Exceptions and Interrupts

Copyright (c) 2021 - 2014 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

Introduction to ARM Cortex-M Microcontrollers
– Embedded Systems, Jonathan W. Valvano

Digital Design and Computer Architecture,
D. M. Harris and S. L. Harris

ARM assembler in Raspberry Pi
Roger Ferrer Ibáñez

https://thinkingeek.com/arm-assembler-raspberry-pi/

Status Reg to General Reg Transfer Instructions

Status Register to General Register Transfer Instructions

MRS {<cond>} Rd, CPSR | SPSR

MRS	Rd, CPSR
MRS	Rd, SPSR
MRS <cond>	Rd, CPSR
MRS <cond>	Rd, SPSR

M R ⬅ S

General Reg to Status Reg Transfer Instructions

General Register to Status Register Transfer Instructions

MSR {<cond>} CPSR_f | SPSR_f, #<32-bit immediate>

MSR {<cond>} CPSR_<field> | SPSR_<field>, Rm

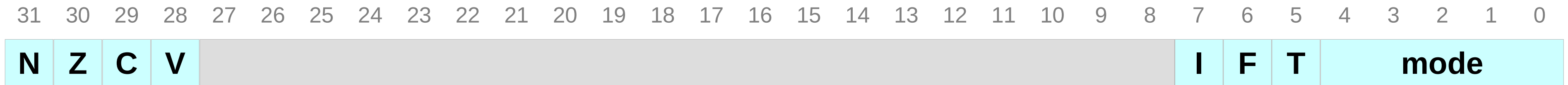
_<field> is one of

_c : the control field	PSR[7: 0]	
_x : the extension field	PSR[15: 8]	(unused on current ARMs)
_s : the status field	PSR[23:16]	(unused on current ARMs)
_f : the flag field	PSR[31:24]	

```
MSR          CPSR_f, #<32-bit immediate>
MSR          SPSR_f, #<32-bit immediate>
MSR <cond>   CPSR_f, #<32-bit immediate>
MSR <cond>   SPSR_f, #<32-bit immediate>
MSR          CPSR_<field>, Rm
MSR          SPSR_<field>, Rm
MSR <cond>   CPSR_<field>, Rm
MSR <cond>   SPSR_<field>, Rm
```

M S ⬅ R

CPSR and SPSR



Current Program Status Register (CPSR)

Saved Program Status Register (SPSR)

N Negative flag

Z Zero flag

C Carry flag

V Overflow flag

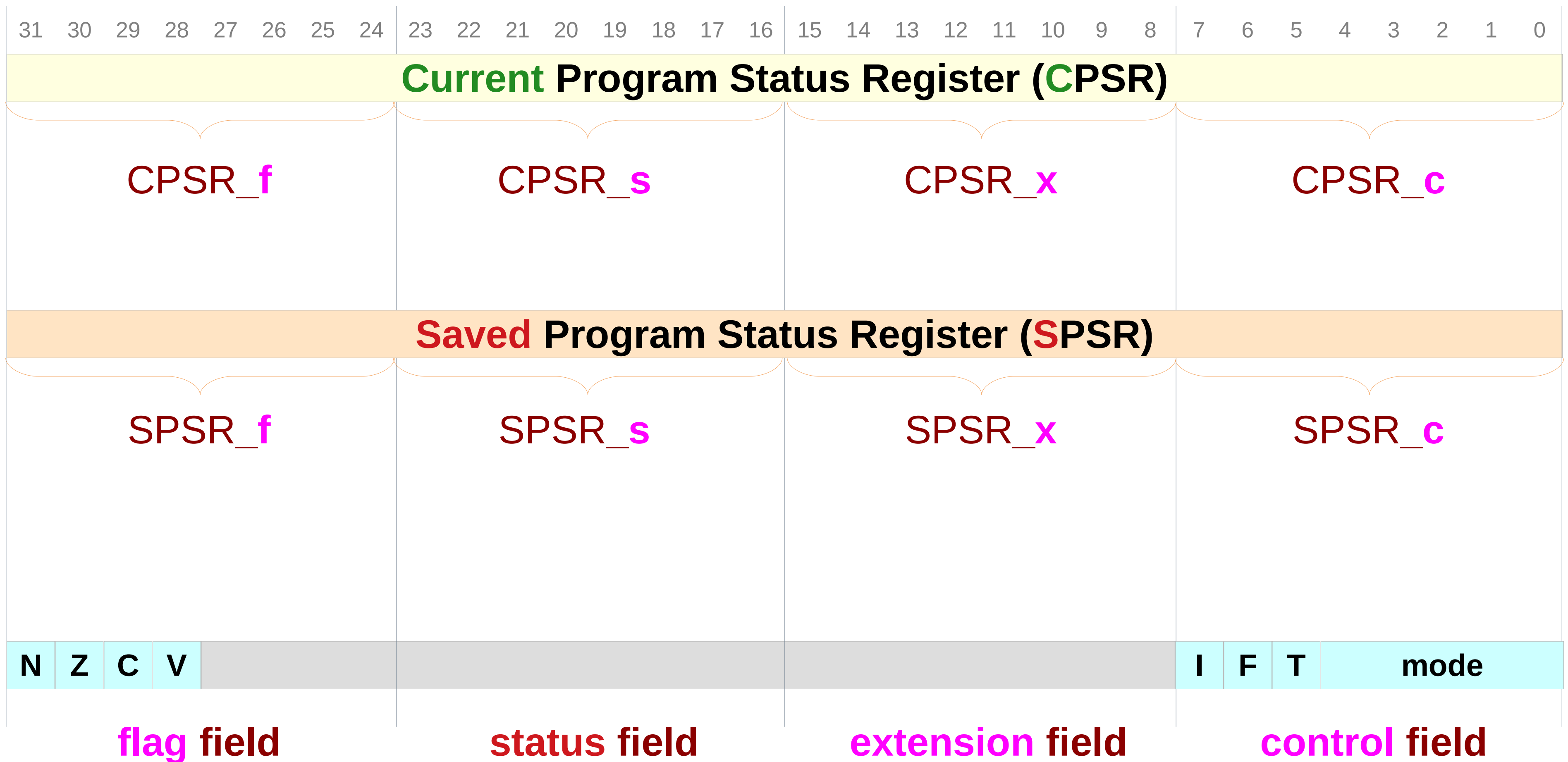
To disable Interrupt (IRQ), set **I**

To disable Fast Interrupt (FIQ), set **F**

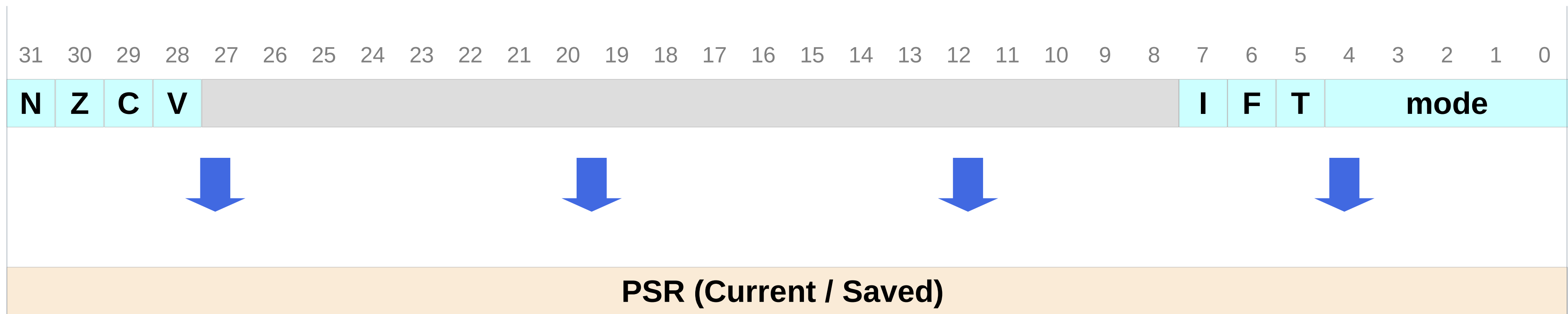
the **T** bit shows running in the Thumb state

Mode					
Usr (usr)	**1**	**0**	**0**	**0**	**0**
Fast Interrupt (fiq)	**1**	**0**	**0**	**0**	**1**
Interrupt (irq)	**1**	**0**	**0**	**1**	**0**
Supervisor (svc)	**1**	**0**	**0**	**1**	**1**
Abort (abt)	**1**	**0**	**1**	**1**	**1**
Undefined (und)	**1**	**1**	**0**	**1**	**1**
System (sys)	**1**	**1**	**1**	**1**	**1**

CPSR and SPSR Fields



To a General Reg From a Status Reg

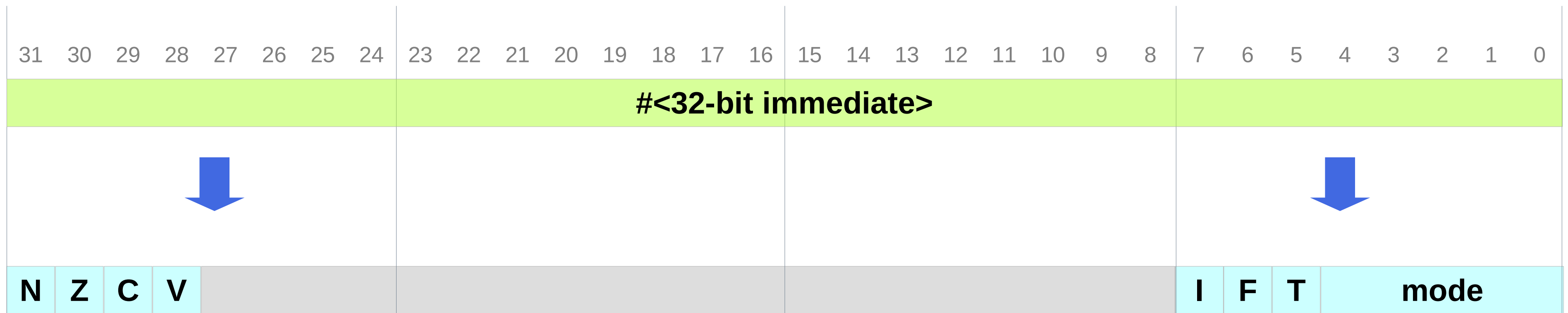


MRS Rd, CPSR
MRS Rd, SPSR

M R ⬅ S

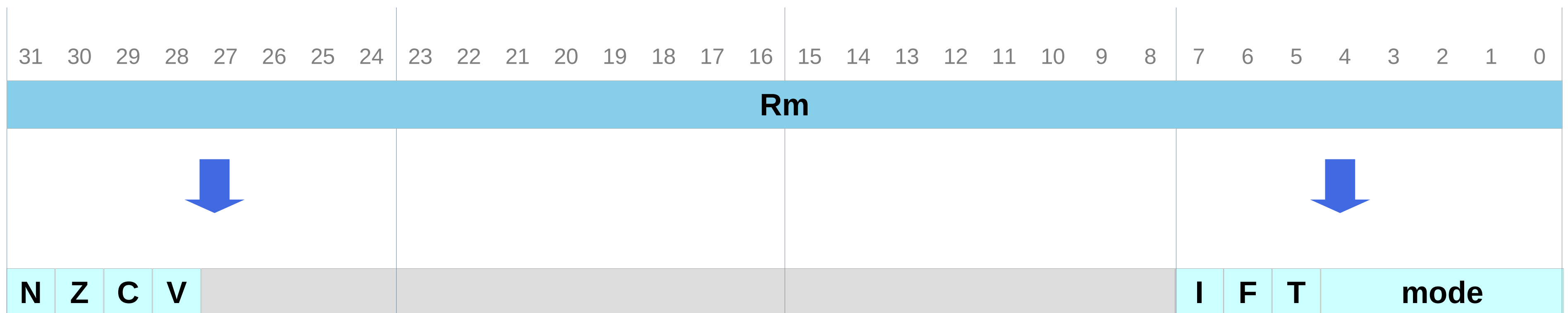
M S ⬅ R

To a Status Reg From a General Reg



MSR CPSR_f , #<32-bit immediate>
MSR SPSR_f , #<32-bit immediate>

MSR CPSR_c , #<32-bit imm>
MSR SPSR_c , #<32-bit imm>



MSR CPSR_f , Rm
MSR SPSR_f , Rm

MSR CPSR_c , Rm
MSR SPSR_c , Rm

Interrupt is an Exception

exceptions

There are four classes of **exception**:

- **interrupt**
- **trap**
- **fault**
- **abort**

Interrupt is one of the classes of **exception**.

Interrupt occurs asynchronously and
it is triggered by signal which is from I/O device
that are external by processor.

After exception handler finish
handling this interrupt (exception processing),
handler will always return to next instruction.

https://stackoverflow.com/questions/7295936/what-is-the-difference-between-interrupt-and-exception-context

Exceptions vs interrupts (1)

Interrupts and exceptions both alter the program flow.

- **interrupts** are used to handle external events (serial ports, keyboard)
- **exceptions** are used to handle instruction faults (division by zero, undefined opcode).

interrupts are handled by the processor after finishing the current instruction.

If it finds a signal on its interrupt pin,
it will look up the address of the interrupt handler
in the interrupt table and pass that routine control.

After returning from the interrupt handler routine,
it will resume program execution at the next instruction
after the interrupted instruction.

https://stackoverflow.com/questions/7295936/what-is-the-difference-between-interrupt-and-exception-context

Exceptions vs interrupts (2)

Exceptions on the other hand
are divided into three kinds.
Faults, **Traps** and **Aborts**.

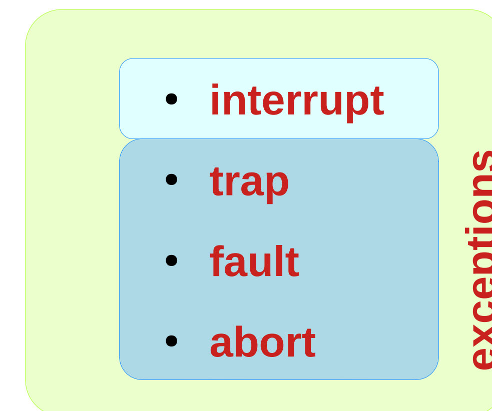
Faults are detected and serviced
by the processor before the faulting instructions.

Traps are serviced
after the instruction causing the trap.

User defined interrupts go into this category and
can be said to be traps;

this includes the MS- DOS **INT 21h**
software interrupt, for example.

Aborts are used only to signal
severe system problems,
when operation is no longer possible.



https://stackoverflow.com/questions/7295936/what-is-the-difference-between-interrupt-and-exception-context

Exceptions vs interrupts (3)

Trap

It is typically a type of synchronous interrupt
caused by an exceptional condition
(e.g., breakpoint, division by zero,
invalid memory access).

Fault

Fault exception is used in a client application
to catch contractually-specified SOAP faults.
By the simple exception message,
you can't identify the reason of the exception,
that's why a Fault Exception is useful.

Abort

It is a type of exception occurs
when an instruction fetch causes an error.

SOAP (formerly an acronym for Simple Object Access Protocol) is a messaging protocol specification for exchanging structured information in the implementation of web services in computer networks.

https://www.geeksforgeeks.org/difference-between-interrupt-and-exception/

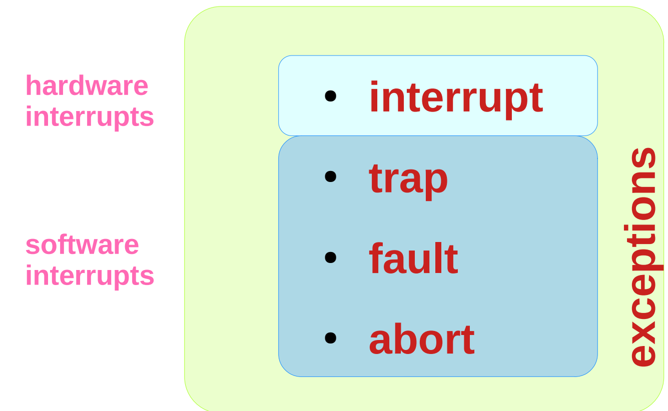
Exceptions vs interrupts (4)

Interrupt is one of the classes of **Exception**.
There are 4 classes of **Exception**
- **interrupt**, **trap**, **fault** and **abort**.

Even though there are many differences,
interrupt belongs to **exception** still

In any computer,
during its normal execution of a program,
there could be events that can cause
the CPU to temporarily halt.
Events like this are called interrupts.

Interrupts can be caused
by either software or hardware faults.



- hardware interrupts are called **Interrupts** — external, asynchronous
- software interrupts are called **Exceptions** — internal, instruction

https://www.geeksforgeeks.org/difference-between-interrupt-and-exception/

Exceptions vs interrupts (5)

The term **Interrupt** is usually reserved for hardware interrupts.

They are program control interruptions caused by external hardware events.

Here, external means external to the CPU.

Hardware interrupts usually come from many different sources

- timer chip
- peripheral devices (keyboards, mouse, etc.)
- I/O ports (serial, parallel, etc.)
- disk drives, CMOS clock
- expansion cards (sound / video card, etc)

That means hardware interrupts almost never occur due to some event related to the executing program.

https://www.geeksforgeeks.org/difference-between-interrupt-and-exception/

Exception is a software interrupt, which can be identified as a special handler routine.

Exception can be identified as an automatically occurring **trap**.

Generally, there are no specific instructions associated with exceptions

traps are generated using a specific instruction
int is x86 jargon for "trap instruction"
- a call to a predefined interrupt handler.

So, an **exception** occurs due to an “exceptional” condition that occurs during program execution.

Exceptions vs interrupts (6)

Interrupt	Exception
• These are Hardware interrupts.	• These are Software Interrupts.
• Occurrences of hardware interrupts usually disable other hardware interrupts.	• This is not a true case in terms of Exception. (does not disable other exceptions)
• These are asynchronous external requests for service (like keyboard or printer needs service).	• These are synchronous internal requests for service based upon abnormal events (think of illegal instructions, illegal address, overflow etc).
• Being asynchronous, interrupts can occur at any place in the program.	• Being synchronous, exceptions occur when there is abnormal event in your program like, divide by zero or illegal memory location.
• These are normal events and shouldn't interfere with the normal running of a computer.	• These are abnormal events and often result in the termination of a program

https://www.geeksforgeeks.org/difference-between-interrupt-and-exception/

Interrupt examples

An event like a key press on the keyboard,
or an internal hardware timer timing out
can *raise* this kind of interrupt and
can *inform* the CPU
that a certain device needs some attention.

the CPU will *stop* whatever it was doing,
provides the service required by the device
and will *get back* to the normal program.

When hardware interrupts occur
and the CPU starts the **ISR**,
other hardware interrupts are *disabled*
(e.g. in 80×86 machines).

If you need other hardware interrupts to occur
while the **ISR** is running,
you need to do that explicitly
by clearing the interrupt flag
with **CLI** / **STI** instruction in 80x86
with **MSR** in ARM

In 80×86 machines,
clearing the interrupt flag will
only affect hardware interrupts.

https://www.geeksforgeeks.org/difference-between-interrupt-and-exception/

Exception examples

Division by zero,
execution of an illegal opcode or
memory related fault could cause exceptions.

Whenever an exception is raised,
the CPU temporarily *suspends* the program
it was executing and starts the **ISR**.

ISR will contain what to do with the exception.

It may correct the problem or
if it is not possible,
it may abort the program gracefully
by printing a suitable error message.

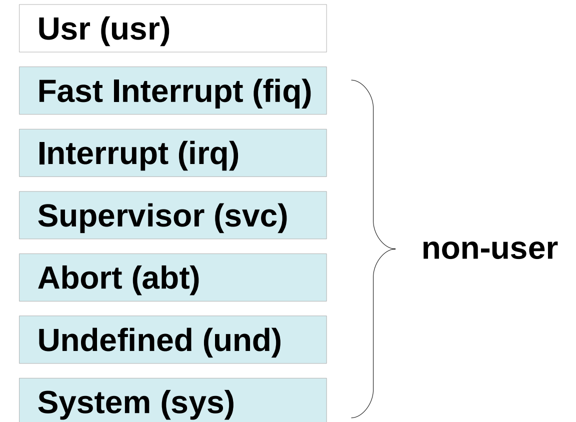
Although a *specific* instruction
does *not* cause an exception,
an exception will *always* be
caused *by an instruction*.

For example, the division by zero error
can only occur during the execution
of the division instruction.

https://www.geeksforgeeks.org/difference-between-interrupt-and-exception/

(1) Mode of operations

- 7 modes of operation.
- most application programs execute in **user** mode
- **Non user** modes (called **privileged** modes) are entered to serve **interrupts** or **exceptions**



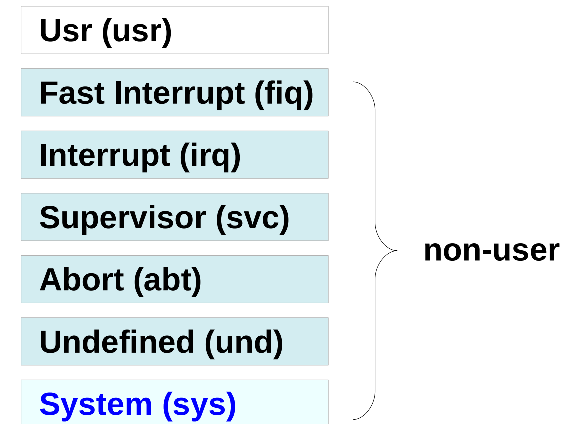
https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(2) Mode of operations

- The **system** mode is special mode for accessing protected resources.

  Because **exception handlers** in **system mode** does not use *registers*,

  errors in **exception handler** cannot corrupt registers

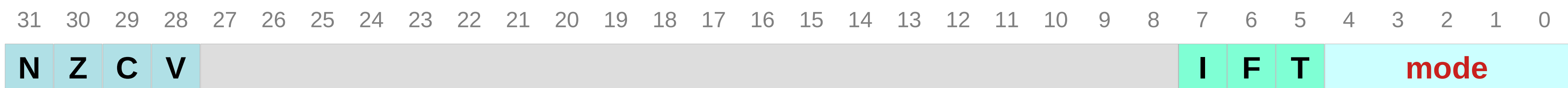


https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(3) Mode of operations

- **switching** between modes can be done <u>manually</u> through modifying the **mode bits** in the **CPSR** register.

Usr (usr)	1	0	0	0	0
Fast Interrupt (fiq)	1	0	0	0	1
Interrupt (irq)	1	0	0	1	0
Supervisor (svc)	1	0	0	1	1
Abort (abt)	1	0	1	1	1
Undefined (und)	1	1	0	1	1
System (sys)	1	1	1	1	1



https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(4) Mode of operations

Processor Mode		Description
USR	User	Normal program execution mode
FIQ	FIQ	Fast data processing mode
IRQ	IRQ	For general purpose interrupts
SVC	Supervisor	A protected mode for the OS
ABT	Abort	When data or instruction fetch is aborted
UND	Undefined	For undefined instructions
SYS	System	Privileged mode for OS Tasks

Switching between these modes requires saving/loading register values

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(4) Mode of operations

- **User** mode is an **unprivileged** mode,
  and has **restricted** access to system resources.

- **Non-user** modes
  - have **full** access to system resources
    in the current **security state**,
  - can change mode freely,
  - execute software as privileged.

- **Non-user** mode are entered
  - to service **exceptions**,
  - or to access privileged resources.

- **Applications** that require **task protection**
  usually execute in **User** mode.

- Some **embedded applications**
  might run entirely in **Non-user** mode.

- An **application** that requires **full** access
  to system resources usually executes
  in **System** mode.

https://www.keil.com/support/man/docs/armasm/armasm_dom1359731126962.htm

(5) Mode of operations

- **Supervisor** (**svc**) mode: A **privileged** mode entered whenever the CPU is reset or when an SVC instruction is executed.

- whereas **System** mode is the only **privileged** mode that is not entered by an **exception**.

  - It can only be entered by executing an instruction that explicitly writes to the **mode bits** of the Current Program Status Register (**CPSR**).
  - So, the **exception handlers** modify the **CPSR** to enter System mode.

- Usage: **Corruption** of the **link register** can be a problem when handling **multiple exceptions** of the same.
- the **System** mode shares the same registers as **User** mode,
  it can run tasks that require privileged access,
  and **exceptions** no longer overwrite the link register.

- Linux kernel has done it this way,
  so that whenever any **interrupt** occurs
  in first level **IRQ handler**,
  it copies **IRQ registers** to **SVC registers** and switch the ARM to **SVC** mode.

https://www.quora.com/In-ARM-processor-what-is-the-difference-in-supervisor-mode-and-system-mode

(3) 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

R8	R9	R10	R11	R12
R8_**fiq**	R9_**fiq**	R10_**fiq**	R11_**fiq**	R12_**fiq**

- **R13** is the stack pointer **SP** (*banked*)
- **R14** is subroutine link register **LR** (*banked*)
- **R15** is program counter **PC**
- **R16** is current program status register **CPSR** (*banked*)

SP (R13)	LR (R14)	SPSR (R16)
SP_**fiq**	LR_**fiq**	SPSR_**fiq**
SP_**irq**	LR_**irq**	SPSR_**irq**
SP_**svc**	LR_**svc**	SPSR_**svc**
SP_**abt**	LR_**abt**	SPSR_**abt**
SP_**und**	LR_**und**	SPSR_**und**

Banked registers

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

The same registers across different modes

User	System	Fast Interrupt	Interrupt	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8_fiq	R8	R8	R8	R8
R9	R9	R9_fiq	R9	R9	R9	R9
R10	R10	R10_fiq	R10	R10	R10	R10
R11	R11	R11_fiq	R11	R11	R11	R11
R12	R12	R12_fiq	R12	R12	R12	R12
R13 (SP)	R13 (SP)	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14 (LR)	R14 (LR)	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

http://www.cs.otago.ac.nz/cosc440/readings/arm-syscall.pdf

Actual number of different registers

16+1	0	7+1	2+1	2+1	2+1	2+1
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4			R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8_fiq	R8	R8	R8	R8
R9	R9	R9_fiq	R9	R9	R9	R9
R10	R10	R10_fiq	R10	R10	R10	R10
R11	R11	R11_fiq	R11	R11	R11	R11
R12	R12	R12_fiq	R12	R12	R12	R12
R13 (SP)	R13 (SP)	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14 (LR)	R14 (LR)	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

31 general purpose registers

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

6 control registers

http://www.cs.otago.ac.nz/cosc440/readings/arm-syscall.pdf

ARM Processor Registers

User	System	Fast Interrupt	Interrupt	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8_fiq	R8	R8	R8	R8
R9	R9	R9_fiq	R9	R9	R9	R9
R10	R10	R10_fiq	R10	R10	R10	R10
R11	R11	R11_fiq	R11	R11	R11	R11
R12	R12	R12_fiq	R12	R12	R12	R12
R13 (SP)	R13 (SP)	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14 (LR)	R14 (LR)	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

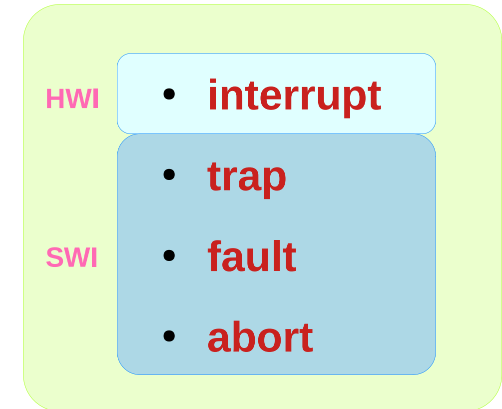
http://www.cs.otago.ac.nz/cosc440/readings/arm-syscall.pdf

(4) Exceptions

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

Examples

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



https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(5) Exceptions and modes

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

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

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(6) Vector table

a table of addresses that the ARM core branches to
when an exception is raised
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]

0x0000 0000	ldr pc, [pc, #offset0]
0x0000 0004	ldr pc, [pc, #offset1]
0x0000 0008	ldr pc, [pc, #offset2]
0x0000 000C	ldr pc, [pc, #offset3]
0x0000 0010	ldr pc, [pc, #offset4]
0x0000 0014	ldr pc, [pc, #offset5]
0x0000 0018	ldr pc, [pc, #offset6]
0x0000 001C	ldr pc, [pc, #offset7]

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(7) Exception priorities

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

Exception	Mode	Priority
Fast Interrupt Request	FIQ	3
Interrupt Request	IRQ	4
SWI and RESET	SVC	6, 1
Pre-fetch or data abort	ABT	5, 2
Undefined Instruction	UND	6

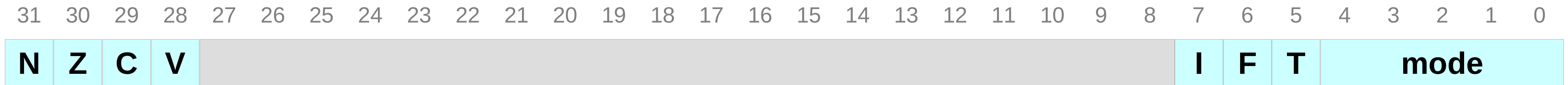
Priority decides which of the currently raised exceptions is more important

I bit and **F** bit decide if the exception handler itself can be interrupted during execution or not?

SWI and Undefined instruction : both are caused by an instruction entering the execution stage of the ARM instruction pipeline

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

CPSR and SPSR – **I** bit and **F** bit



Current Program Status Register (CPSR)

Saved Program Status Register (SPSR)

Exception	Priority	I bit	F bit	Mode
Reset	1	**1**	**1**	**SVC**
Data Abort	2	**1**	-	**ABT**
FIQ	3	**1**	**1**	**FIQ**
IRQ	4	**1**	-	**IRQ**
Prefetch	5	**1**	-	**ABT**
SWI	6	**1**	-	**SVC**
Undefined	6	**1**	-	**UND**

To disable Interrupt (IRQ), set **I**

To disable Fast Interrupt (FIQ), set **F**

the **T** bit shows running in the Thumb state

I bit and **F** bit decide if the exception handler itself can be interrupted during execution or not?

(8) Link register offset

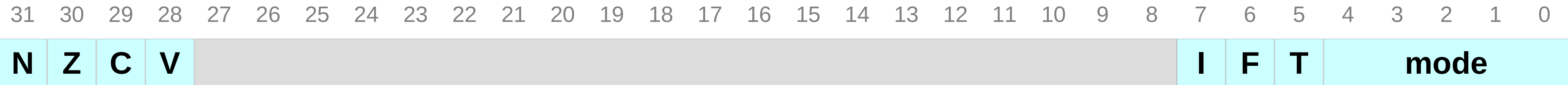
Link Register is used to return the **PC**
to the appropriate place in the interrupted task
since this is not always the old **PC** value.
It is modified depending on the type of exception.

The **PC** has advanced beyond the instruction
which caused 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**

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

CPSR and SPSR – **I** bit and **F** bit



Current Program Status Register (CPSR)

Saved Program Status Register (SPSR)

Exception	Priority	I bit	F bit	Mode	Return
Reset	1	**1**	**1**	**SVC**	**None**
Data Abort	2	**1**	-	**ABT**	**LR – 8**
FIQ	3	**1**	**1**	**FIQ**	**LR – 4**
IRQ	4	**1**	-	**IRQ**	**LR – 4**
Prefetch	5	**1**	-	**ABT**	**LR – 4**
SWI	6	**1**	-	**SVC**	**LR**
Undefined	6	**1**	-	**UND**	**LR**

(9) Entering and returning exception handler

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 (**LR** - 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)

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

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(10) Assigning interrupts

It is up to the system designer
who can decide which HW peripheral
can produce which interrupt.

But system designers have adopted
a standard design for assigning interrupts:

SWI are used to call privileged OS routines.
IRQ are assigned to general purpose interrupts
like periodic timers.
FIQ is reserved for one single interrupt source
that requires fast response time.

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(11) 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

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

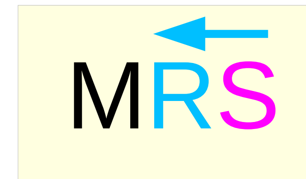
(12) Enabling and disabling interrupts

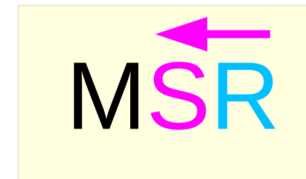
This is done by modifying the CPSR, this is done using only 3 ARM instruction:

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
```

x

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(13) 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.

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(14) 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!!

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(15) Interrupt handling – non-nested

- 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.

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(15) Nested interrupt handling

- Handling more than one interrupt at a time
  is possible by enabling interrupts
  before fully serving the current interrupt.
- Small latency
- For complex systems
- No difference between interrupts by priorities,
  so normal interrupts can block critical interrupts.

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(16) Nested interrupt handling

- 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

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(16) Priortized 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.
  - deterministic interrupt latency
  - overhead

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(17) other interrupt handling

- Re-entrant interrupt handler:
  re-enable interrupts earlier and
  support priorities, so the latency is reduced.
- Prioritized standard interrupt handler:
  arranges priorities in a special way
  to reduce the time needed to decide
  on which interrupt will be handled.
- Prioritized grouped interrupt handler:
  groups some interrupts into subset
  which has a priority level,
  this is good for large amount of interrupt sources.

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(18) other interrupt handling

Availability of different modes of operation in ARM
helps in exception handling in a structured way.

Context switching is one of the main issues
affecting interrupt latency,
and this is resolved in ARM FIQ mode
by increasing number of banked registers.

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:

- What type of interrupts are there?
- How many interrupts are there?

https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

Exception entry and return sequence (1)

At **exception entry**, the processor saves
R0-R3, **R12**, **LR**, **PC** and **PSR** on the stack.

Saving **PC** means that
the address of the next instruction to be executed
after return from the exception handler
is saved on the stack.

LR is also updated with EXC_RETURN and that
when the EXC_RETURN value is loaded to the **PC**,
the **exception return** sequence begins.

R0	**a0**	
R1	**a1**	
R2	**a2**	
R3	**a3**	
R4	v1	
R5	v2	
R6	v3	
R7	v4	
R8	v5	
R9	v6	SB
R10	v7	
R11	v8	FP
R12		**IP**
R13		SP
R14		**LR**
R15		**PC**

LR ← EXC_RETURN
PC ← LR

https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/4557/cortex-m4-exception-return-sequence

Exception entry and return sequence (2)

the EXC_RETURN values are special values
that are recognized by the hardware
rather than proper pc values.

Loading an EXC_RETURN value into the **PC**
initiates the hardware sequence
– the reverse of the interrupt sequence
– the return sequence

That reverse sequence will then
load the actual pc to resume at.

You don't explicitly load the various registers,
that is all done automatically by the return sequence.

Return Sequence

LR ← EXC_RETURN
PC ← **LR**

Automatic Hardware Sequence

- no explicit register loading
- only have to change
  the EXC_RETURN **value**

https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/4557/cortex-m4-exception-return-sequence

Exception entry and return sequence (3)

Loading **PC** with the value of **LR** is suffiient.

LR already holds EXC_RETURN, and
you do not have to worry about
which stack you need to use;
the EXC_RETURN in **LR** is
pre-encoded with the correct value.

Normally you only have to
change the EXC_RETURN **value**
when you're writing a **context-switcher**.

LR ← EXC_RETURN
PC ← LR

https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/4557/cortex-m4-exception-return-sequence

Exception entry and return sequence (4)

The EXC_RETURN is a nice feature of the **Cortex** architecture.

No need to have a RFI instruction (Return From Interrupt)

no difference in writing an interrupt-routine
and a normal subroutine
for a **Cortex-M** based microcontroller.

https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/4557/cortex-m4-exception-return-sequence

Exception entry and return sequence (5)

1. An interrupt is signalled; a pending-flag is set.
2. The interrupt is started, the registers **xPSR**, **PC**, **LR**, **R12**, **R3-R0** are all pushed onto the **interrupt-stack**.
3. The processor state is changed to use the interrupt-state.
4. The LR is loaded with the EXC_RETURN value (which is one of these: 0xFFFFFFF1, 0xFFFFFFF9, 0xFFFFFFFD, 0xFFFFFFE1, 0xFFFFFFE9 or 0xFFFFFFED).
5. The PC is loaded with the address from the interrupt-vector.
6. Your Interrupt Service Routine (ISR) is executed.
7. You make sure the LR register is saved/restored if it's changed.
8. You finish your Interrupt Service Routine by executing a **BX LR** instruction.
9. The EXC_RETURN value from the LR register is now moved into PC. The core now sees that this is a special return-address, so it restores the registers from the current stack.
10. When the registers are restored, the execution continues where it was interrupted.

https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/4557/cortex-m4-exception-return-sequence

Exception entry and return sequence (5)

2. The interrupt is started, the registers **xPSR**, **PC**, **LR**, **R12**, **R3-R0** are all pushed onto the **interrupt-stack**.

9. The EXC_RETURN value from the LR register is now moved into PC. The core now sees that this is a special return-address, so it restores the registers from the current stack.

https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/4557/cortex-m4-exception-return-sequence

Exception entry and return sequence (5)

Interrupt code typically uses stacks.
And there is a separate R13 for each mode (except one).
So there is a separate stack per mode
and at machine startup, it needs to be initialized.

- Initialization via a **MSR** (move into status register) instruction to change mode.
- Then store a value to (that) SP.
- Then use a **MSR** to put mode back

http://www2.unb.ca/~owen/courses/2253-2017/slides/08-interrupts.pdf

Exception entry and return sequence (5)

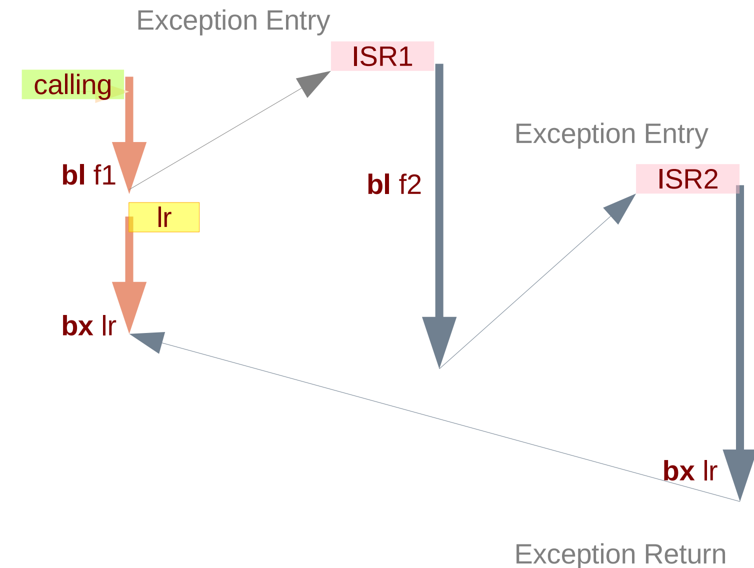
Mrs. MRS
●MSR moves to a status register.
–Status registers are CPSR, SPSR
–Underscores after (eg CPSR_cf) indicate
which sub-parts of the status register are affected.
Used in book code but not described... _cxsf is all of it?.
●MRS moves a status register into a regular register.

https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/4557/cortex-m4-exception-return-sequence

Exception entry and return sequence (6)

the hardware entry and return sequence
allows the processor not actually
to do the return sequence
if there is a **pending interrupt**

Instead, it immediately start
handling the new interrupt
without having to load the registers on return
and then store them again
before entering the new interrupt handler.



https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/4557/cortex-m4-exception-return-sequence

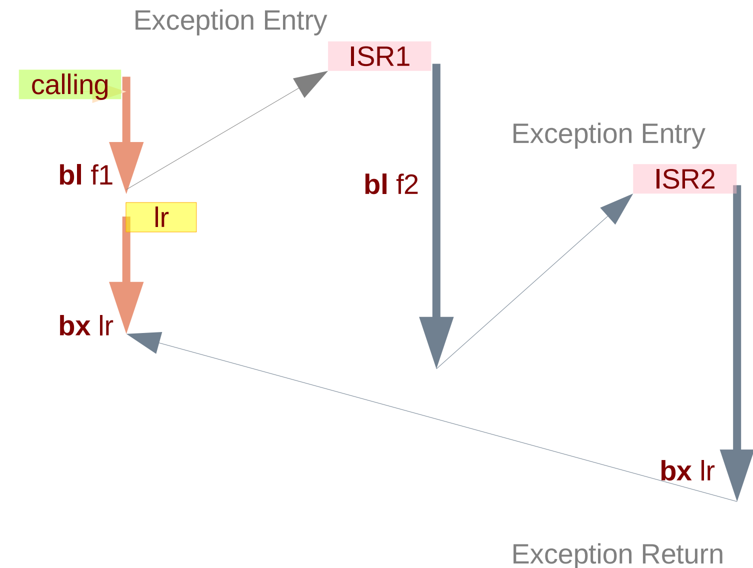
Exception entry and return sequence (7)

If an exception is still in pending state
when another exception handler has been completed,

instead of returning to the interrupted program
and then entering the exception sequence again,

a **tail-chain** scenario will occur,
where the processor will not have to
restore all register values from the stack and
push them back to the stack again.

The tail-chaining of exceptions allows
lower exception processing overhead
and better energy efficiency.



https://stackoverflow.com/questions/13029201/tail-chaining-of-interrupts

Exception entry and return sequence (8)

it's possible that another interrupt will be handled by tail-chaining.

This may occur between step 8 and step 9.

8. You finish your Interrupt Service Routine by executing a **BX LR** instruction.
9. The EXC_RETURN value from the **LR** register is now moved into PC.
   The core now sees that this is a special return-address,
   so it restores the registers from the current stack.

the registers **R0**-**R3** and **R12** will not contain values identical
to what is on the stack on interrupt entry.

In fact, you can never trust what's in **R0-R3** and **R12**,
so if you need those values
  for instance if you're using SVC,
  or if you're making some debug-facility,
then fetch them from the stack.

R0	**a0**
R1	**a1**
R2	**a2**
R3	**a3**
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	**IP**
R13	
R14	**LR**
R15	**PC**

https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/4557/cortex-m4-exception-return-sequence

Exception entry and return sequence (9)

That makes sense given the wonderful features
such as Tail chaining or pop pre-emption.

As the **AAPCS** calls for, **R0**-**R3** can be used
as input parameters/arguments to the function being called,
but it is rather safer that the function/subroutine
should fetch the values from stack
instead of directly referring to.

It is seemed that the handler would not know under
which circumstances it is executing -
either because of tail chaining or
it entering the handler from the thread mode.

If the handler is entered from thread mode
executing normal user program,
then the **R0**-**R3** will be having correct value
but if it is something like tail chaining,
those may not be correct.

https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/4557/cortex-m4-exception-return-sequence

References

[1] http://wiki.osdev.org/ARM_RaspberryPi_Tutorial_C
[2] http://blog.bobuhiro11.net/2014/01-13-baremetal.html
[3] http://www.valvers.com/open-software/raspberry-pi/
[4] https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html