Exception Programming

Young Won Lim
3/12/22

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

This document was produced by using LibreOffice.

(9) Entering and returning exception handler

Return has to be done from ARM state.

no Thumb instruction to copy SPSR back into CPSR

v6 vectored interrupts will take the IRQ handler address directly
from the VIC (Vectored Interrupt Controller)
via the IRQADDR input to the core
and start executing from that address

Also the 1156 can take exceptions written in Thumb-2 code
in Thumb state, by setting the TE bit of cp15 register 1.
This is set on reset by the TEINIT signal to the core.

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

Exception Return Instructions

To return from an exception
- Use a data processing instruction
- The actual instruction used depends on the exception being handled

- With the **S** bit set
- With the **PC** as the destination register

- In privileged modes this not only updates the **PC** but also copies the **SPSR** into the **CPSR**

- For SWI and Undefined Instruction handlers
  MOVS pc,lr

- For FIQ, IRQ and Prefetch Abort handlers
  SUBS pc,lr,#4

- For Data Abort handlers
  SUBS pc,lr,#8

- LDM with ^ qualifier can also be used if LR adjusted before being stacked
  LDMFD sp!,{pc}^

- v6 cores also have RFE - more later

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

Exception Return

When returning from an exception,
can do both of the required actions
using a single data processing instruction with the S flag set,
and the PC as destination register

In privileged modes this not only updates the PC
but also copies the SPSR into the CPSR
However the actual instruction depends
on which exception is being handled.

Also possible to use a Load Multiple instruction (using the ^ qualifier)
to return if in a privileged mode with the PC as the destination.

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

(9) Entering and returning exception handler

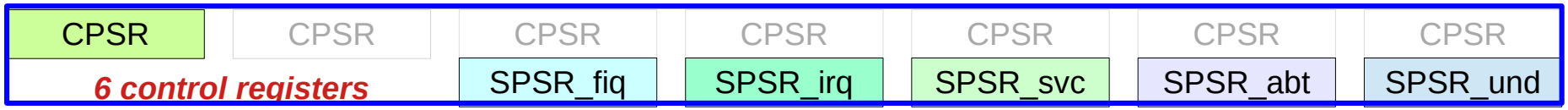
Entering exception handler

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.

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

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)



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

Entering and exiting an exception handler

- Preserve the address of the next instruction.
- Copy CPSR to the appropriate SPSR
  one of the banked registers for each mode of operation.
- Force the CPSR mode bits to a value depending on the raised exception.
- Force the PC to fetch the next instruction from the exception vector table.
- Now the handler is running in the mode associated with the raised exception.
- When handler is done, the CPSR is restored from the saved SPSR.
- PC is updated with the value of (LR - offset) and
  the offset value depends on the type of the exception.

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

Returning after exception handling

When the handler has finished its task,
it returns to the caller (in software)

- The mode needs to be put back to its pre-interrupt value.
  And the PC needs to be put back to the correct instruction.
  - to the instruction that had the exception (and did not successfully finish) or
  - to the next instruction, depending on the kind of an exception
- Change PC
  - **SUBS PC, LR, #4**
  - **SUBS PC, LR, #8**
- magic CPSR restore when PC is the destination and the S flag set
- on entry to handler adjust **LR** (eg subtract 4)
  - **STMDB sp!, {some regs, lr}**
- then to return, use a
  - **LDMIA sp!, {some regs, pc}^**
    - ^ means to restore CPSR also.

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

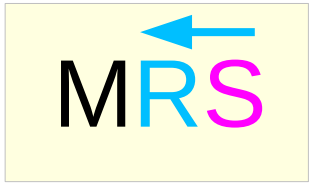
And this can be done by 2 methods:

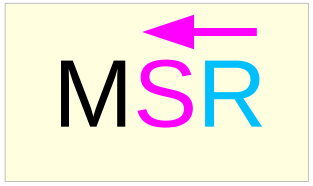
- allow nested interrupt handling
- give priorities to different interrupt sources

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

(12) Enabling and disabling interrupts

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
```

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

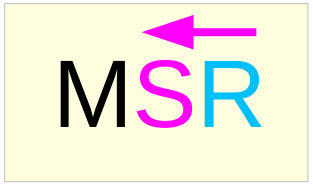
https://www.sciencedirect.com/topics/computer-science/software-interrupt

MRS / MSR instructions

MSR moves a regular register to a status register.

- status registers are CPSR, SPSR
- Underscores after (eg CPSR_cf) indicate which sub-parts of the status register are affected.

MRS moves a status register into a regular register



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

(13) Interrupt stack

exception handling uses stacks extensively
because each exception has
a specific mode of operation,
so switching between modes occurs and
saving the previous mode data is required before switching
so that the core can switch back to its old state successfully.

each mode has a dedicated register for a stack pointer.

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

(13) Interrupt stack

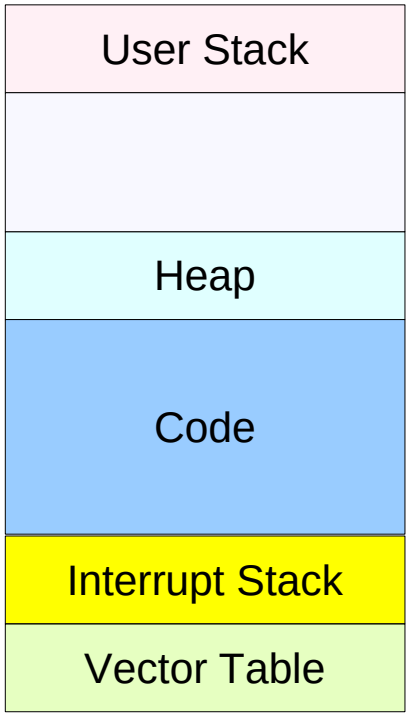
the design of these stacks depends on
some factors like operating system requirements
for stack design and target hardware physical limits
on size and position in memory.

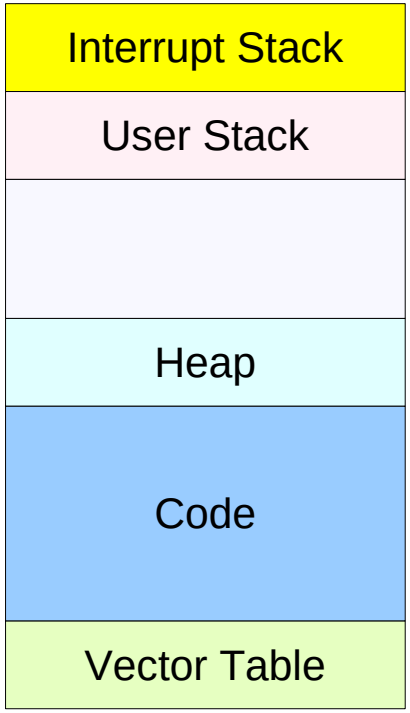
Most of ARM based systems has the stack designed
such that the top of it is located at high memory address.

A good stack design tries to avoid stack overflow
because this causes instability in embedded systems.

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

(13) Interrupt stack





two memory layouts

the first is the traditional stack layout.

the second layout has the advantage
that when overflow occurs,
the vector table remains untouched
so the system has the chance
to correct itself.

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

Multiple stacks

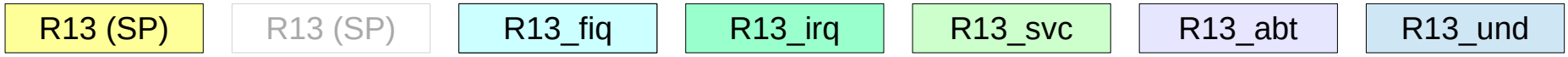
Interrupt code typically uses stacks.
there is a separate **R13** (**SP**) for each mode (except one - system mode).

So there is a separate stack per mode

at the startup, each **SP** needs to be initialized.

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

```
MSR   CPSR_c, R1 ; Mode1
MOV   SP, R0
MSR   CPSR_c, R2; Mode2
```



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

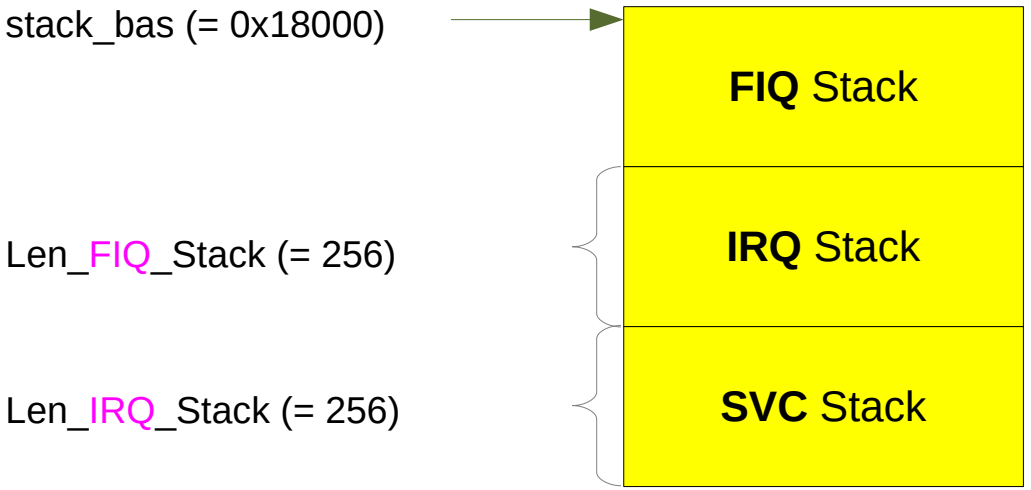
https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

Stack pointer initialization

The stack_bas symbol can be a hard-coded address,
or it can be defined in a separate assembler source file
and located by a scatter file.

```
stack_bas          DCD    0x18000
Len_FIQ_Stack      EQU    256
Len_IRQ_Stack      EQU    256
```

the example allocates 256 bytes of stack
for Fast Interrupt Request (FIQ) and
Interrupt Request (IRQ) mode,
but you can do the same for any other execution mode.



https://developer.arm.com/documentation/dui0471/m/embedded-software-development/stack-pointer-initialization

Stack pointer initialization

To set up the stack pointers,
enter each mode with interrupts disabled,
and assign the appropriate value to the stack pointer.

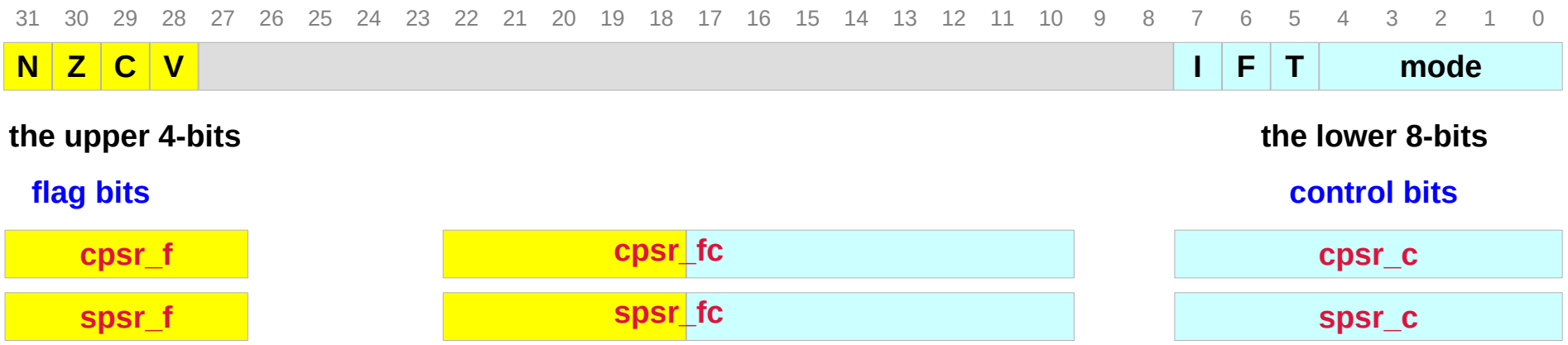
```
Len_FIQ_Stack    EQU    256
Len_IRQ_Stack    EQU    256
stack_bas        DCD    0x18000
```

The stack pointer value set up in the reset handler
is automatically passed as a parameter
to __user_initial_stackheap() by C library initialization code.

Therefore, this value must not be modified by
__user_initial_stackheap()

https://developer.arm.com/documentation/dui0471/m/embedded-software-development/stack-pointer-initialization

CPSR_c, CPSR_f, CPSR_fc



Current Program Status Register (CPSR)

Saved Program Status Register (SPSR)

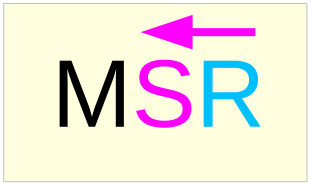
To disable Interrupt (IRQ), set **I**

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

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

https://www.sciencedirect.com/topics/computer-science/software-interrupt

Stack pointer initialization




```
; *************************************************************
; This example does not apply to ARMv6-M and ARMv7-M profiles
; *************************************************************
Len_FIQ_Stack       EQU    256
Len_IRQ_Stack       EQU    256
stack_bas           DCD     0x18000

; Reset_Handler                  ; stack_base could be defined above, or located in a scatter file
   LDR     R0, stack_base ;      ; Enter each mode in turn and set up the stack pointer
   MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit     ; FIQ mode, Interrupts disabled
   MOV     sp, R0                                  ; SP_fiq initialization

   SUB     R0, R0, #Len_FIQ_Stack
   MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit     ; IRQ mode, Interrupts disabled
   MOV     sp, R0                                  ; SP_irq initialization

   SUB     R0, R0, #Len_IRQ_Stack
   MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit     ; SVC mode, Interrupts disabled
   MOV     sp, R0                                  ; SP_svc initialization

   ; Leave processor in SVC mode
```

https://developer.arm.com/documentation/dui0471/m/embedded-software-development/stack-pointer-initialization

Stack pointer initialization

```
MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit

        MRS     r1, cpsr
        ORR     r1, r1, #0xD1
        MSR     cpsr_c, r1

MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit

        MRS     r1, cpsr
        ORR     r1, r1, #0xD2
        MSR     cpsr_c, r1

MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit

        MRS     r1, cpsr
        ORR     r1, r1, #0xD3
        MSR     cpsr_c, r1
```

I	F	T		mode					
1	1	0	Usr (usr)	1	0	0	0	0	0x10
1	1	0	Fast Interrupt (fiq)	1	0	0	0	1	0x11
1	1	0	Interrupt (irq)	1	0	0	1	0	0x12
1	1	0	Supervisor (svc)	1	0	0	1	1	0x13
1	1	0	Abort (abt)	1	0	1	1	1	0x17
1	1	0	Undefined (und)	1	1	0	1	1	0x1D
1	1	0	System (sys)	1	1	1	1	1	0x1F

https://developer.arm.com/documentation/dui0471/m/embedded-software-development/stack-pointer-initialization

CPS instruction

You can't change between modes using instructions which directly write to the CPSR mode bits in User mode. proper way is to use a **svc** (supervisor call) and execute necessary instruction requested.

CPS (Change Processor State) Instruction

only permitted in privileged software execution, and has no effect in User mode.

CPSIE – Interrupt or Abort Enable

CPSID – Interrupt or Abort Disable

a : Enables or disables imprecise aborts.

i : Enables or disables IRQ interrupts.

f : Enables or disables FIQ interrupts.

CPSIE if

; enable interrupts and fast interrupts

CPSID A

; disable imprecise aborts

CPSID ai, #17

; disable imprecise aborts and interrupts, and enter FIQ mode

CPS #16

; enter User mode

https://stackoverflow.com/questions/20653025/msr-cpsr-c-0x13-doesnt-work-using-arm-assembly

SWI (Software Interrupt)

- Some ISAs, including ARMv4, have a special **SWI** instruction that, when executed, causes the system to act like when a hardware device requested an interrupt.
- A hardware interrupt is like an unscheduled subroutine call that also puts the processor into an more privileged mode.
- Handler code is trusted and part of the operating system.
- So an **SWI** instruction is often used to invoke an OS service subroutine.

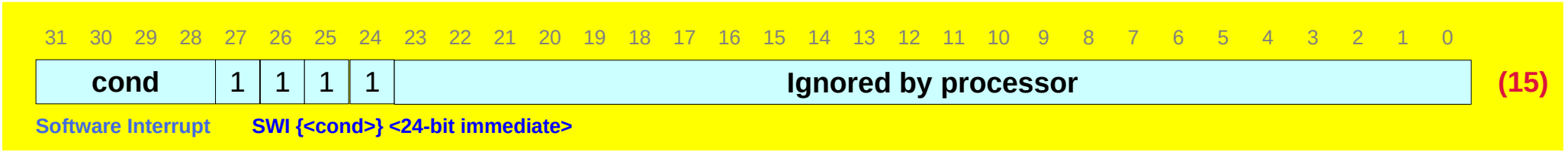
http://www2.unb.ca/~owen/courses/2253-2017/slides/08-interrupts.pdf

SWI / SVC opcode

- the **SWI** instruction is called its new name, **SVC**

- **SWI** and **SVC** are same thing, it is just a name change.
  Previously, the **SVC** instruction was called **SWI**, Software Interrupt.

- the opcode for **SVC** (and SWI) is partially user defined
- bit 0-23 is user defined and is like a parameter to SVC handler
- bits 24-27 are b1111 and these 4 bits makes CPU
  to realize that the opcode is **SVC** (or **SWI**).



https://stackoverflow.com/questions/8459279/are-arm-instructuons-swi-and-svc-exactly-same-thing

SWI (Software Interrupt)

Software can generate an exception.
Use **SWI** to request an operating-system service.

SWI handler has to use the value in R14
to find the actual instruction,
in order to extract the “SVC number”
field and thus know which OS service was requested.

```
LDR     r10,  [lr, #-4]          ; Read the SWI instruction
BIC     r10,  r10, #0xff000000   ; Mask off top 8 bits
BL      service_routine          ; r10 - contains the SWI number
```

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

Error Exceptions

- **Undefined instruction**
  - can be intentional (emulate a “missing” instruction)
- **Prefetch abort**
  - an attempt to fetch instruction fails
    (eg, PC is not a valid memory location)
- **Data abort**
  - A LD or ST with an illegal address
  - A store to a read-only address

- Sometimes, the response should be to die gracefully. But other times, we may be able to recover and continue.

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

Overall Approach

For an exception, we need to

- save the current state (including CPSR)
- reset the PC to the handler code [& change mode]
- execute the handler
- restore the saved state, including the PC & mode

State saving and PC resetting are done by hardware.
Handler and restoring done by software.

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

Priorities in IRQ

- Even within a given exception (eg **IRQ**),
  some hardware units (eg disk) are more urgent
  than others (eg keyboard).
- To prioritize, could check all interrupt request inputs
  Then software can check each possible device
  to see who's knocking...starting with the most urgent.
- Or a special priority device, a VIC, can take care of this.
  - devices' IRQ lines go to VIC
  - only VIC actually interrupts CPU
  - CPU can ask VIC for the handler address
    of the highest priority active interrupt request.
    [talking to devices: stay tuned!]

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

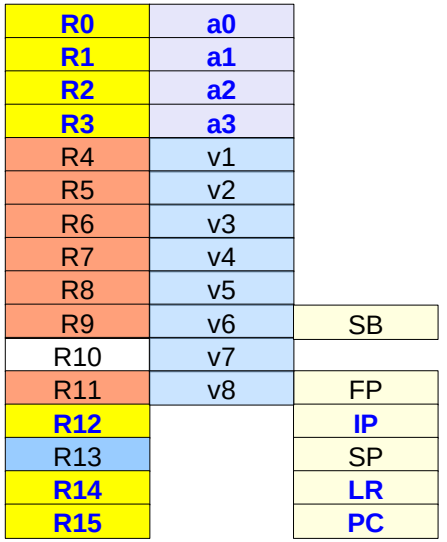
https://www.ic.unicamp.br/~celio/mc404-2013/arm-manuals/ARM_exception_slides.pdf

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.



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.

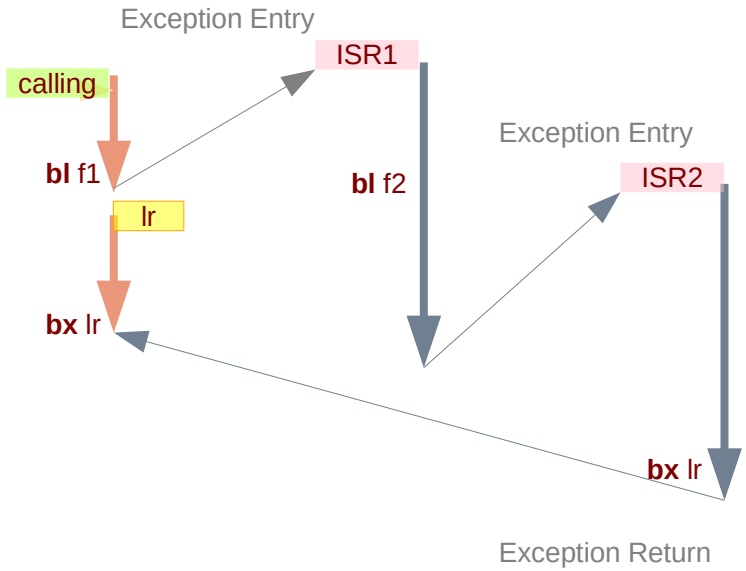
- Initialization via a **MSR** (move into status register) instruction to change mode.
- Then store a value to (that) SP.
- Then use a **MSR** to put mode back

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

Exception entry and return sequence (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

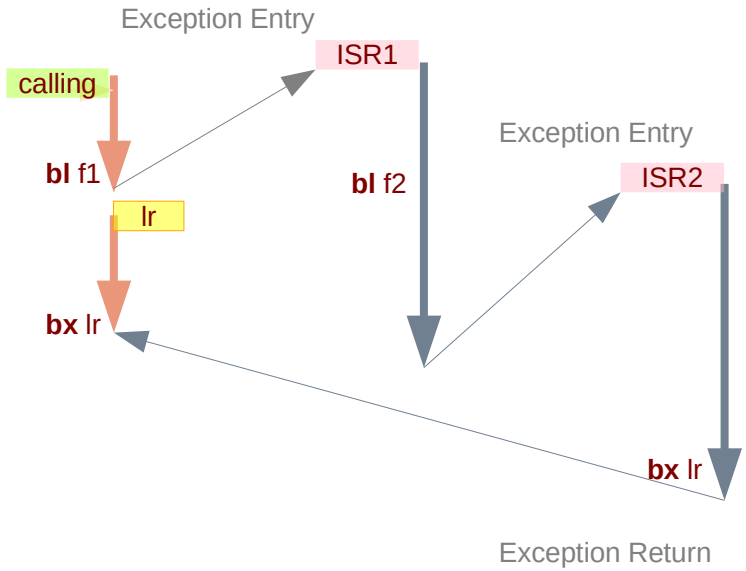
Exception entry and return sequence (7)

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)

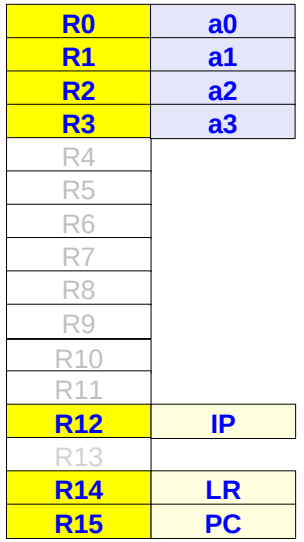
it's possible that another interrupt will be handled by tail-chaining.

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.



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

Finding the right handler

For the different kinds of exceptions, there are different handlers. When an exception occurs, the hardware determines the source of the exception as a 3-bit number, which it uses to index the vector table (which starts in memory at address 0)

0x1c	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0c	Prefetch Abort
0x08	SWI
0x04	Undefined Instruction
0x00	Reset

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

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