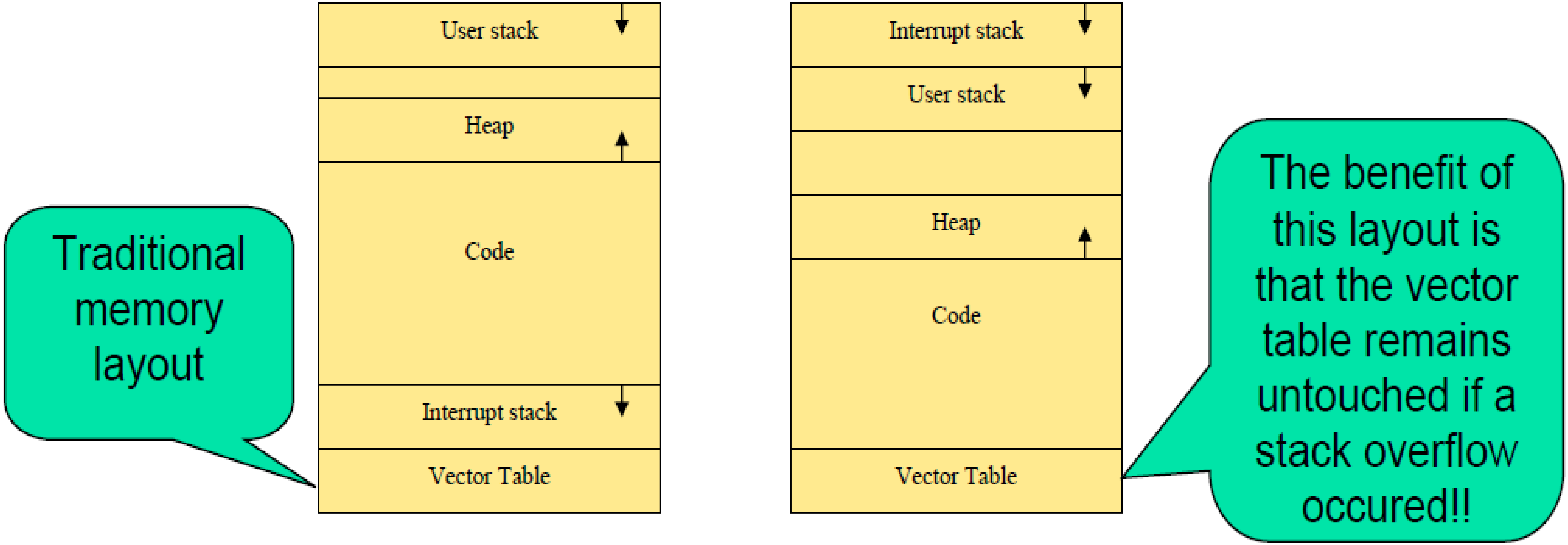
- OS Requirements.
- Target hardware.

A good stack design tries to avoid stack overflow because it cause instability in embedded systems.

Interrupt stack

Two design decisions need to be made for the stacks:

- The location
- The size



Traditional memory layout

The benefit of this layout is that the vector table remains untouched if a stack overflow occured!!

Interrupt Handling Schemes

1. Non-nested Interrupt Handler
2. Nested Interrupt Handler
3. Re-entrant Interrupt Handler
4. Prioritized simple Interrupt Handler

Interrupt Handling Schemes:

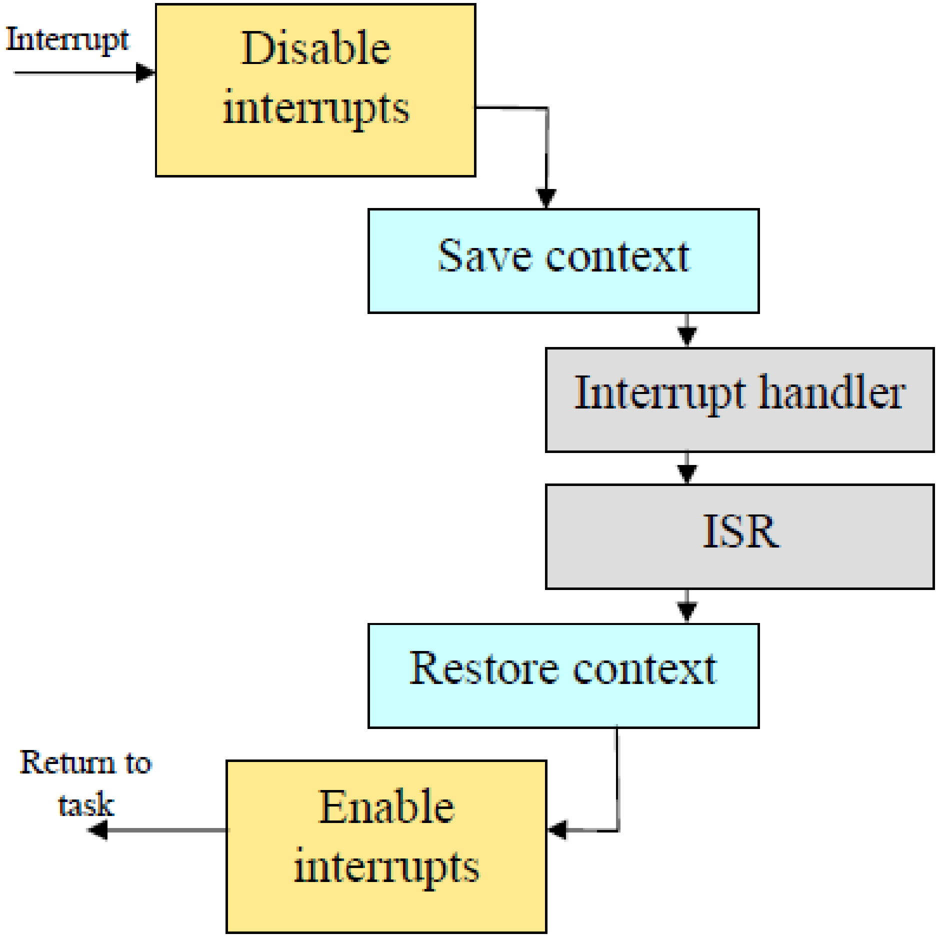
▪ Non-nested interrupt handling scheme

- This is the simplest interrupt handler.

- Interrupts are disabled until control is returned back to the interrupted task.

- One interrupt can be served at a time.

- Not suitable for complex embedded systems.



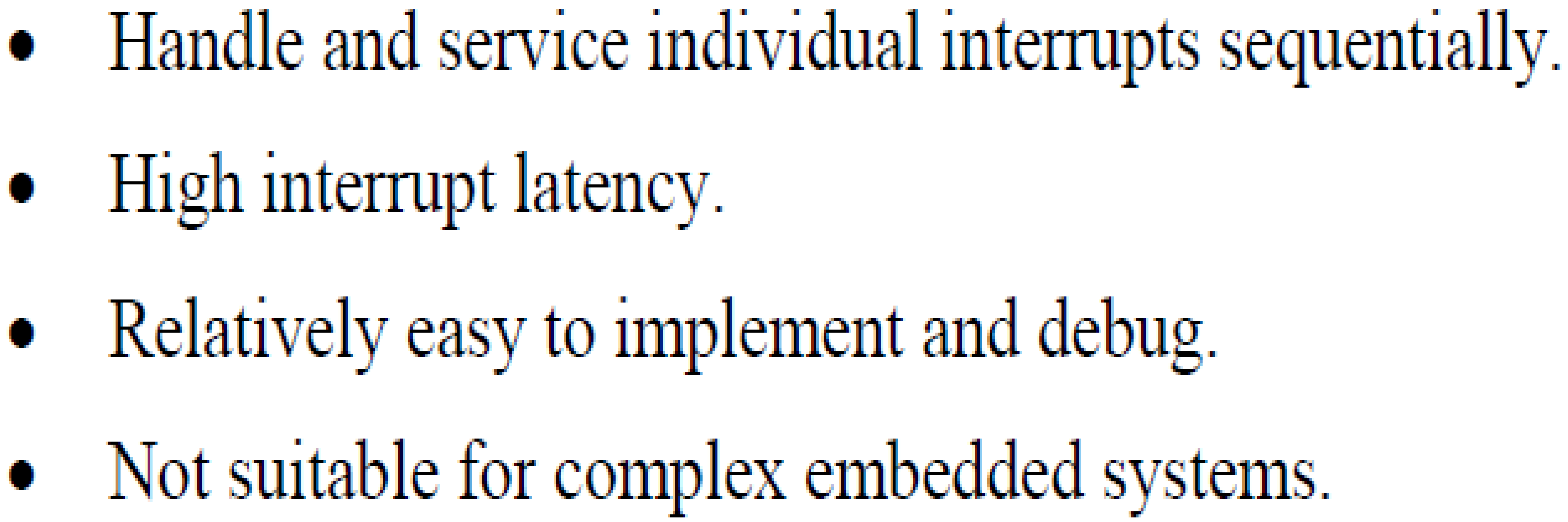
Interrupt Handling Schemes:
1. Non-nested Interrupt handling

This is the simplest interrupt handler. Interrupts are disabled until control is returned back to the interrupted task. So only one interrupt can be served at a time and that is why this scheme is not suitable for complex embedded systems which most probably have more than one interrupt source and require concurrent handling. Figure 5 shows the steps taken to handle an interrupt:

Initially interrupts are disabled, When IRQ exception is raised and the ARM processor disables further IRQ exceptions from occurring. The mode is changed to the new mode depending on the raised exception. The register ***CPSR*** is copied to the ***SPSR*** of the new mode. Then the ***PC*** is set to the correct entry in the vector table and the instruction there will direct the ***PC*** to the appropriate handler. Then the context of the current task is saved a subset of the current mode non banked register. Then the interrupt handler executes some code to identify the interrupt source and decide which ISR will be called. Then the appropriate ISR is called. And finally the context of the interrupted task is restored, interrupts are enabled again and the control is returned to the interrupted task.

Interrupt Handling Schemes:
1. Non-nested Interrupt handling

- Summary:



- Handle and service individual interrupts sequentially.
- High interrupt latency.
- Relatively easy to implement and debug.
- Not suitable for complex embedded systems.

Interrupt Handling Schemes: contd.

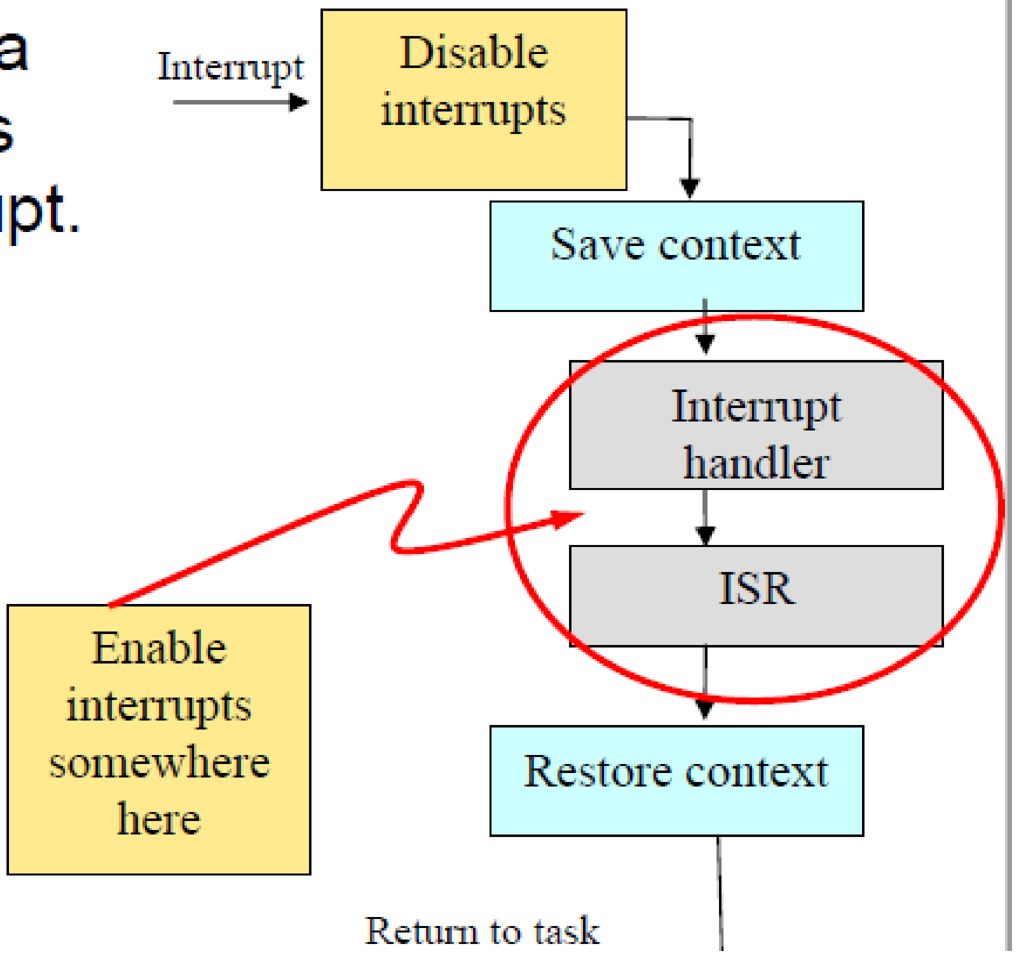
▪ Nested interrupt handling scheme(1)

•Handling more than one interrupt at a time is possible by enabling interrupts before fully serving the current interrupt.

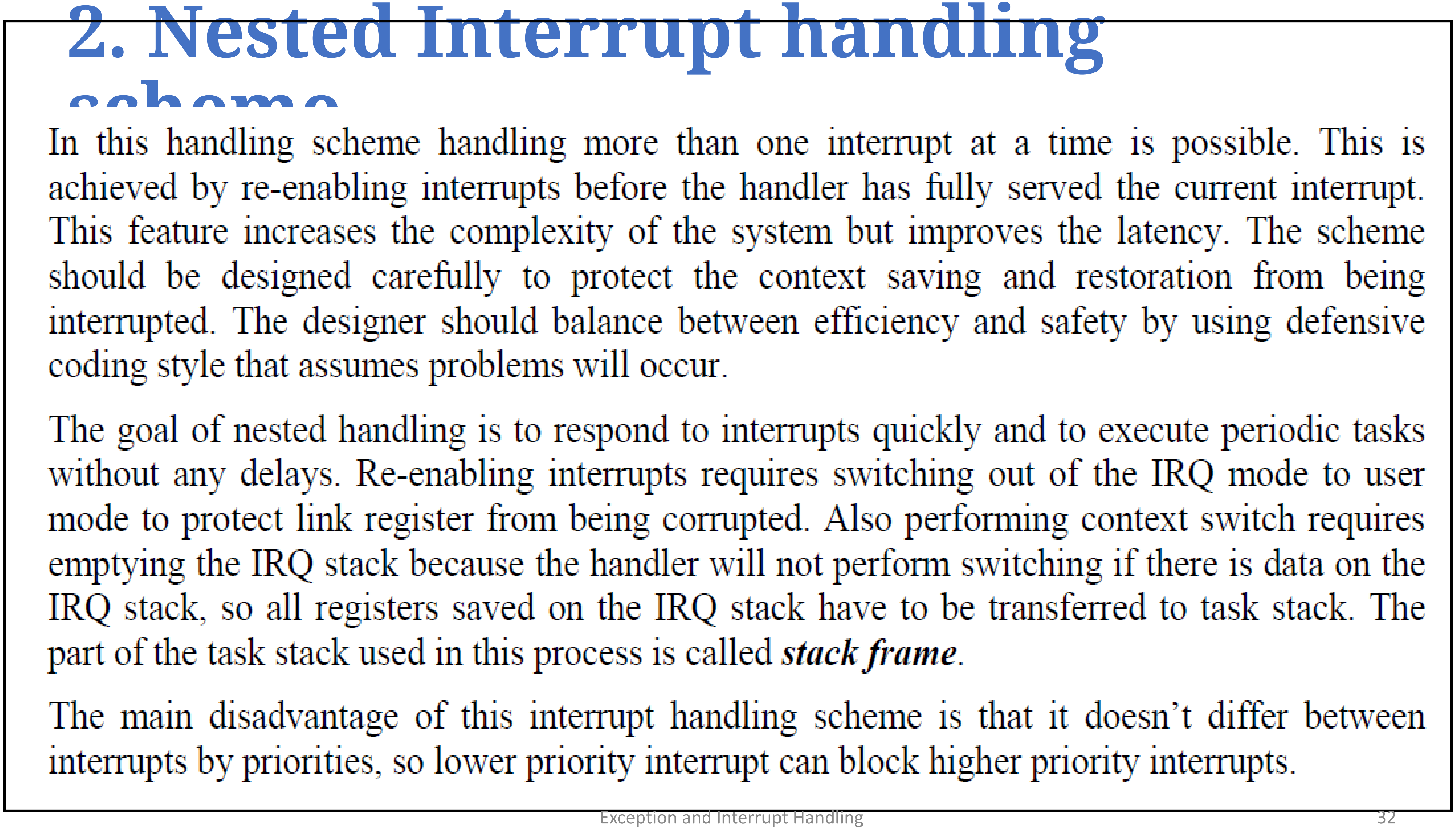
•Latency is improved.

•System is more complex.

•No difference between interrupts by priorities, so normal interrupts can block critical interrupts.



2. Nested Interrupt handling scheme

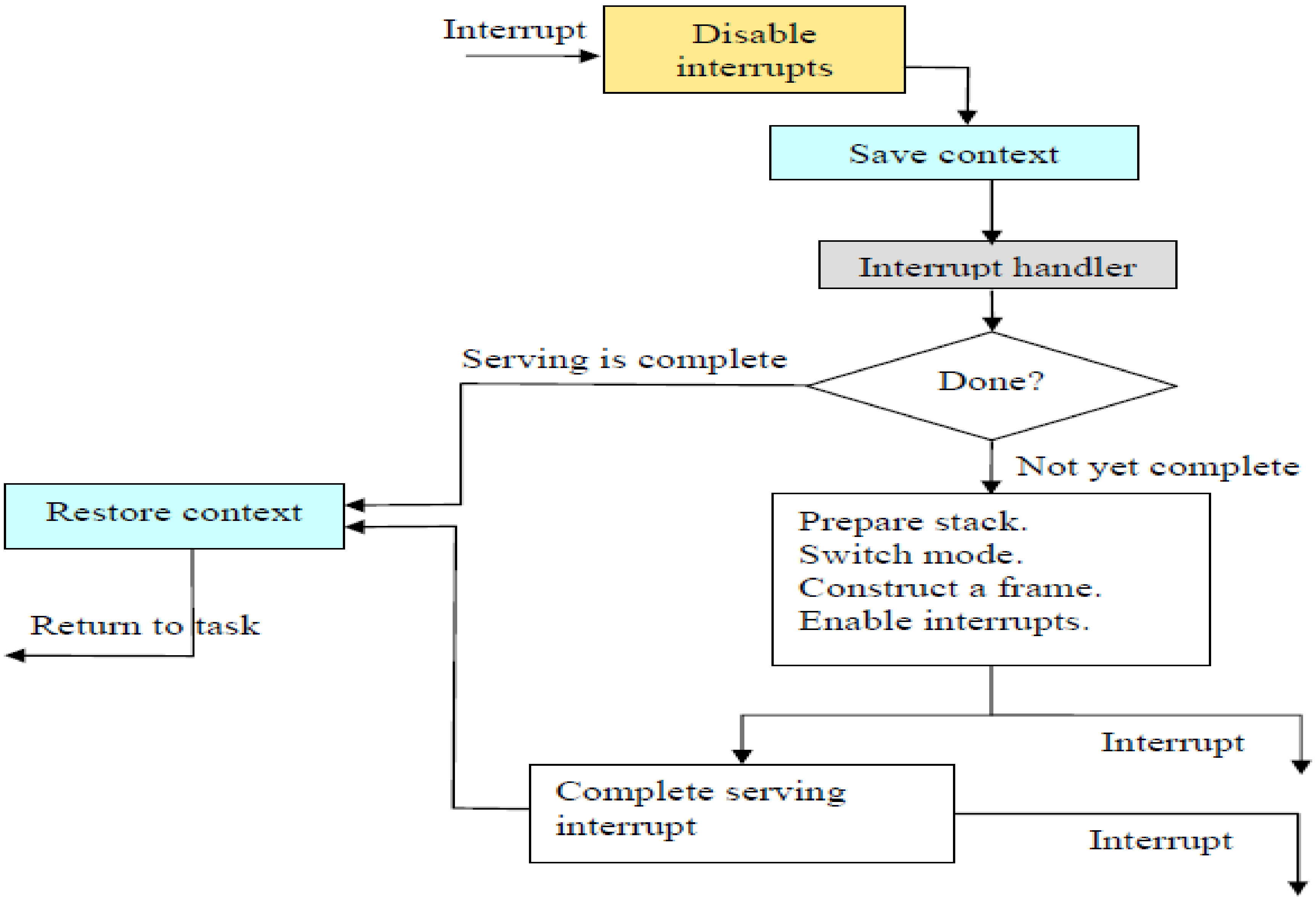


In this handling scheme handling more than one interrupt at a time is possible. This is achieved by re-enabling interrupts before the handler has fully served the current interrupt. This feature increases the complexity of the system but improves the latency. The scheme should be designed carefully to protect the context saving and restoration from being interrupted. The designer should balance between efficiency and safety by using defensive coding style that assumes problems will occur.

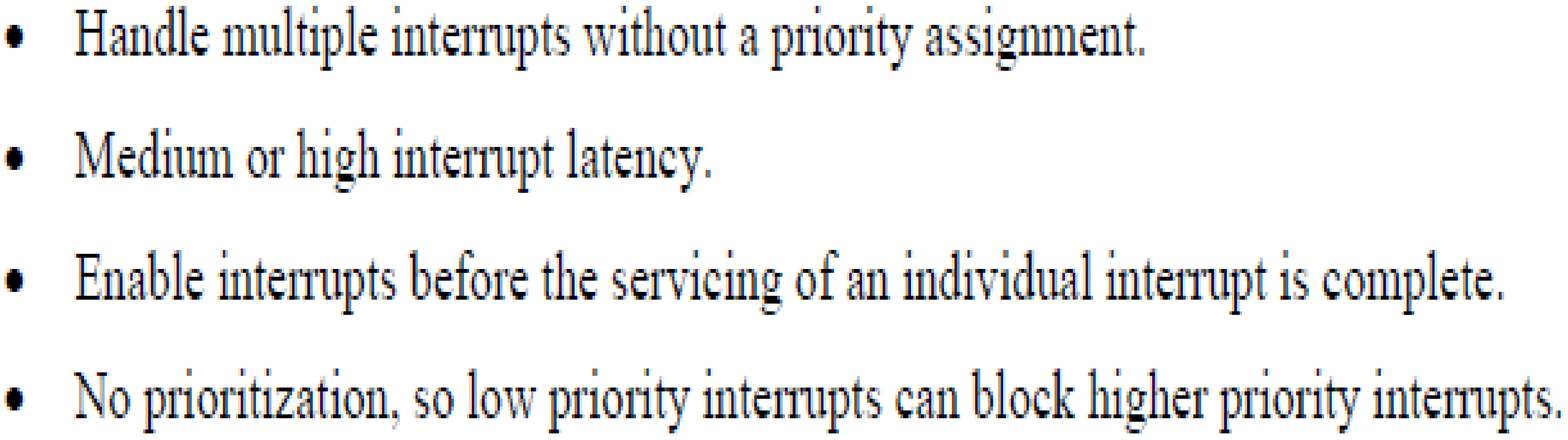
The goal of nested handling is to respond to interrupts quickly and to execute periodic tasks without any delays. Re-enabling interrupts requires switching out of the IRQ mode to user mode to protect link register from being corrupted. Also performing context switch requires emptying the IRQ stack because the handler will not perform switching if there is data on the IRQ stack, so all registers saved on the IRQ stack have to be transferred to task stack. The part of the task stack used in this process is called ***stack frame***.

The main disadvantage of this interrupt handling scheme is that it doesn't differ between interrupts by priorities, so lower priority interrupt can block higher priority interrupts.

2. Nested Interrup handling scheme



Nested Interrupt Summary



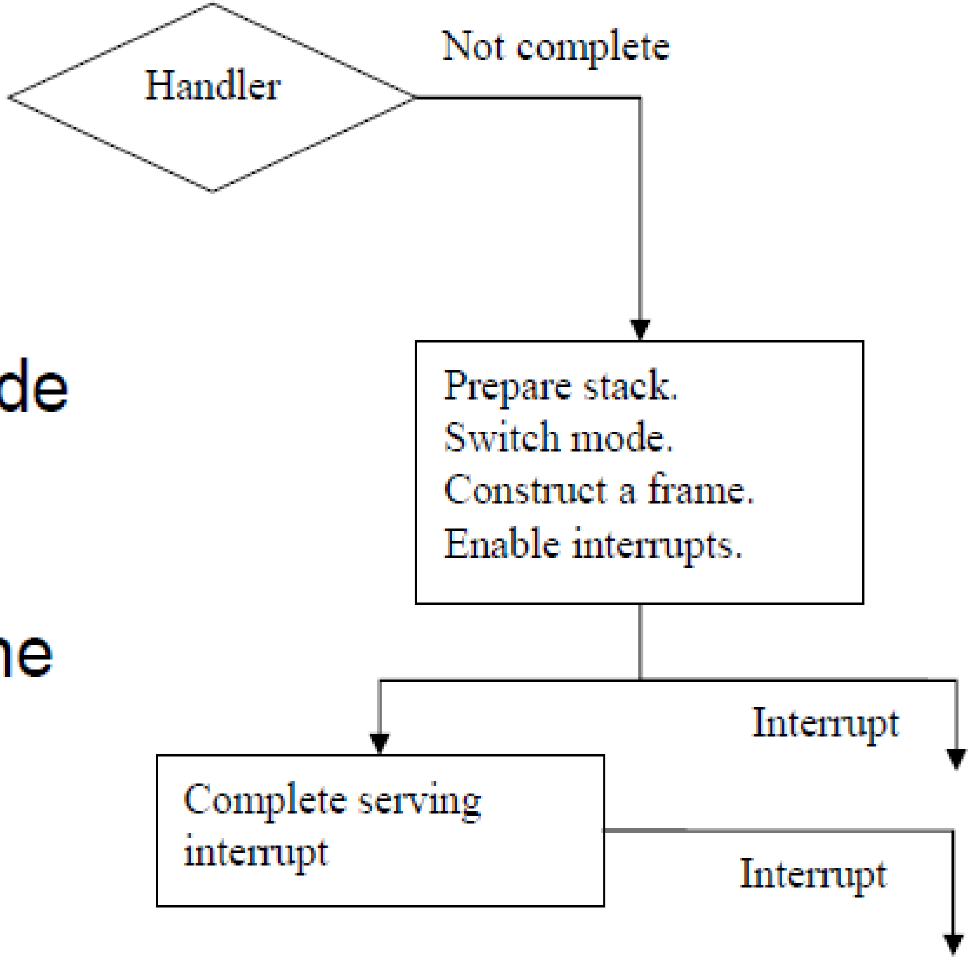
- Handle multiple interrupts without a priority assignment.
- Medium or high interrupt latency.
- Enable interrupts before the servicing of an individual interrupt is complete.
- No prioritization, so low priority interrupts can block higher priority interrupts.

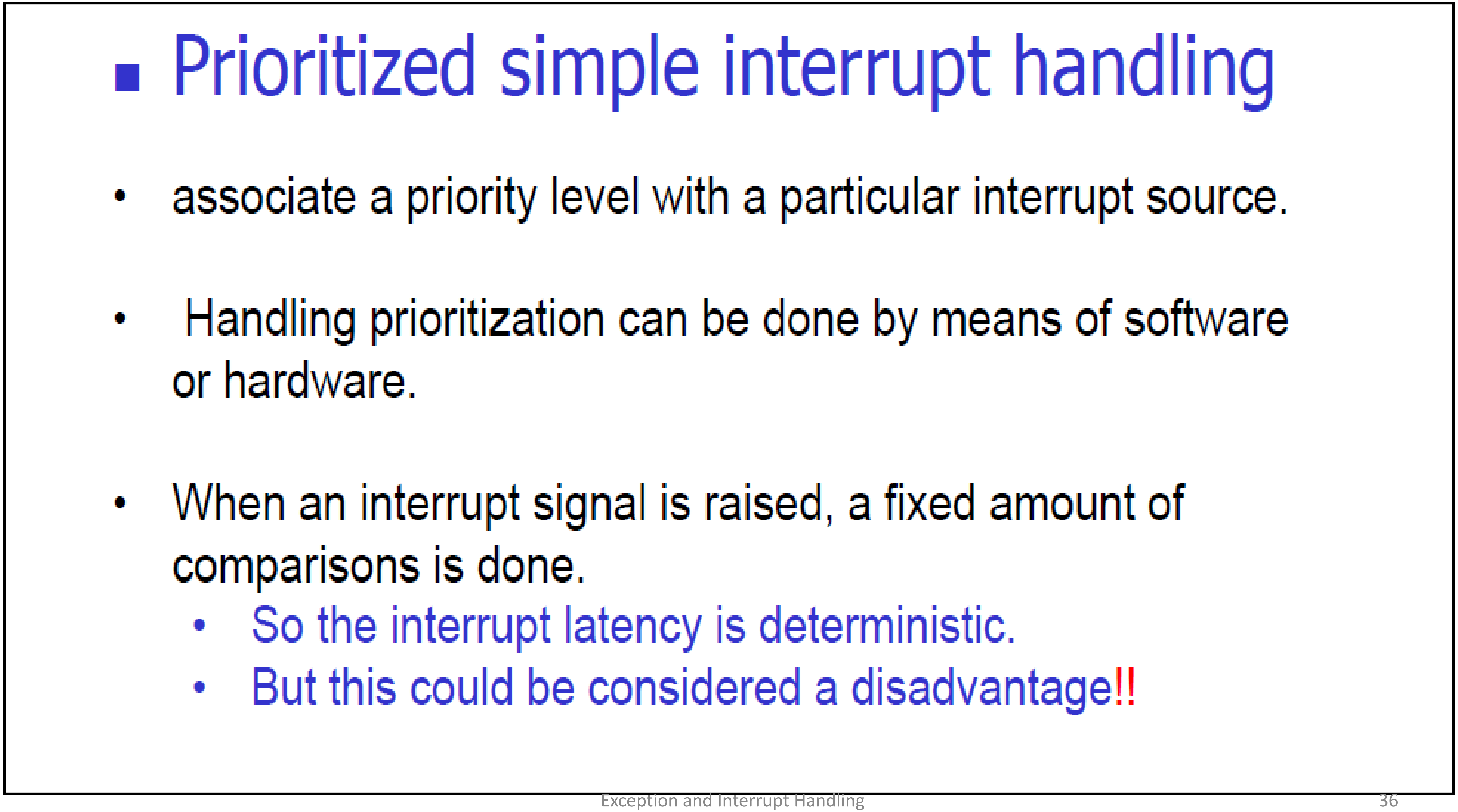
Nested interrupt handling scheme(2)

- The handler tests a flag that is updated by the ISR

- Re enabling interrupts requires switching out of current interrupt mode to either SVC or system mode.

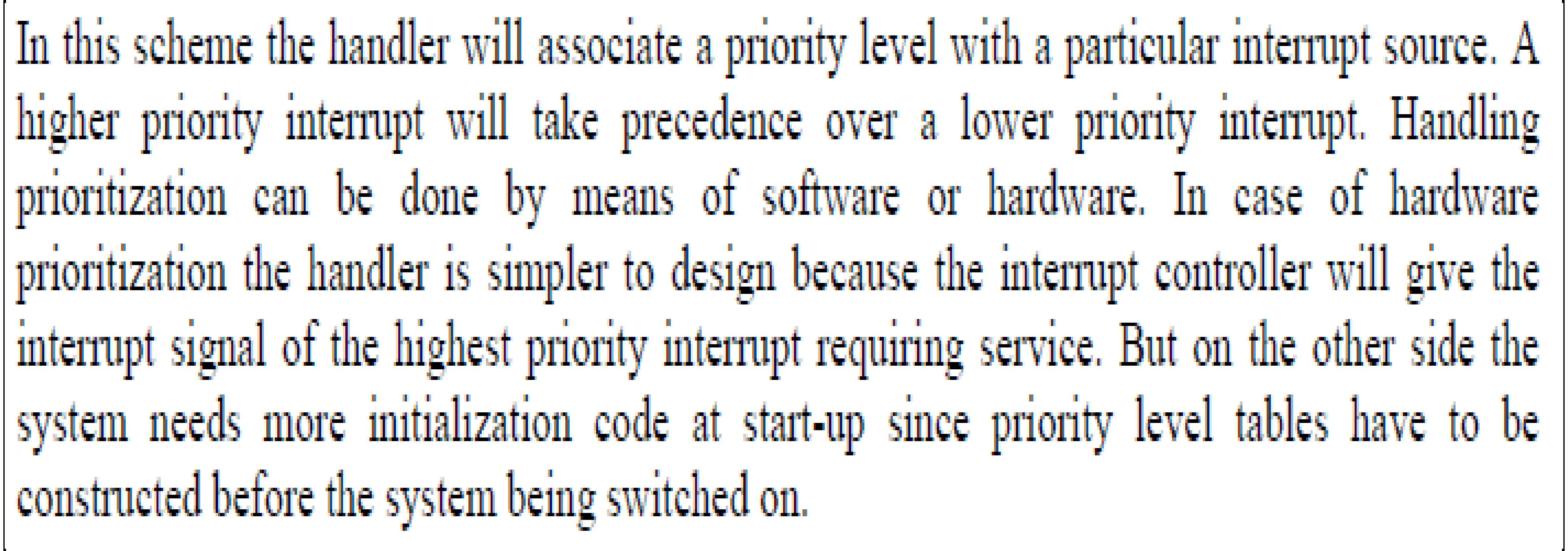
- Context switch involves emptying the IRQ stack into reserved blocks of memory on SVC stack called stack frames.



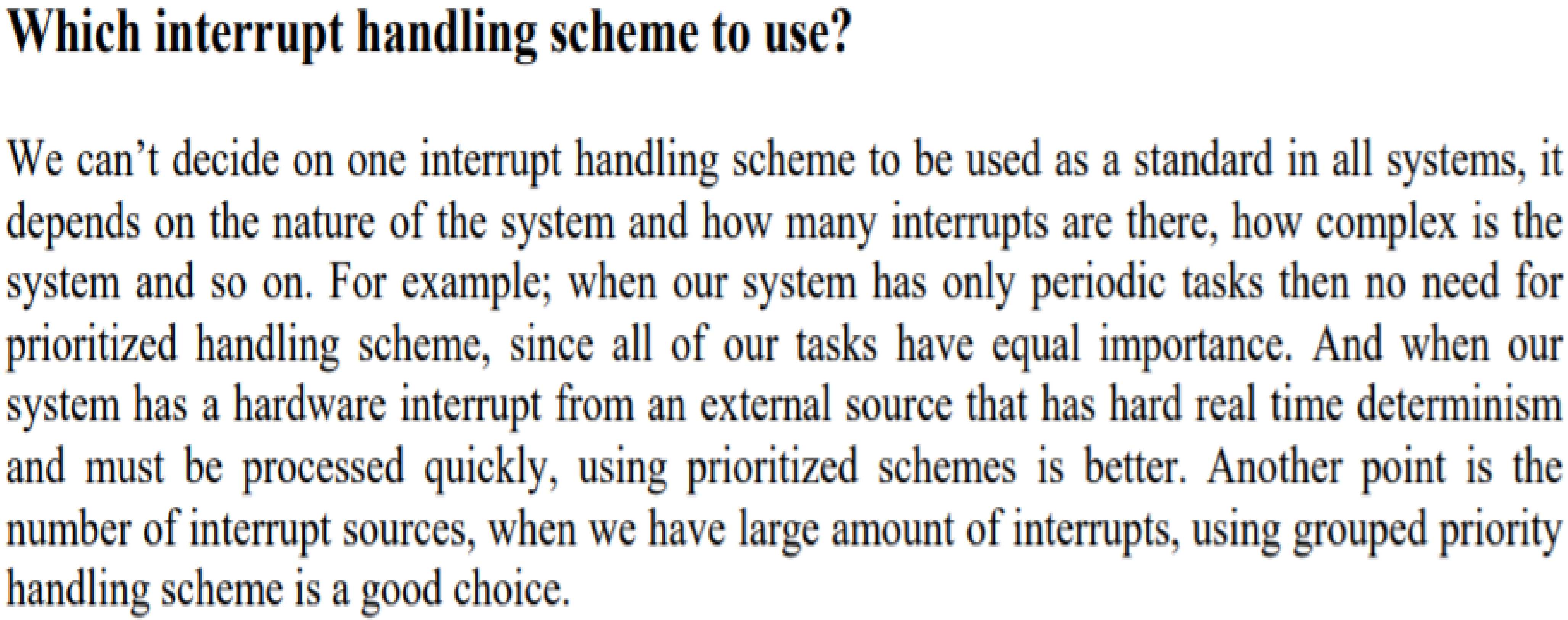


Prioritized simple interrupt handling

- associate a priority level with a particular interrupt source.
- Handling prioritization can be done by means of software or hardware.
- When an interrupt signal is raised, a fixed amount of comparisons is done.
  - So the interrupt latency is deterministic.
  - But this could be considered a disadvantage!!



In this scheme the handler will associate a priority level with a particular interrupt source. A higher priority interrupt will take precedence over a lower priority interrupt. Handling prioritization can be done by means of software or hardware. In case of hardware prioritization the handler is simpler to design because the interrupt controller will give the interrupt signal of the highest priority interrupt requiring service. But on the other side the system needs more initialization code at start-up since priority level tables have to be constructed before the system being switched on.



Which interrupt handling scheme to use?

We can’t decide on one interrupt handling scheme to be used as a standard in all systems, it depends on the nature of the system and how many interrupts are there, how complex is the system and so on. For example; when our system has only periodic tasks then no need for prioritized handling scheme, since all of our tasks have equal importance. And when our system has a hardware interrupt from an external source that has hard real time determinism and must be processed quickly, using prioritized schemes is better. Another point is the number of interrupt sources, when we have large amount of interrupts, using grouped priority handling scheme is a good choice.