Exceptions

Copyright (c) 2022 - 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

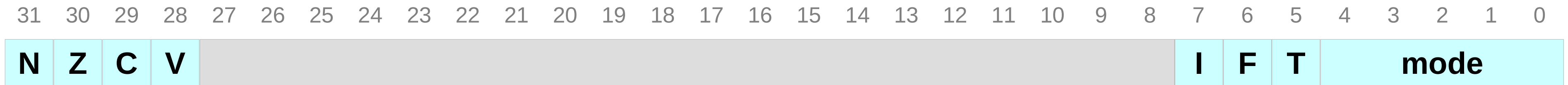
_<field> is one of

_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

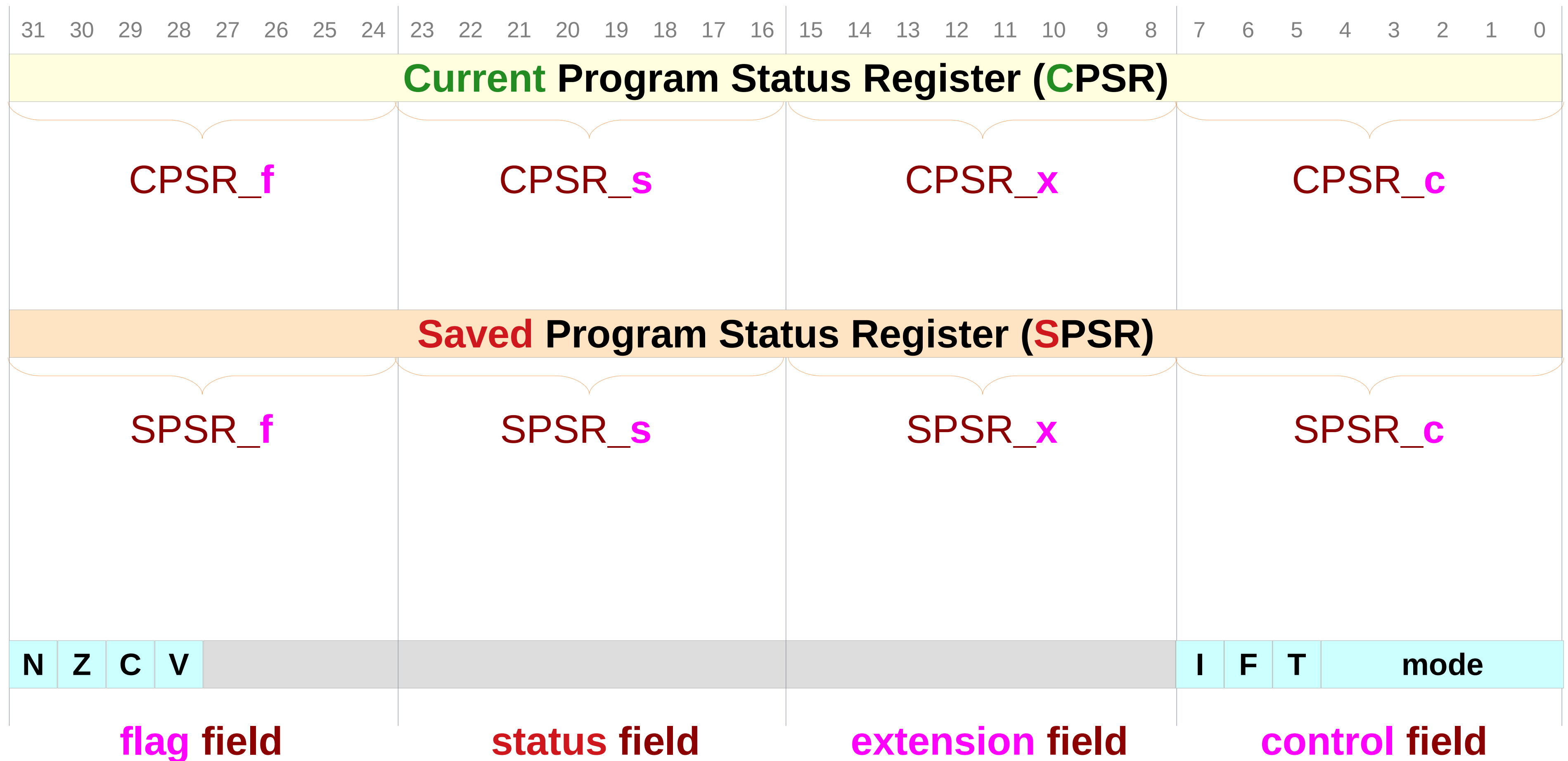
To disable Interrupt (IRQ), set **I**

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

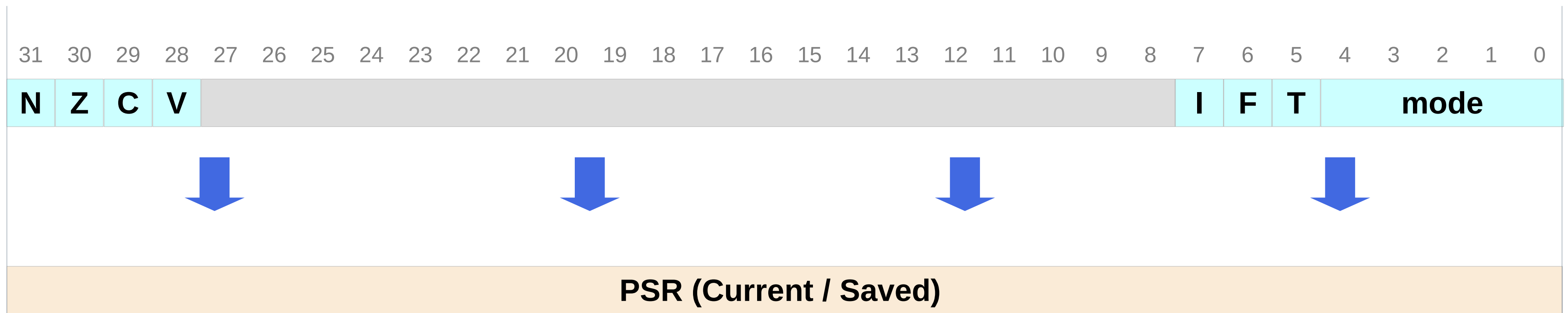
the **T** bit shows running in the Thumb state

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

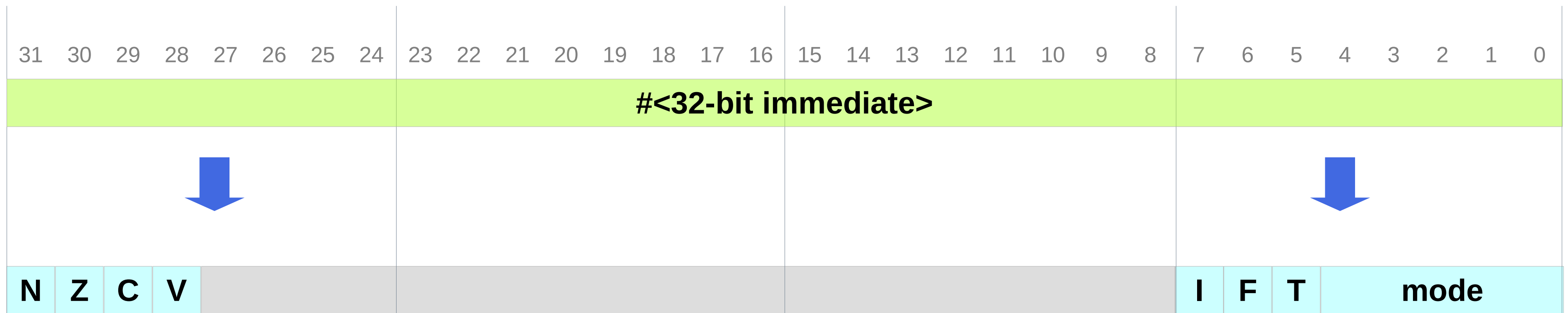


MRS Rd, CPSR
MRS Rd, SPSR

M R ⬅ S

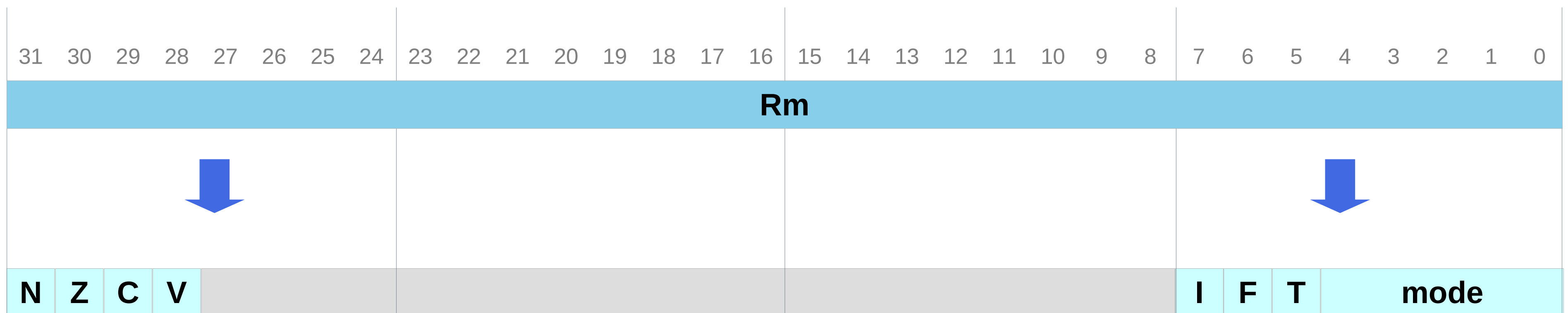
M S ⬅ R

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

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.

exceptions

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

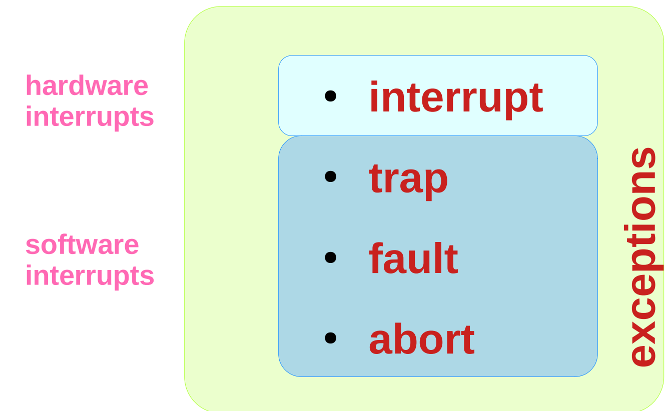
It is a type of exception occurs when an instruction fetch causes an error.

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.

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.

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

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.

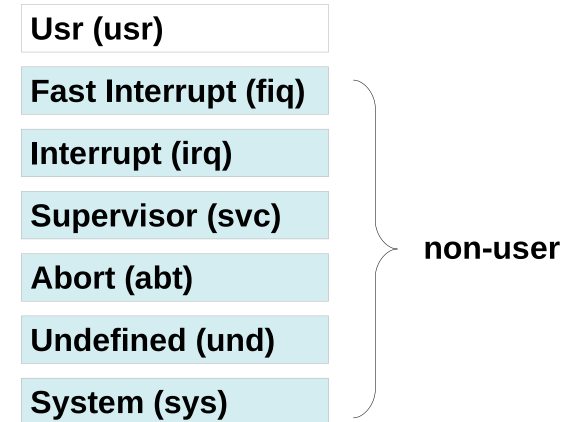
Although a *specific* instruction
does *not* cause an exception,
an exception will *always* be
caused *by an instruction*.

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**



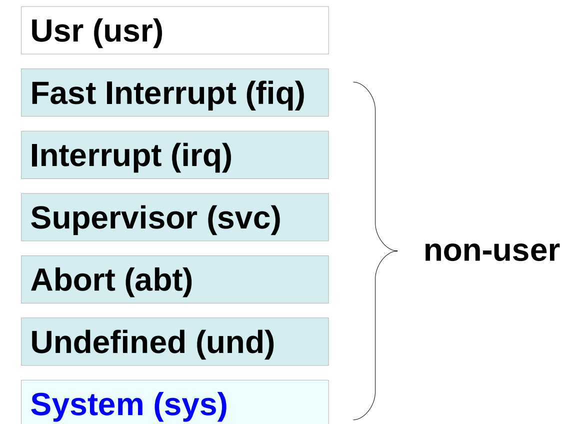
https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(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

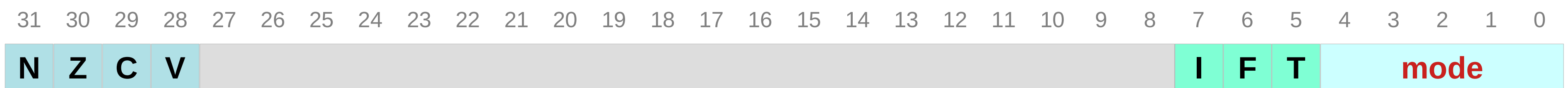


https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(3) Mode of operations

- **switching** between modes can be done manually 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

Registers in exception handlers

- The mode change associated with an exception occurring means that as a minimum, the particular exception handler called will have access to
  - its own stack pointer (SP_<mode>)
  - its own link register (LR_<mode>)
  - Its own saved program status register (SPSR_<mode>)
  - for a FIQ handler, 5 other general purpose registers (r8_FIQ to r12_FIQ)
  - other registers will be shared with the previous mode
  - SP_<mode> must maintain 8-byte alignment at external interfaces
-
- The exception handler must ensure that other (corrupted) registers are restored to their original state upon exit
  - This can be done by storing the contents of any working registers
  - on the stack and restoring them before returning

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.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)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

31 general purpose registers

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

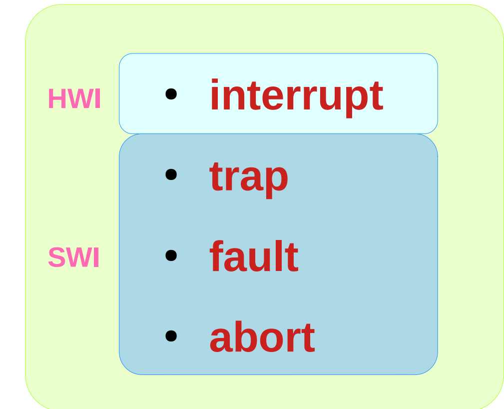
http://www.cs.otago.ac.nz/cosc440/readings/arm-syscall.pdf

(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

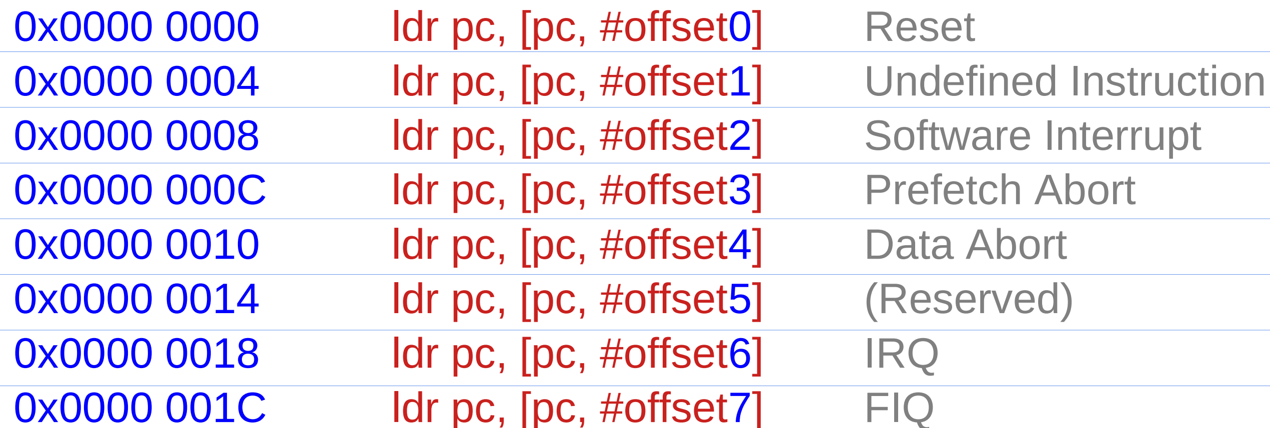
https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

(6) Vector table

a table of branching instructions
by which the ARM core branches to the correct ISR
when an exception is raised

example branching instruction

ldr pc, [pc, #_IRQ_handler_offset]

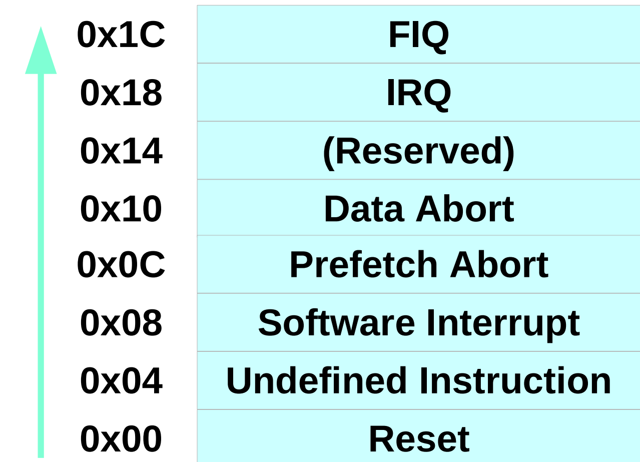


0x0000 0000	ldr pc, [pc, #offset0]	Reset
0x0000 0004	ldr pc, [pc, #offset1]	Undefined Instruction
0x0000 0008	ldr pc, [pc, #offset2]	Software Interrupt
0x0000 000C	ldr pc, [pc, #offset3]	Prefetch Abort
0x0000 0010	ldr pc, [pc, #offset4]	Data Abort
0x0000 0014	ldr pc, [pc, #offset5]	(Reserved)
0x0000 0018	ldr pc, [pc, #offset6]	IRQ
0x0000 001C	ldr pc, [pc, #offset7]	FIQ

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

(6) Vector table

- **Reset** - executed on power on
- **Undef** - when an invalid instruction reaches the execute stage of the pipeline
- **SWI** - when a software interrupt instruction is executed
- **Prefetch** - when an instruction is fetched from memory that is invalid for some reason, if it reaches the execute stage then this exception is taken
- **Data** - if a load/store instruction tries to access an invalid memory location, then this exception is taken
- **IRQ** - normal interrupt
- **FIQ** - fast interrupt



Address	Vector
0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector table may be placed at **0xFFFF0000** on ARM720T, ARM9 family and later devices

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

(6) Vector table

branching instructions at the vector table

- **B <Add>**
- **LDR pc, [pc, #offset]**
- **LDR pc, [pc, #-0xff0]**
- **MOV pc, #immediate**

http://classweb.ece.umd.edu/enee447.S2019/ARM-Documentation/ARM-Interrupts-3.pdf

(6) Vector table

- **B <Addr>**

used to make branching to the memory location
with address "Addr" relative to the current location of the pc.

- **LDR pc, [pc, #offset]**

used to load in the PC register
the old PC value + an offset value

- **LDR pc, [pc, #-0xff0]**

used only when an interrupt controller is available,
to load a specific ISR address from the vector table.

The vector interrupt controller (VIC) is
placed at memory address 0xfffff000
this is the base address of the VIC.
The ISR address is always located at 0xfffff030.

- **MOV pc, #immediate**

Load in the PC the value "immediate".

http://classweb.ece.umd.edu/enee447.S2019/ARM-Documentation/ARM-Interrupts-3.pdf

(6) Vector table

- **Branch Instruction** **B <Addr>**

direct branch always to handler address label
The handler must be within 32MB of the branch instruction,
which may not be possible with some memory organizations

- **Move PC instruction** **MOV pc, #immediate**

directly load the PC with a handler address label
located on applicable address boundary
Address must be able to be stored in 8-bits,
rotated right an even number of places

- **Load PC instruction** **LDR pc, [pc, #offset]** **LDR pc, [pc, #-0xff0]**

The PC is forced directly to the handler's address
by storing the address in a suitable memory location
(within 4KB of the vector address).
loading the vector with an instruction
which loads the PC with the contents of the chosen memory location.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

(6) Vector table

Note that the Load PC cannot be written using MOV
because the address location of the Undef handler
cannot be generated
using 8-bits rotated right an even number of places.

for the Move PC example the value 0x03 is
rotated right four bits which is stored as
two lots of 2 bits and is hence encoded as 0xA30F203

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

IRQ with VIC (1)

the startup assembly file in the Keil environment

Exception vectors should be linked and programmed correctly.
This is usually managed by the linker.
Also appropriate handlers need to be programmed
at the respective locations.

For instance at the IRQ vector (0x18)
the following instruction should exist
if the ISR address is read directly
from the **VIC** Vector Address Register
(register address: 0xFFFFF030)

```
LDR   PC  [PC,#-0xFF0]
```

0x18 + 0x8 – 0xff0 = - 0xfd0 = 0xFFFFF030

the base address of the VIC
0xfffff000

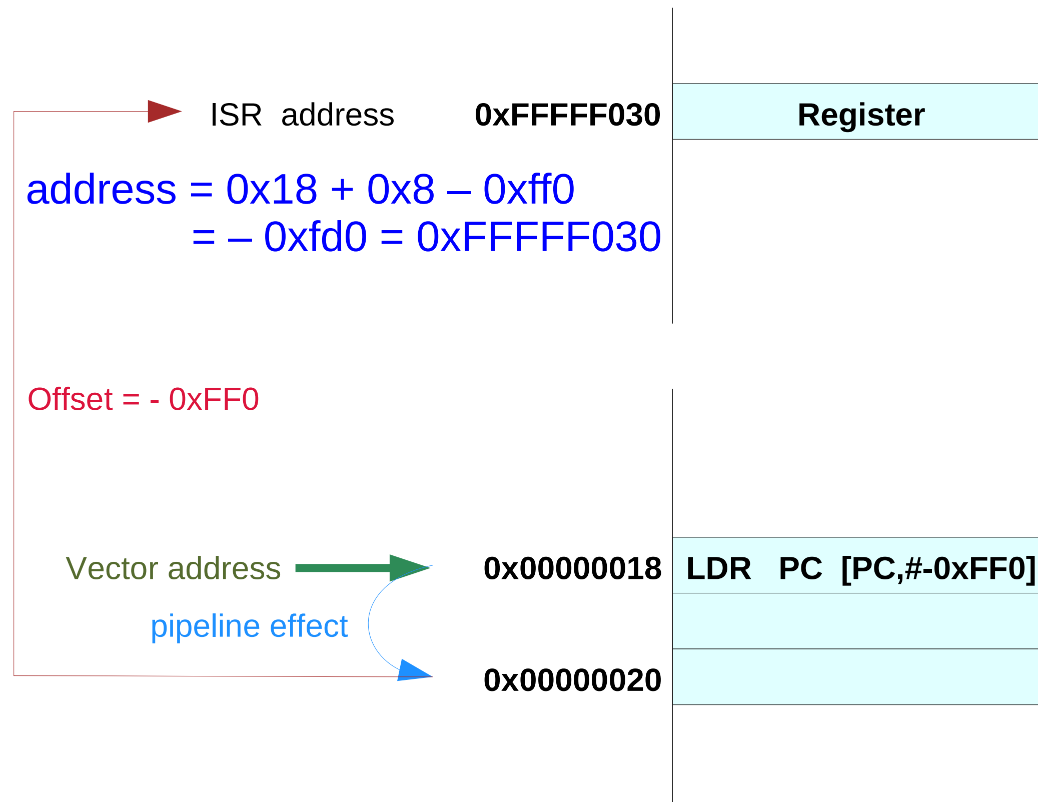
the ISR address is always at
at 0xfffff030.

offset address = -0xFF0
= 0xFFFFF010
address = 0xFFFFF030
vector address = 0x00000018
pipeline effect = 0x00000008

0x0000 0000	ldr pc, [pc, #offset0]	Reset
0x0000 0004	ldr pc, [pc, #offset1]	Undefined Instruction
0x0000 0008	ldr pc, [pc, #offset2]	Software Interrupt
0x0000 000C	ldr pc, [pc, #offset3]	**Prefetch Abort**
0x0000 0010	ldr pc, [pc, #offset4]	**Data Abort**
0x0000 0014	ldr pc, [pc, #offset5]	(Reserved)
0x0000 0018	ldr pc, [pc, #**-0x0ff0**]	**IRQ**
0x0000 001C	ldr pc, [pc, #offset7]	**FIQ**

https://www.nxp.com/docs/en/application-note/AN10381.pdf

IRQ with VIC (2)



the base address of the VIC
0xfffff000

the register address is always at
at 0xfffff030.

offset address = -0xFF0
= 0xFFFFF010
address = 0xFFFFF030
vector address = 0x00000018
pipeline effect = 0x00000008

Offset = (address location - vector address - pipeline effect)
= – 0xFD0 – 0x18 – 0x8 = – 0xFF0

https://www.nxp.com/docs/en/application-note/AN10381.pdf

Undef with VIC (1)

LDR PC, [PC+offset]
LDR pc, [pc, #-0xff0]

offset address
= (address location - vector address - pipeline effect)
= 0xFFC - 0x4 - 0x8
= 0xFF0

0x4 + 0x8 – 0xff0 =
– 0xfe4 = 0xFFFFF01C

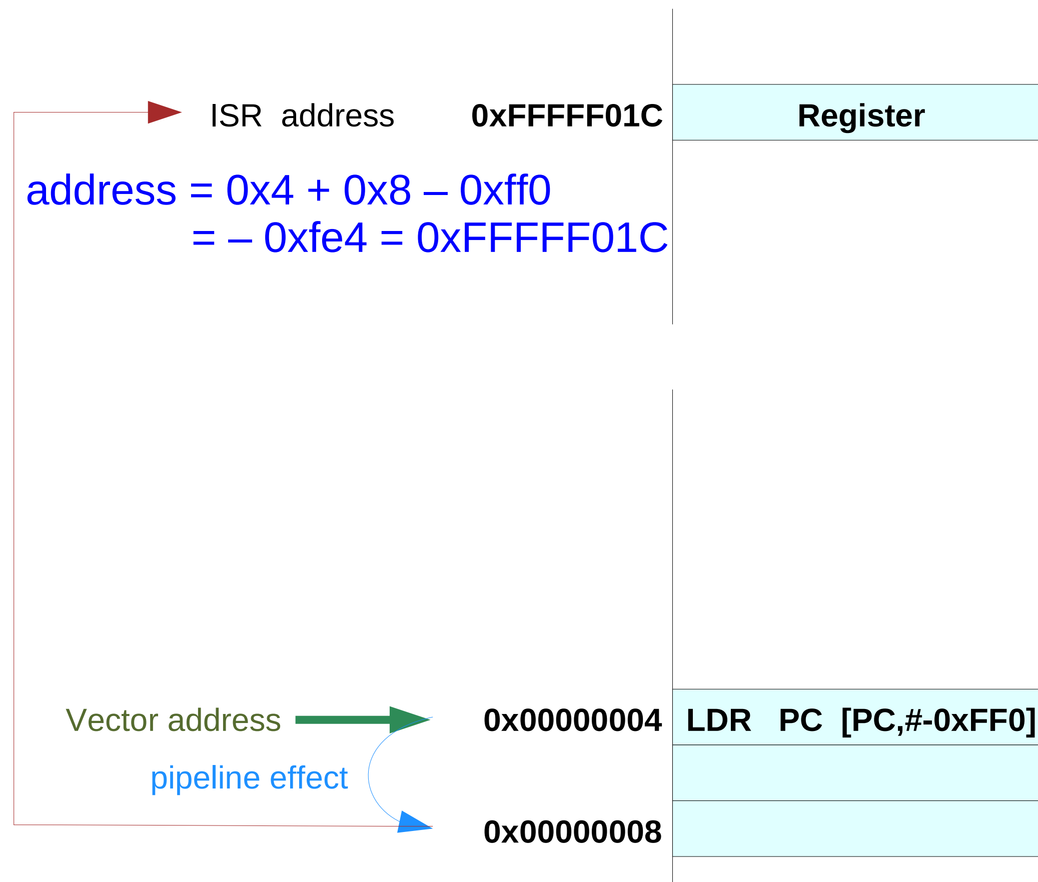
the base address of the VIC
0xfffff000

the ISR address is always
at 0xfffff030.

0x0000 0000	ldr pc, [pc, #offset0]	Reset
0x0000 0004	ldr pc, [pc, #offset1]	**Undefined** Instruction
0x0000 0008	ldr pc, [pc, #offset2]	Software Interrupt
0x0000 000C	ldr pc, [pc, #offset3]	**Prefetch Abort**
0x0000 0010	ldr pc, [pc, #offset4]	**Data Abort**
0x0000 0014	ldr pc, [pc, #offset5]	(Reserved)
0x0000 0018	ldr pc, [pc, #**-0x0ff0**]	**IRQ**
0x0000 001C	ldr pc, [pc, #offset7]	**FIQ**

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Undef with VIC (2)



the base address of the VIC
0xfffff000

the ISR address is always at
at 0xfffff01C.

offset address = -0xFF0
= 0xFFFFF010
address = 0xFFFFF01C
vector address = 0x00000004
pipeline effect = 0x00000008

Offset = (address location - vector address - pipeline effect)
= – 0xFE4 – 0x4 – 0x8 = – 0xFF0

https://www.nxp.com/docs/en/application-note/AN10381.pdf

(6) Vector table

3. Stack pointers should be programmed correctly for FIQ and IRQ.

4. The VIC is programmed correctly with the ISR address.
This needs to be handled in the application.

5. Compiler supported keywords are used for the Interrupt handlers.
For instance in Keil, an ISR function could have the following form.
void IRQ_Handler()__irq
More details on compiler keywords is provided in the next section.

https://www.nxp.com/docs/en/application-note/AN10381.pdf

ARM Interrupt Controller

When a peripheral or device requires attention,
it raises an interrupt to the processor.

An interrupt controller provides
a programmable governing policy
software to determine
which peripheral or device can
interrupt the processor
at any specific time
by setting the appropriate bits
in the interrupt controller registers.

There are two types of interrupt controller available
for the ARM processor:

- standard interrupt controller
- vector interrupt controller (VIC).

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

Interrupt Vector Table

- Interrupt vector table contains
  the address of the IRQ handlers of every interrupt.
- They point the program counter where to go,
  if an interrupt occurs.
- Priorly VIC was referred as Interrupt vector table,
  because they just point the address when an interrupt occurs.
  They don't completely handle them as per priority.

https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-processors

Exception Return Instructions

```
    AREA vectors, CODE, READONLY
    ENTRY

Vector_Table
    LDR     pc, reset_addr
    LDR     pc, undef_addr
    LDR     pc, swi_addr
    LDR     pc, prefetch_addr
    LDR     pc, abort_addr
    NOP     ; Reserved
    LDR     pc, irq_addr
FIQ_Handler
    ; FIQ handler code, < 4kB in size

reset_addr      DCD Reset_Handler
undef_addr      DCD Undef_Handler
swi_addr        DCD Swi_Handler
    ...
```

One typical approach is to use a literal pool for all of the addresses, so that
they can be modified later if necessary

You can include the FIQ handler at the end of the vector table (assuming it's < 4kB) but move the other handlers around to any location in the memory map. If you use LDR pc, ... from a literal pool, you won't suddenly find that it breaks your vector table instructions if the handlers change location.

http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf

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**.

ldr pc, [pc, #_IRQ_handler_offset]

0x0000 0000	ldr pc, [pc, #offset0]	Reset
0x0000 0004	ldr pc, [pc, #offset1]	Undefined Instruction
0x0000 0008	ldr pc, [pc, #offset2]	Software Interrupt
0x0000 000C	ldr pc, [pc, #offset3]	Prefetch Abort
0x0000 0010	ldr pc, [pc, #offset4]	Data Abort
0x0000 0014	ldr pc, [pc, #offset5]	(Reserved)
0x0000 0018	ldr pc, [pc, #offset6]	IRQ
0x0000 001C	ldr pc, [pc, #offset7]	FIQ

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

Vector table (2)

Reset vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.

Undefined instruction vector is used when the processor cannot decode an instruction.

Software interrupt vector is called when you execute a **SWI** instruction. The SWI instruction is frequently used to invoke an operating system routine.

0x0000 0000	ldr pc, [pc, #offset0]	**Reset**
0x0000 0004	ldr pc, [pc, #offset1]	**Undefined Instruction**
0x0000 0008	ldr pc, [pc, #offset2]	**Software Interrupt**
0x0000 000C	ldr pc, [pc, #offset3]	Prefetch Abort
0x0000 0010	ldr pc, [pc, #offset4]	Data Abort
0x0000 0014	ldr pc, [pc, #offset5]	(Reserved)
0x0000 0018	ldr pc, [pc, #offset6]	IRQ
0x0000 001C	ldr pc, [pc, #offset7]	FIQ

https://www.sciencedirect.com/topics/computer-science/exception-vector-table

Vector table (3)

0x0000 0000	ldr pc, [pc, #offset0]	Reset
0x0000 0004	ldr pc, [pc, #offset1]	Undefined Instruction
0x0000 0008	ldr pc, [pc, #offset2]	Software Interrupt
0x0000 000C	ldr pc, [pc, #offset3]	**Prefetch Abort**
0x0000 0010	ldr pc, [pc, #offset4]	**Data Abort**
0x0000 0014	ldr pc, [pc, #offset5]	(Reserved)
0x0000 0018	ldr pc, [pc, #offset6]	**IRQ**
0x0000 001C	ldr pc, [pc, #offset7]	**FIQ**

Prefetch abort vector occurs
when the processor attempts to fetch an instruction
from an address without the correct access permissions.
The actual abort occurs in the decode stage.

Data abort vector is similar to a prefetch abort
but is raised when an instruction attempts
to access data memory without the correct access permissions.

Interrupt request vector is used by external hardware
to interrupt the normal execution flow of the processor.
It can only be raised if **IRQ**s are not masked in the CPSR.

Fast interrupt request vector is similar to the interrupt request
but is reserved for hardware requiring faster response times.
It can only be raised if **FIQ**s are not masked in the CPSR.

https://www.sciencedirect.com/topics/computer-science/exception-vector-table

Typical vector table using a literal pool

```
        AREA vectors, CODE, READONLY
        ENTRY
Vector_Table
        LDR pc,  Reset_Addr
        LDR pc,  Undefined_Addr
        LDR pc,  SVC_Addr
        LDR pc,  Prefetch_Addr
        LDR pc,  Abort_Addr
        NOP      ; Reserved vector
        LDR pc,  IRQ_Addr

FIQ_Handler
        ; FIQ handler code - max 4kB in size

Reset_Addr      DCD Reset_Handler
Undefined_Addr  DCD Undefined_Handler
SVC_Addr        DCD SVC_Handler
Prefetch_Addr   DCD Prefetch_Handler
Abort_Addr      DCD Abort_Handler
                DCD 0           ;Reserved vector
IRQ_Addr        DCD IRQ_Handler
        ...
        END
```

0x0000 0000	ldr pc, [pc, #offset0]	Reset
0x0000 0004	ldr pc, [pc, #offset1]	Undefined Instruction
0x0000 0008	ldr pc, [pc, #offset2]	Software Interrupt
0x0000 000C	ldr pc, [pc, #offset3]	**Prefetch Abort**
0x0000 0010	ldr pc, [pc, #offset4]	**Data Abort**
0x0000 0014	ldr pc, [pc, #offset5]	(Reserved)
0x0000 0018	ldr pc, [pc, #offset6]	**IRQ**
0x0000 001C	ldr pc, [pc, #offset7]	**FIQ**

https://jianjiandudu.wordpress.com/2016/12/20/interrupt4-vector-table/

Typical vector table using a literal pool

- when the processor is reset then hardware sets the pc to 0x0000 and starts executing by fetching the instruction at 0x0000.
- when an undefined instruction is executed or tries to be executed the hardware responds by setting the pc to 0x0004 and starts executing the instruction at 0x0004.
- when irq interrupt happens, the hardware finishes the instruction it is executing starts executing the instruction at address 0x0018. and so on.

```
00000000 <_start>:
   0:  ea00000d  b   3c <_reset>
   4:  e59ff014  ldr pc, [pc, #20]  ; 20 <_undefined_instruction>
   8:  e59ff014  ldr pc, [pc, #20]  ; 24 <_software_interrupt>
   c:  e59ff014  ldr pc, [pc, #20]  ; 28 <_prefetch_abort>
  10:  e59ff014  ldr pc, [pc, #20]  ; 2c <_data_abort>
  14:  e59ff014  ldr pc, [pc, #20]  ; 30 <_not_used>
  18:  e59ff014  ldr pc, [pc, #20]  ; 34 <_irq>
  1c:  e59ff014  ldr pc, [pc, #20]  ; 38 <_fiq>
```

0x0000 0000	ldr pc, [pc, #offset0]	Reset
0x0000 0004	ldr pc, [pc, #offset1]	Undefined Instruction
0x0000 0008	ldr pc, [pc, #offset2]	Software Interrupt
0x0000 000C	ldr pc, [pc, #offset3]	**Prefetch Abort**
0x0000 0010	ldr pc, [pc, #offset4]	**Data Abort**
0x0000 0014	ldr pc, [pc, #offset5]	(Reserved)
0x0000 0018	ldr pc, [pc, #offset6]	**IRQ**
0x0000 001C	ldr pc, [pc, #offset7]	**FIQ**

https://stackoverflow.com/questions/21312963/arm-bootloader-interrupt-vector-table-understanding

Typical vector table using a literal pool

```
00000020 <_undefined_instruction>:
  20:   00000000    andeq   r0, r0, r0

00000024 <_software_interrupt>:
  24:   00000000    andeq   r0, r0, r0

00000028 <_prefetch_abort>:
  28:   00000000    andeq   r0, r0, r0

0000002c <_data_abort>:
  2c:   00000000    andeq   r0, r0, r0

00000030 <_not_used>:
  30:   00000000    andeq   r0, r0, r0

00000034 <_irq>:
  34:   00000000    andeq   r0, r0, r0

00000038 <_fiq>:
  38:   00000000    andeq   r0, r0, r0

0000003c <_reset>:
  3c:   00000000    andeq   r0, r0, r0
```

```
00000000 <_start>:
   0:  ea00000d  b   3c <_reset>
   4:  e59ff014  ldr pc, [pc, #20]  ; 20 <_undefined_instruction>
   8:  e59ff014  ldr pc, [pc, #20]  ; 24 <_software_interrupt>
   c:  e59ff014  ldr pc, [pc, #20]  ; 28 <_prefetch_abort>
  10:  e59ff014  ldr pc, [pc, #20]  ; 2c <_data_abort>
  14:  e59ff014  ldr pc, [pc, #20]  ; 30 <_not_used>
  18:  e59ff014  ldr pc, [pc, #20]  ; 34 <_irq>
  1c:  e59ff014  ldr pc, [pc, #20]  ; 38 <_fiq>
```

- change the pc
- start execution at *these addresses*
- save the state of the machine
- switch processor modes if necessary
- start executing at *the new address* from the vector table

https://stackoverflow.com/questions/21312963/arm-bootloader-interrupt-vector-table-understanding

Typical vector table using a literal pool

one word, one instruction for each location.

if we never expect to have any of these exceptions,
we do not need a branch instruction at address zero
    for example you can just have your program start,
    there is nothing magic about the memory at these addresses.

If you expect to have these exceptions,
then you have two choices for instructions that are one word
and can jump out of the way of the exception that follows.

- branch
- load pc.

```
0:   ea00000d  b    3c <_reset>
4:   e59ff014  ldr pc, [pc, #20]    ; 20 <_undefined_instruction>
```

https://stackoverflow.com/questions/21312963/arm-bootloader-interrupt-vector-table-understanding

Typical vector table using a literal pool

When the hardware takes an exception,

- the PC is automatically set to the address of the relevant exception vector
- the processor begins executing the instruction at that address.

- When the processor comes out of reset, the PC is automatically set to base+0.
- An undefined instruction sets the PC to base+4, etc.

The base address of the vector table (base) is either 0x00000000, 0xFFFF0000, or VBAR depending on the processor and configuration.

Note that this provides limited flexibility in where the vector table gets placed and you'll need to consult the ARM documentation in conjunction with the reference manual for the device that you are using to get the right value to be used.

0x0000 0000	ldr pc, [pc, #offset0]	Reset
0x0000 0004	ldr pc, [pc, #offset1]	Undefined Instruction
0x0000 0008	ldr pc, [pc, #offset2]	Software Interrupt
0x0000 000C	ldr pc, [pc, #offset3]	**Prefetch Abort**
0x0000 0010	ldr pc, [pc, #offset4]	**Data Abort**
0x0000 0014	ldr pc, [pc, #offset5]	(Reserved)
0x0000 0018	ldr pc, [pc, #offset6]	**IRQ**
0x0000 001C	ldr pc, [pc, #offset7]	**FIQ**

0xFFFF 0000	ldr pc, [pc, #offset0]	Reset
0xFFFF 0004	ldr pc, [pc, #offset1]	Undefined Instruction
0xFFFF 0008	ldr pc, [pc, #offset2]	Software Interrupt
0xFFFF 000C	ldr pc, [pc, #offset3]	**Prefetch Abort**
0xFFFF 0010	ldr pc, [pc, #offset4]	**Data Abort**
0xFFFF 0014	ldr pc, [pc, #offset5]	(Reserved)
0xFFFF 0018	ldr pc, [pc, #offset6]	**IRQ**
0xFFFF 001C	ldr pc, [pc, #offset7]	**FIQ**

https://stackoverflow.com/questions/21312963/arm-bootloader-interrupt-vector-table-understanding

Typical vector table using a literal pool

The layout of the table (4 bytes per exception) makes it necessary to immediately branch from the vector to the *actual* exception handler.

The reasons for the **LDR PC, label** approach are twofold

- because a PC-relative branch is limited to (24 << 2) bits (+/-32MB) using B would constrain the layout of the code in memory somewhat;
- by loading an absolute address (LDR PC, label) the handler can be located anywhere in memory.
- it makes it very simple to change exception handlers at runtime, by simply writing a different address to that location, rather than having to assemble and hotpatch a branch instruction.

0x0000 0000	ldr pc, [pc, #offset0]	Reset
0x0000 0004	ldr pc, [pc, #offset1]	Undefined Instruction
0x0000 0008	ldr pc, [pc, #offset2]	Software Interrupt
0x0000 000C	ldr pc, [pc, #offset3]	**Prefetch Abort**
0x0000 0010	ldr pc, [pc, #offset4]	**Data Abort**
0x0000 0014	ldr pc, [pc, #offset5]	(Reserved)
0x0000 0018	ldr pc, [pc, #offset6]	**IRQ**
0x0000 001C	ldr pc, [pc, #offset7]	**FIQ**

https://stackoverflow.com/questions/21312963/arm-bootloader-interrupt-vector-table-understanding

Typical vector table using a literal pool

There's little value to having a remappable reset vector in this way,
however, which is why you tend to see that one implemented
as a simple branch to skip over the rest of the vectors to the real entry point code.

https://stackoverflow.com/questions/21312963/arm-bootloader-interrupt-vector-table-understanding

(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

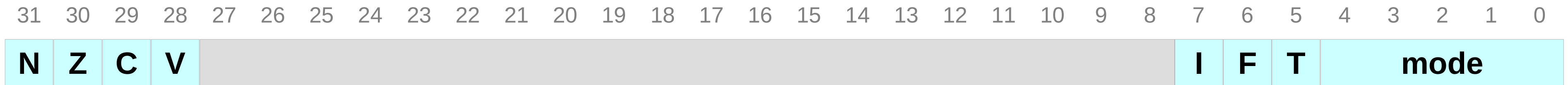
Priority decides which of the currently raised exceptions is more important

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

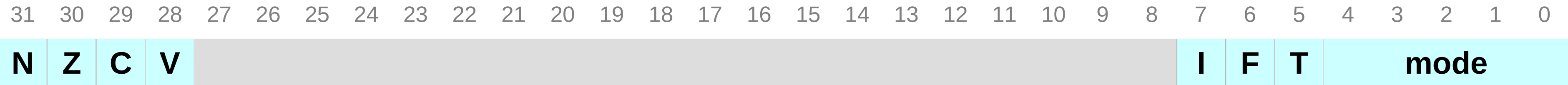
Link Register is used to return the **PC**
to the appropriate place in the interrupted task
since this is not always the old **PC** value.
It is modified depending on the type of exception.

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**

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