Vectored Interrupt Programming

Young Won Lim
6/9/23

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/

Standard Interrupt Controller

The **standard** interrupt controller
sends an interrupt signal to the processor core
when an external device requests servicing.

It can be programmed to ignore or mask
an individual device or set of devices.

The interrupt handler determines
which device requires servicing
by reading a device bitmap register
in the interrupt controller.

standard *interrupt controller*

programmable mask

interrupt source
by reading a device register

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

Vectored Interrupt Controller

The VIC is more powerful
than the standard interrupt controller

- prioritizes interrupts
- simplifies the determination of interrupt source
  (of which device caused the interrupt)

- a priority is associated with a handler address
  for each interrupt request

- the VIC asserts an interrupt signal to the core
  only when the priority of a new interrupt is higher than that of the currently executing interrupt handle

VIC = IVT + Priority

Vectored Interrupt Controller (VIC)
Interrupt Vector Table (IVT)

priority – handler address

preemptive interrupt handling

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

Multiple ISR handlers

Usually in the old generation controllers,
there is only one ISR
that handles multiple interrupt sources.

the ISR checks the particular register
to find the interrupt source
– who is interrupting the processor.

large interrupt latency

to reduce interrupt latency,
ARM has come up with an idea of
a vector interrupt controller (**VIC**)
where each interrupt can have separate ISR's

each ISR address will be
stored in the **Interrupt Vector Table**.

Default ISR

ISR determines the interrupt source

One ISR – multiple interrupt sources

Large latency

Separate ISR's, IVT

IRQ source 1	-----	*ISR 1 address*
IRQ source 2	-----	*ISR 2 address*
IRQ source 3	-----	*ISR 3 address*
…		…

If IRQ source i, *then jump to ISRi*

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

IRQ sources and ISR addresses

The VIC provides a software interface
to the interrupt system.

In a standard interrupt controller,
software must determine
    the **source** that is requesting service
    where its **ISR** is loaded.

SW (ISR) determines the interrupt source

Jump address must be loaded

In a vectored interrupt controller,
hardware supplies
    the **starting address**,
    or vector address, of the ISR
    corresponding to the interrupt **source**
    that has the highest priority

HW determines the interrupt source

A table lookup of IVT provides
the jump address (the specific ISR)

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

Interrupt Vector Table

Interrupt vector table (IVT)

contains the address of the IRQ handlers
of every interrupt.

directs the PC where to go,
when an interrupt occurs.

refers early generation of VIC,
because they just point the address
when an interrupt occurs.

priority was not fully applied

VIC = IVT + Priority

Old VIC = IVT only, no 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-proc

VIC interrupt handling types (1)

the VIC interrupt handling types

- make the core jump directly to the handler address for the device **(Vectored IRQ)**
- either call the standard interrupt exception handler, which can load the handler address for the device from the VIC **(Non-Vectored IRQ)**

VIC = IVT + Priority

Vectored IRQ
unique ISR
jump to the address

Non-vectored IRQ
default ISR
load the address

in lpc214x

VICVectAddr	: holds the address of the associated **ISR** i.e the one which is currently active.
VICDefVectAddr	: stores the address of the "default/common" ISR for a **Non-Vectored IRQ** occurs

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

VIC interrupt handling types (2)

VIRQ (Vectored IRQ) has
dedicated IRQ service routine
for *each* Vectored interrupt source

NVIRQ (Non-Vectored IRQ) has
the same IRQ service routine
for *all* Non-Vectored Interrupts.

VIC = IVT + Priority

Vectored IRQ
dedicated ISR for each IRQ source

Non-vectored IRQ
default ISR for all IRQ sources

in lpc214x

VICVectAddr	: holds the address of the associated **ISR** i.e the one which is currently active.
VICDefVectAddr	: stores the address of the “default/common” ISR for a **Non-Vectored IRQ** occurs

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

VIC interrupt handling types (3)

Vectored means that
the CPU is <u>aware</u> of the address of the ISR
when the interrupt occurs

Non-Vectored means that
CPU <u>doesn't</u> <u>know</u> the address of the ISR
nor the source of the IRQ
when the interrupt occurs
it needs to be <u>supplied</u> with the ISR address.

For the Vectored Interrupt Controller,
the system internally maintains a table
IVT (**Interrupt Vector Table**)

which contains the information
about Interrupts sources
and their corresponding ISR address.

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

VIC = IVT + Priority

Vectored IRQ
ISR address is known
<u>jump</u> to the address

Non-vectored IRQ
IRQ address <u>not</u> known
<u>load</u> the address

IRQ source 1	*ISR 1 address*
IRQ source 2	*ISR 2 address*
IRQ source 3	*ISR 3 address*
…	…

ARM FIQ

In an ARM system, two levels of interrupts are available:

Fast Interrupt reQuest (**FIQ**)
– For fast, low latency interrupt handling.

Interrupt ReQuest (**IRQ**)
– For more general interrupts.

- a single FIQ source system for a low-latency interrupt
  - the ISR is executed directly
    without determining the source of the interrupt
  - this reduces the interrupt latency
- the banked registers of FIQ mode
  can be used more efficiently,
  without incurring a context save overhead

Single FIQ source system

Register bank

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

3 categories of IRQ's

The ARM Vectored Interrupt Controller (VIC)
takes 32 interrupt request inputs and
programmably assigns them into 3 categories,

- **FIQ**
- vectored **IRQ**
- non-vectored **IRQ**.

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

Nested Vectored Interrupt Controller

A Nested Vectored Interrupt Controller (NVIC)
is used to manage the interrupts
from multiple interrupt sources.

NVIC is closely integrated with the processor core
to achieve low-latency interrupt processing and
efficient processing of late arriving interrupts.

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

NVIC vs. VIC (1)

every interrupt with certain priority levels

each interrupt is serviced / processed
with its own priority level.

Servicing / processing the interrupt means
the processing of the part of codes
inside the IRQ handler
of the respective interrupt.

Interrupt handling of

- Nested Vectored Interrupt Controller (NVIC)
- Vectored Interrupt Controller (VIC)
- Interrupt Vector Table (IVT)

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-proc

NVIC vs. VIC (2)

Example assumption :

Priority 1 (P1) - highest
Priority 2 (P2) - second highest

There are two different interrupts X and Y
with priority levels P1 and P2 respectively.

- interrupts X and Y occur at the same time.
- interrupt Y (P2) has occured first and
  while servicing interrupt Y (P2)
  interrupt X (P1) occurs

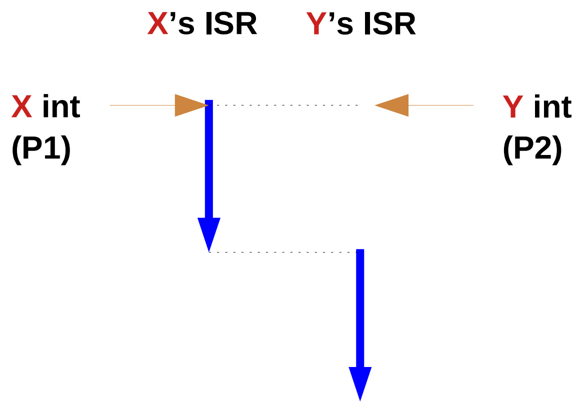
https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proc

Nested VIC handling (1)

- If interrupts X and Y occur at the same time.

  first X (P1) is processed,
  Y (P2) is put on hold.

  After processing X,
  Y is processed.



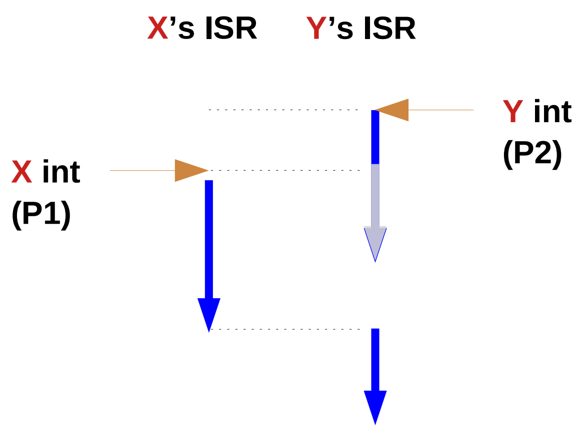
https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proc

Nested VIC handling (2)

- If interrupt Y (P2) has occured first and
  interrupt X (P1) occurs
  while servicing interrupt Y (P2)

  Then, the controller puts
  the interrupt Y's IRQ handler on hold
  and processes
  the interrupt X's IRQ handler completely
  and then resumes
  the interrupt Y's IRQ handler

  So, it processes interrupt
  by nesting them within each other.



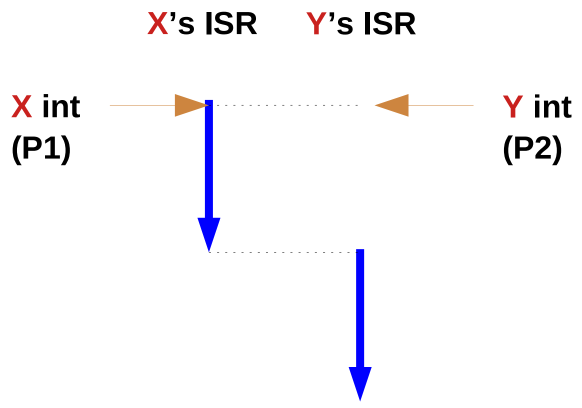
https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proce

VIC handling (1)

- If interrupts X and Y occur at the same time.

  first X (P1) is processed,
  Y (P2) is put on hold.

  After processing X,
  Y is processed.



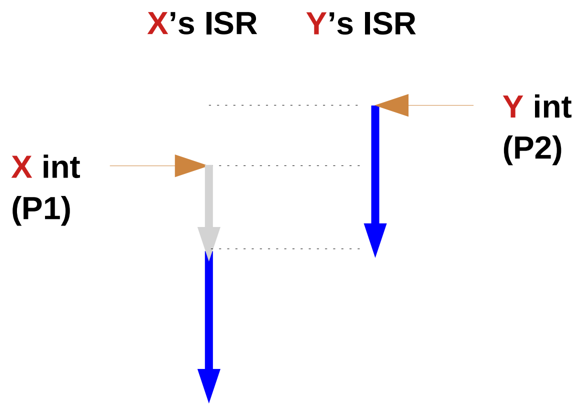
https://www.quora.com/What-is-the-difference-between-ARMs-nested-vectored-interrupt-controller-and-an-interrupt-vector-table-which-seems-to-be-used-by-the-other-proc

VIC handling (2)

- If interrupt Y (P2) has occured first and
  interrupt X (P1) occurs
  while servicing interrupt Y (P2)

  Then, the controller processes
  the interrupt Y's IRQ handler completely

  and then the processes
  the interrupt X's IRQ handler



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-proc

NVIC features in cortex M (1)

- external interrupts (1 ~ 240)
- bits of priority (3 ~ 8)
- a dynamic **re-prioritization** of interrupts.
- **priority grouping** enables the selection of

  preempting interrupt levels

  non-preempting interrupt levels.
- support for **tail-chaining** and **late arrival** of interrupts.

  This enables back-to-back interrupt processing

  without the overhead of state saving and restoration

  between interrupts.

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

NVIC features in cortex M (2)

- processor state **automatically**

  saved on interrupt **entry**,

  restored on interrupt **exit**,

  with no instruction overhead.
- Optional Wake-up Interrupt Controller (WIC),

  providing ultra-low-power sleep mode support.
- Vector table can be located in either RAM or flash.

All interrupts including the core exceptions
are managed by the NVIC.

The NVIC maintains knowledge of
the stacked, or nested, interrupts
to enable tail-chaining of interrupts.

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

Nesting, Tail Chaining, and Late Arrival

- **preemption**
  interrupts the context
  by pushing registers onto a stack
  and popping them later
  to return to the interrupted context

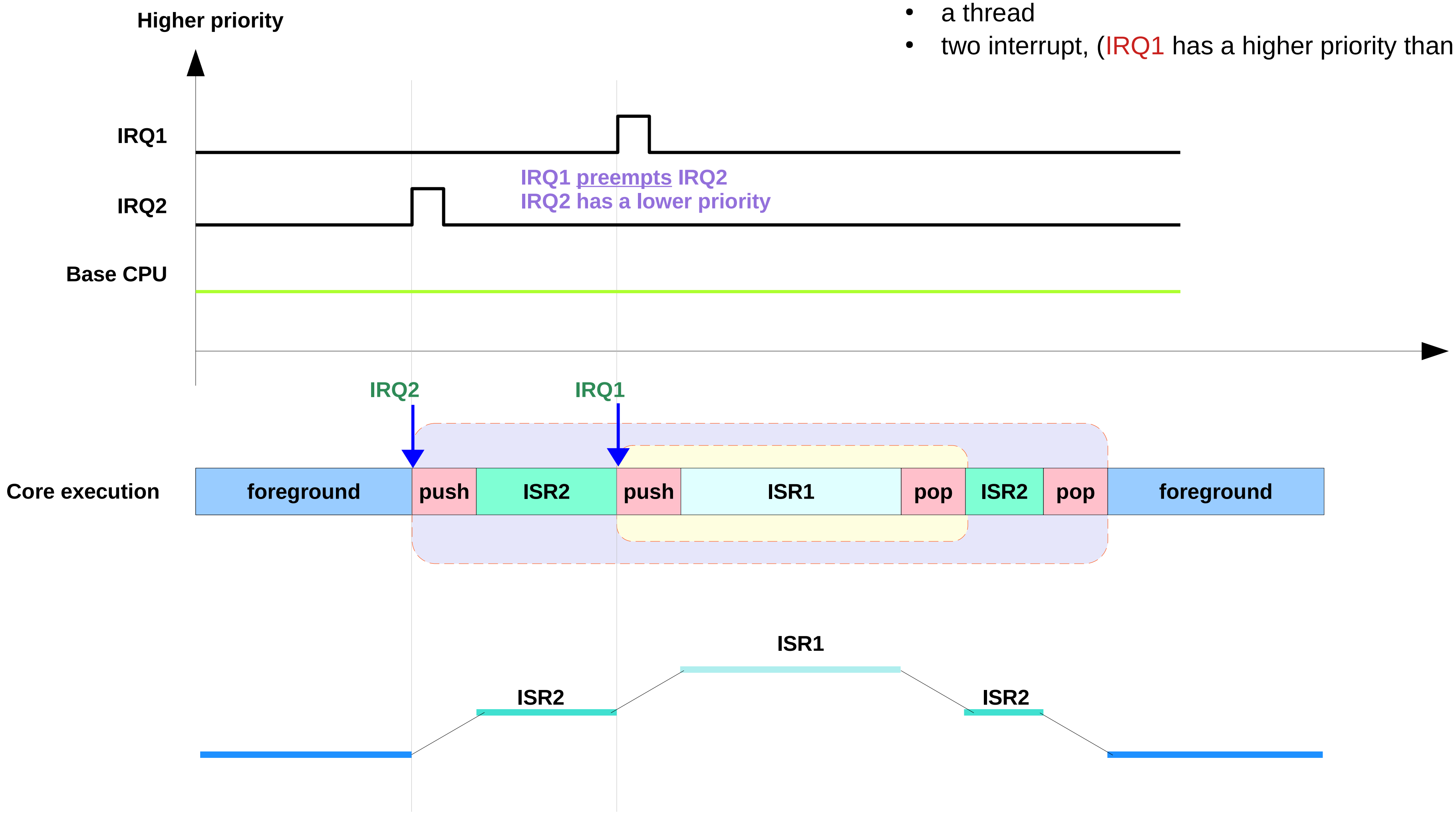
- **tail-chaining**
  allows additional handlers to be executed
  without additional pushing and popping of registers.

consider a diagram
    priority on the vertical axis
    time on the horizontal.

https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Nesting (1)

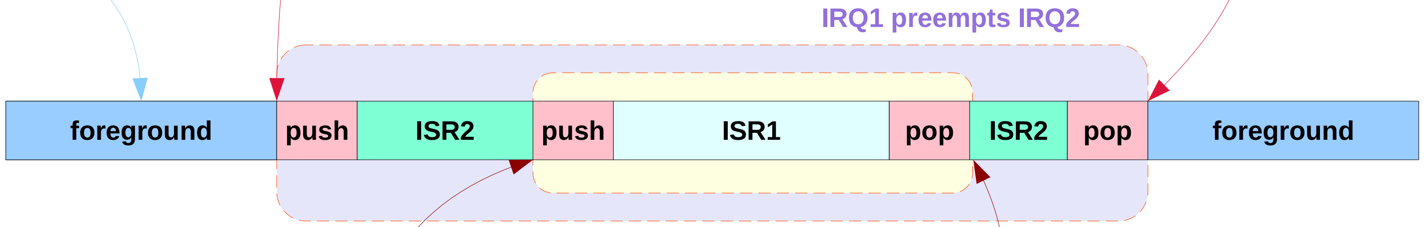
- a thread
- two interrupt, (IRQ1 has a higher priority than IRQ2)



https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Nesting (2)

- initially, the thread is running at base priority level.
- at some point IRQ2 is requested and the thread is immediately preempted by pushing it onto the stack, and start running the ISR2
- When ISR2 completes, we pop back to the thread.



- when ISR1 completes, we pop back to the next highest priority ISR2.
- while ISR2 is active, IRQ1 requests since IRQ1 has a higher priority than IRQ2, ISR2 is also preempted and pushed onto the stack, and ISR1 is executed.

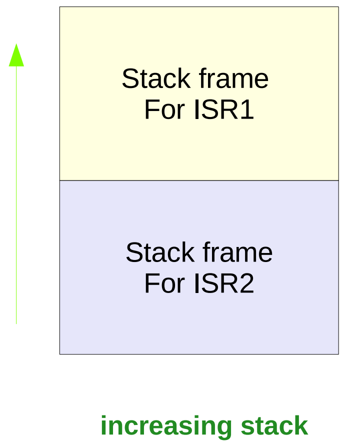
https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

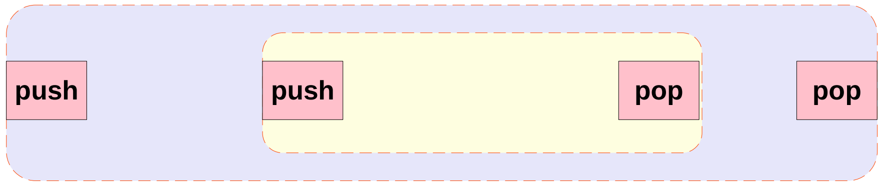
Nesting (3)

- The benefit
    - *distinct* levels of priority
    - *always* working on the most important task
    - *minimize* the interrupt latency
      for the highest priority interrupt at any time.

- The cost
  - a few cycles performing housekeeping (**push**, **pop**)
  around the interrupts.

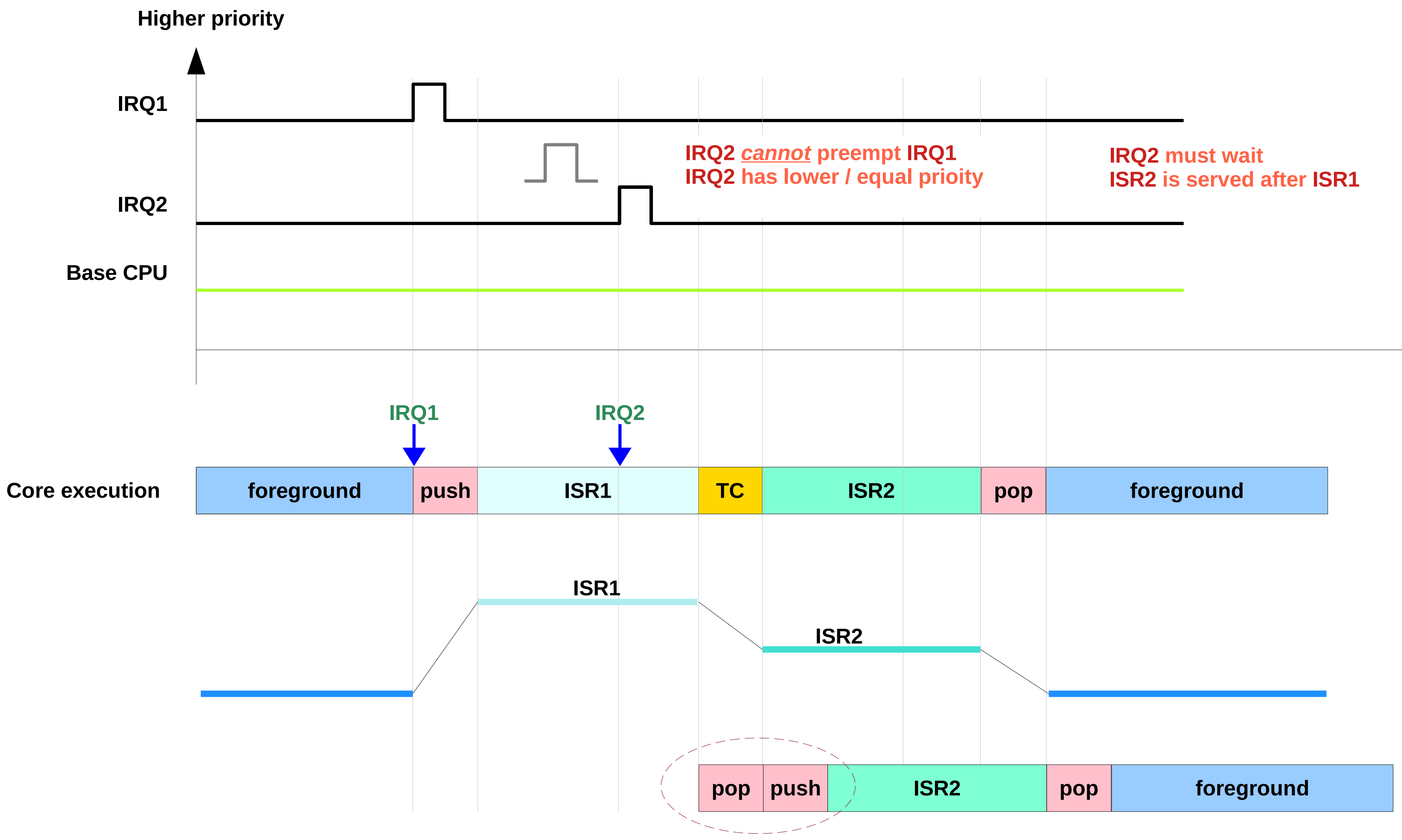
- creating *multiple* stack frames
  increases the need for stack memory
  consumes energy for several memory cycles





https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Tail chaining (1)



https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

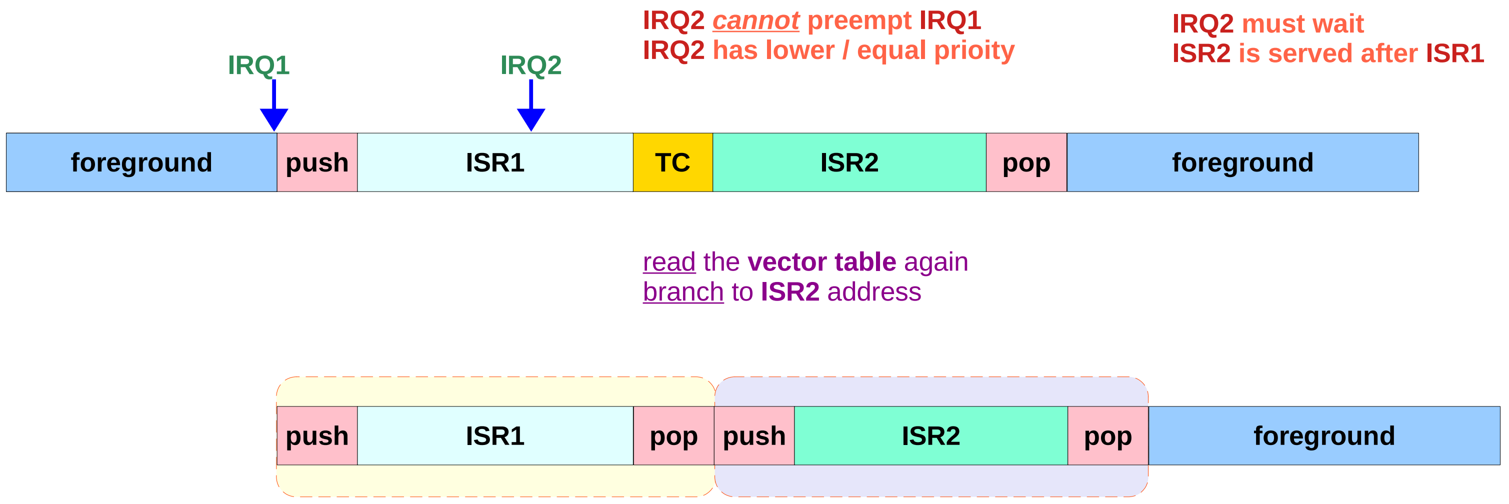
Tail chaining (2)

- Priority (IRQ2) ≤ Priority (IRQ1)
  thus IRQ2 cannot preempt IRQ1.

- IRQ1 preempts the thread with a stack push.
- while ISR1 runs, IRQ2 occurs,
  - IRQ2 remains pending
  - ISR1 runs to completion

- At the end of ISR1,
  the NVIC then arbitrates to IRQ2 and runs ISR2
  simply by *reading* the vector table again and
  branching to that address.

- Only when ISR2 is *completed*
  and there are no other pending interrupts,
  the stack popped to return to the thread.

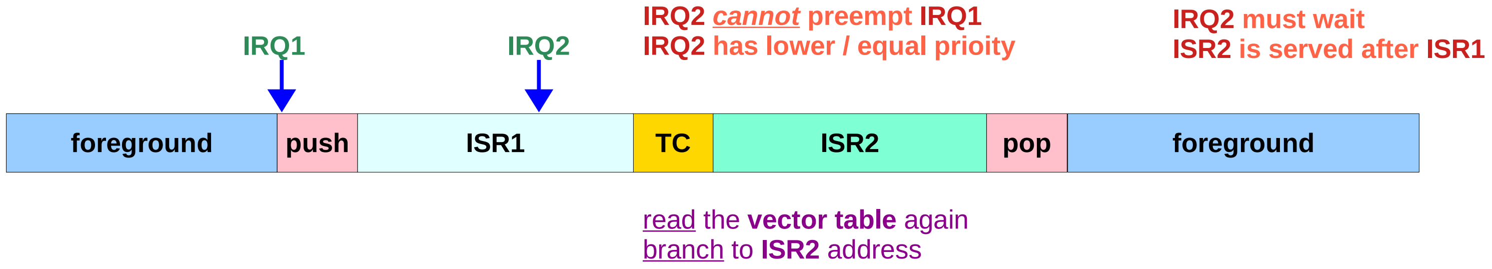


https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Tail chaining (3)

- Priority (IRQ2) ≤ Priority (IRQ1)
  thus IRQ2 cannot preempt IRQ1.

- At the end of ISR1, (**tail chaining**)
  the NVIC then arbitrates to IRQ2 and runs ISR2
  simply by *reading* the vector table again and
  branching to that address.


IRQ1
IRQ2
IRQ2 *cannot* preempt IRQ1
IRQ2 has lower / equal prioity
IRQ2 must wait
ISR2 is served after ISR1
foreground
push
ISR1
TC
ISR2
pop
foreground
read the vector table again
branch to ISR2 address

- In this case, there was less *control* of interrupt latency.
  - cannot preempt, must wait

- as any lower or equal priority interrupt
  that occurred while another interrupt was active,
  would have to wait for that active ISR to complete.

https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

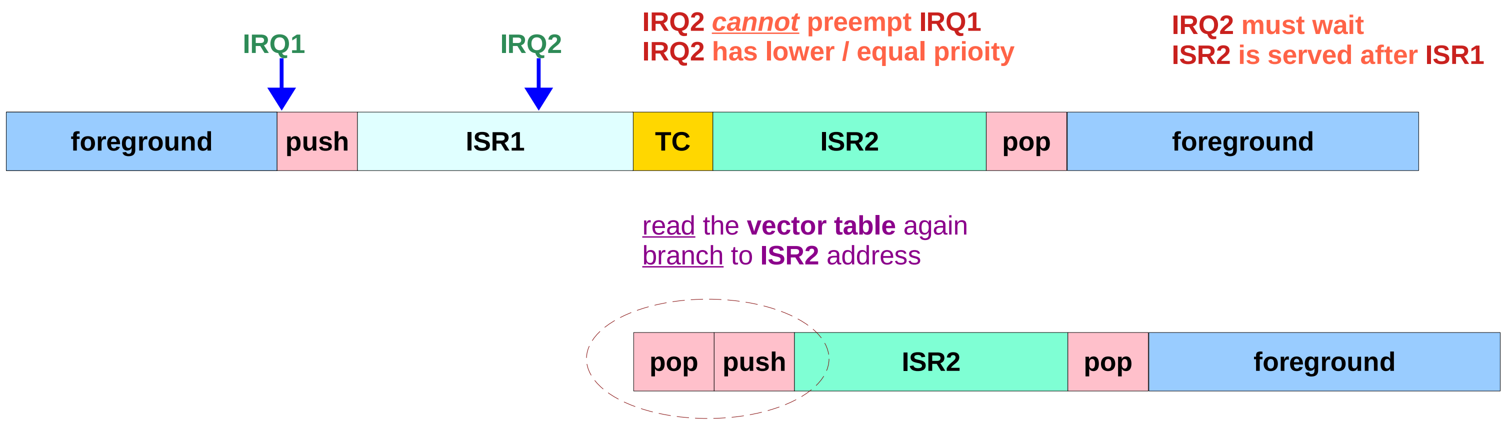
Tail chaining (4)

to perform the housekeeping between interrupts

- *fewer cycles* were spent
- *less energy* used
- *less memory space* used

these lead to

- *better overall throughput*
- *lower power*
- *smaller memory requirements*

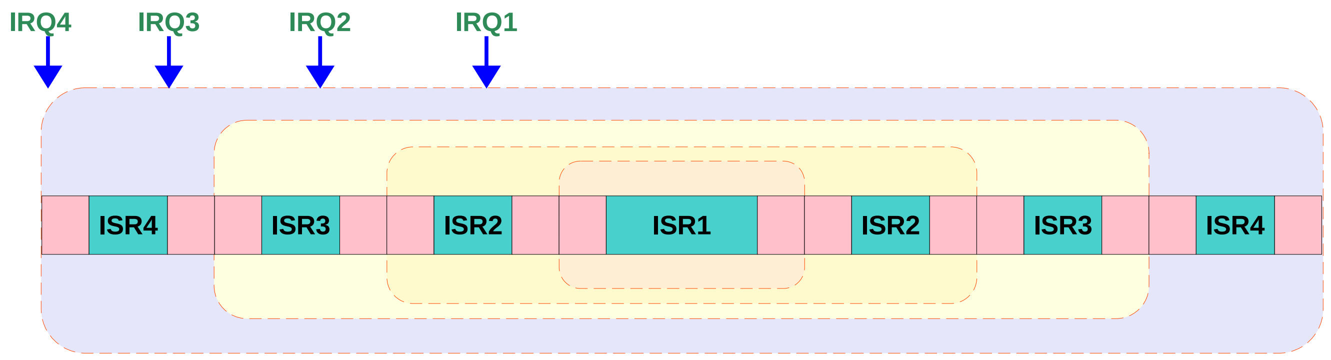


https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

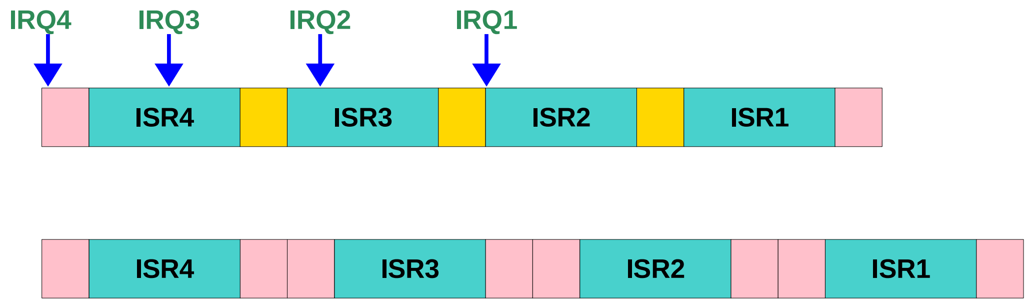
Tail chaining (5)

- ARM recommends programming interrupts into
  as *few* priority levels as needed,
  and therefore, using tail-chaining as widely as possible
  to take advantage of these benefits.

Priority (IRQ4)
< Priority (IRQ3)
< Priority (IRQ2)
< Priority (IRQ1)
4 distinct priority levels

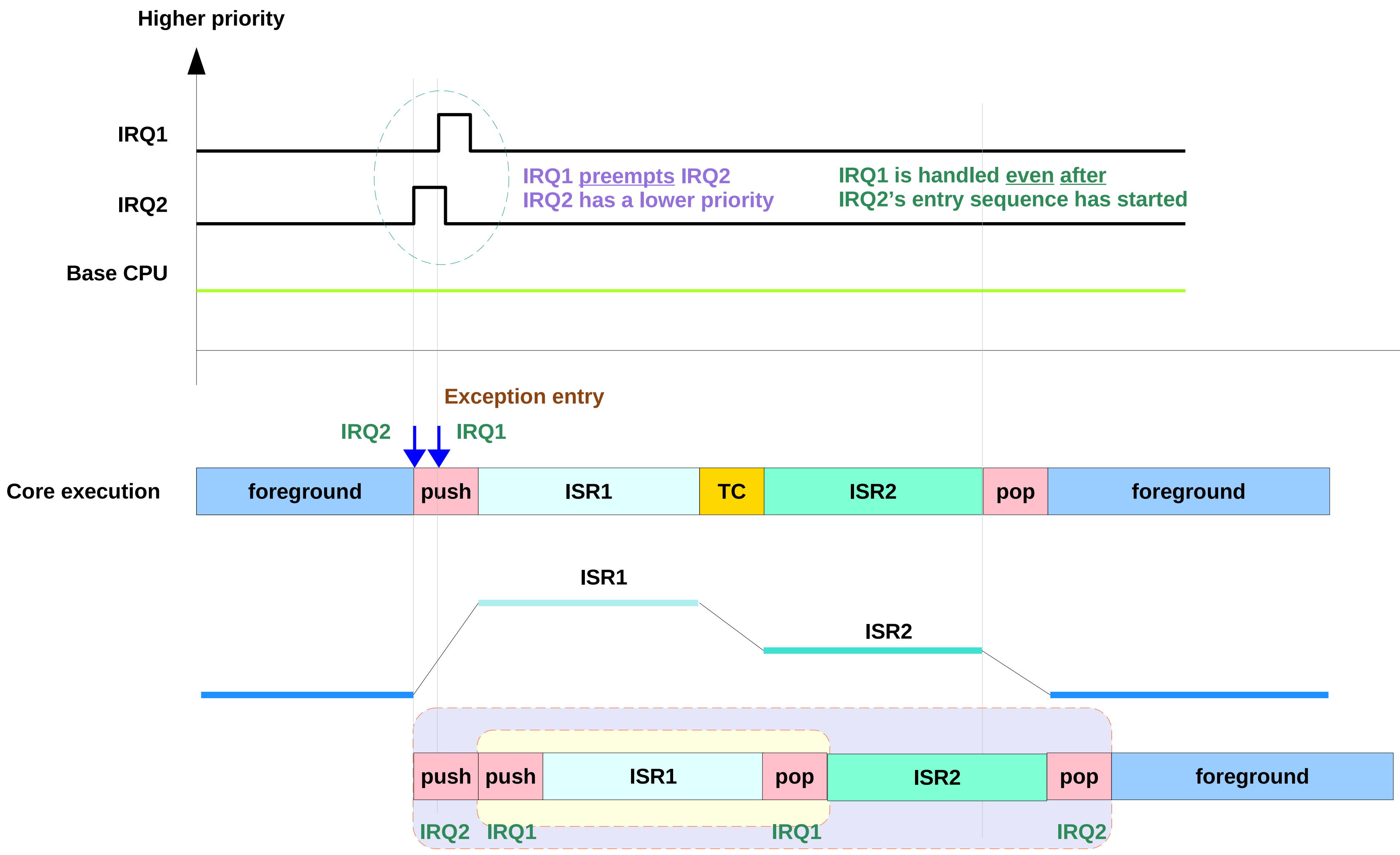


Priority (IRQ4)
= Priority (IRQ3)
= Priority (IRQ2)
= Priority (IRQ1)
the same priority level



https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

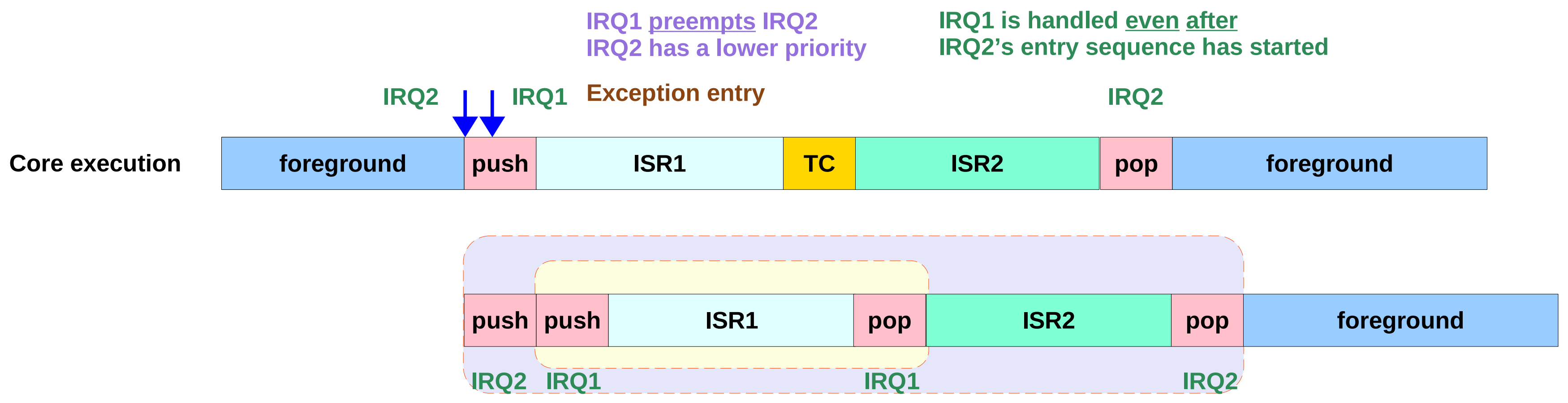
Late arrival **A** (1)



https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Late arrival **A** (2)

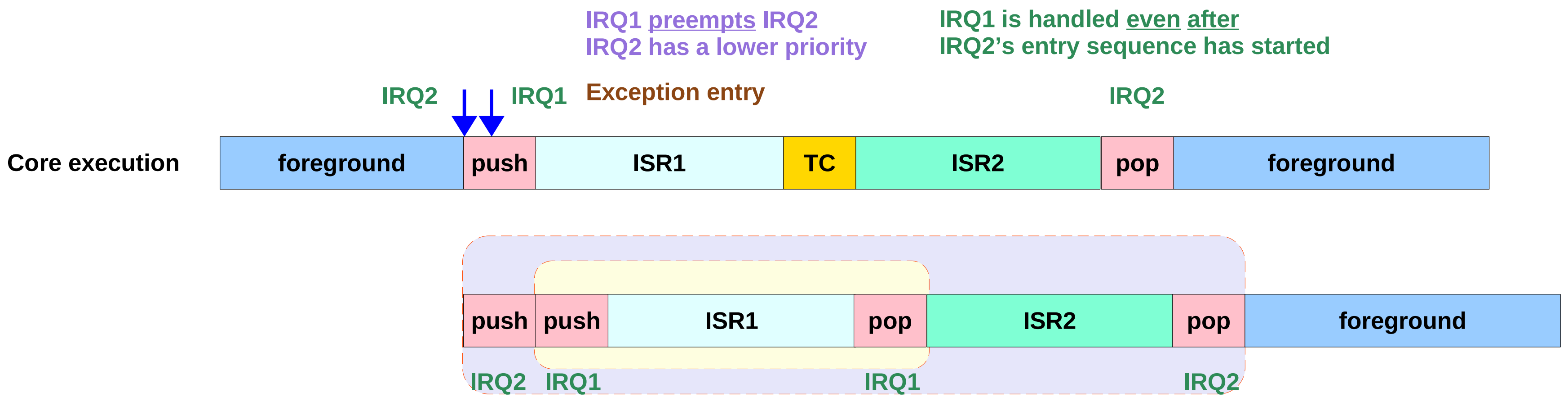
- a higher priority exception is handled before a lower priority exception
- just after the entry sequence of a lower priority exception has started
- the lower priority exception is handled after the higher priority exception is completed



https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

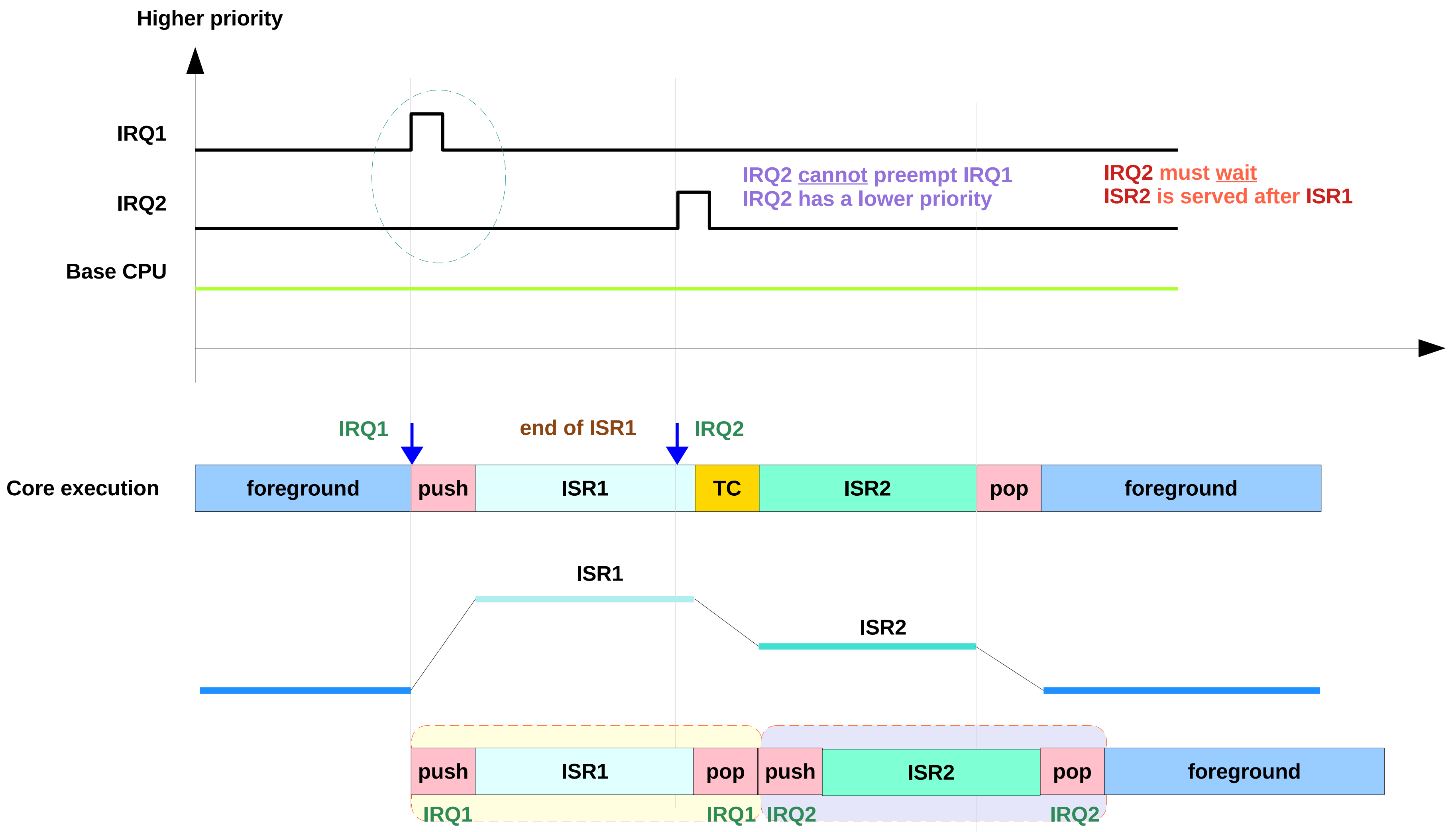
Late arrival **A** (3)

- also, in the case of the late-arriving interrupt, the processor might execute its ISR <u>after</u> <u>fewer</u> cycles of interrupt latency.
- a lower priority IRQ2 interrupt causes the interrupt entry sequence to start.
- the interrupted context has its registers pushed onto the stack.
- while this is happening, a higher priority IRQ1 interrupt occurs
- The processor still has to read the vector table to get the new vector
- but does <u>not</u> need to restart the stack push, so some cycles may be saved.



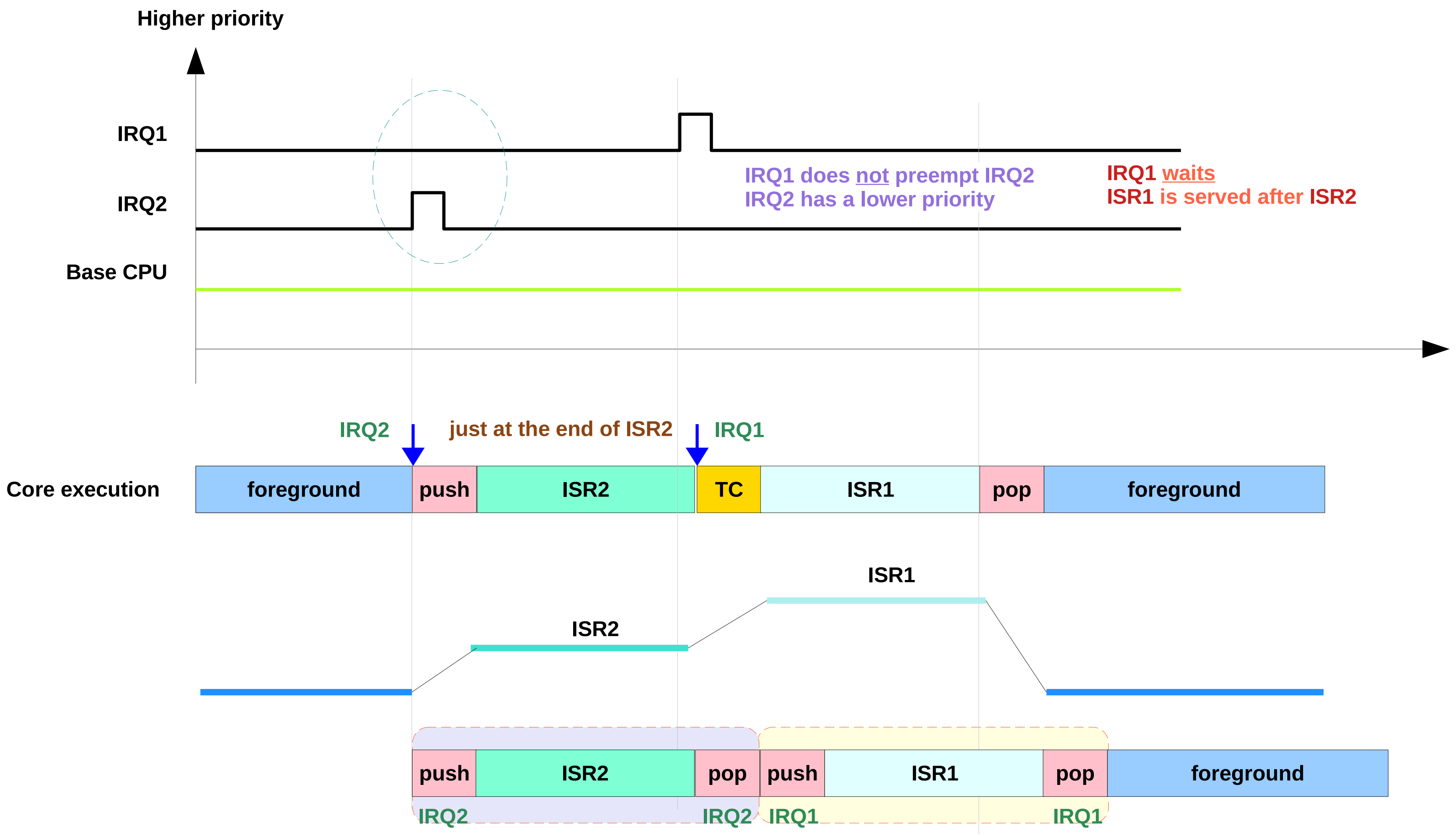
https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Late arrival **B** (1-1)



https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Late arrival **B** (1-2)



https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Late arrival **B** (2)

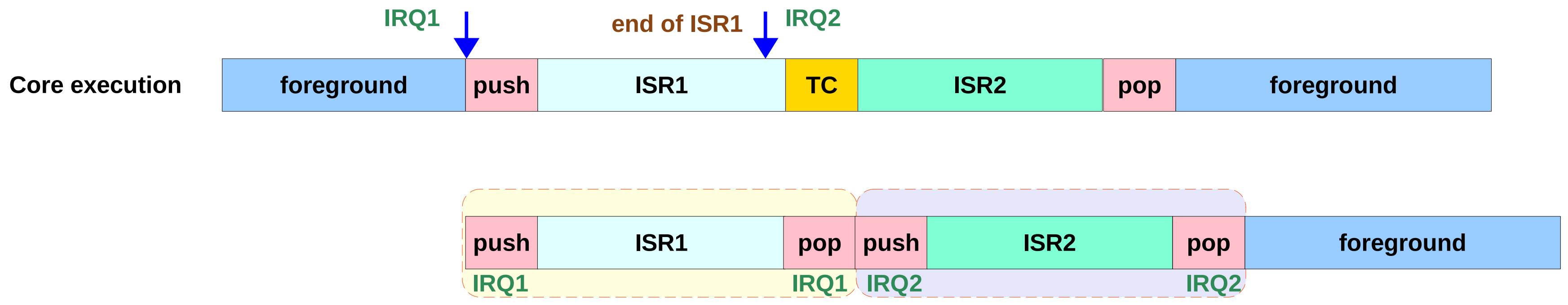
- A similar case arises
  if a new interrupt ISR2 arrives
  just before the end of an ISR1,

- Priority (IRQ2) **<** Priority (IRQ1)
- Priority (IRQ2) **>** any other pending or active ISR

- so that the newly detected interrupt
  immediately becomes the next interrupt
  to be handled in priority order.

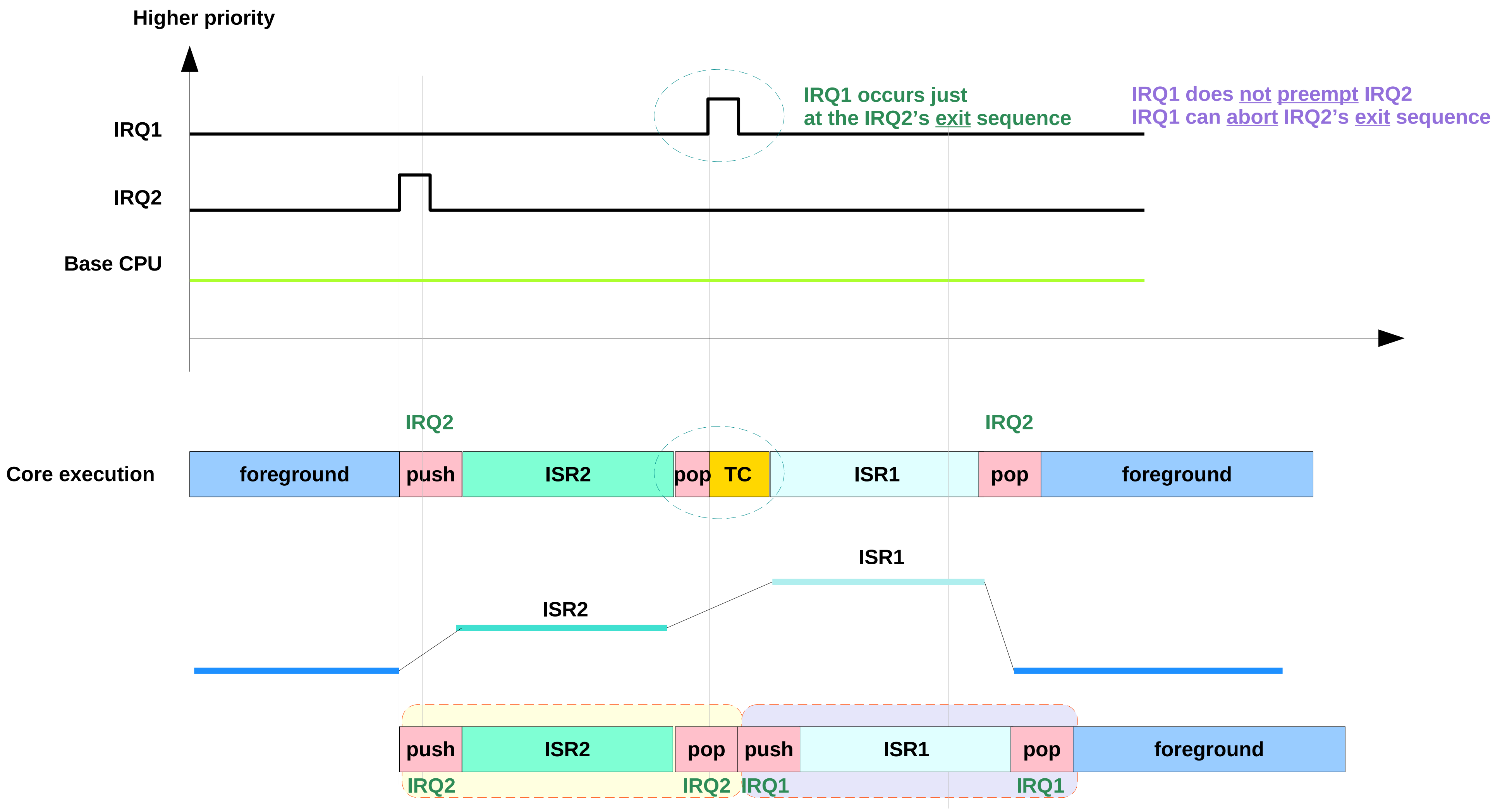
- Again, the vector table needs to be read
  to access the new ISR1,
  but tail-chaining does not require
  any stacking operation

- The interrupt latency could be
  lower than normal.



https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Late arrival **C** (1)



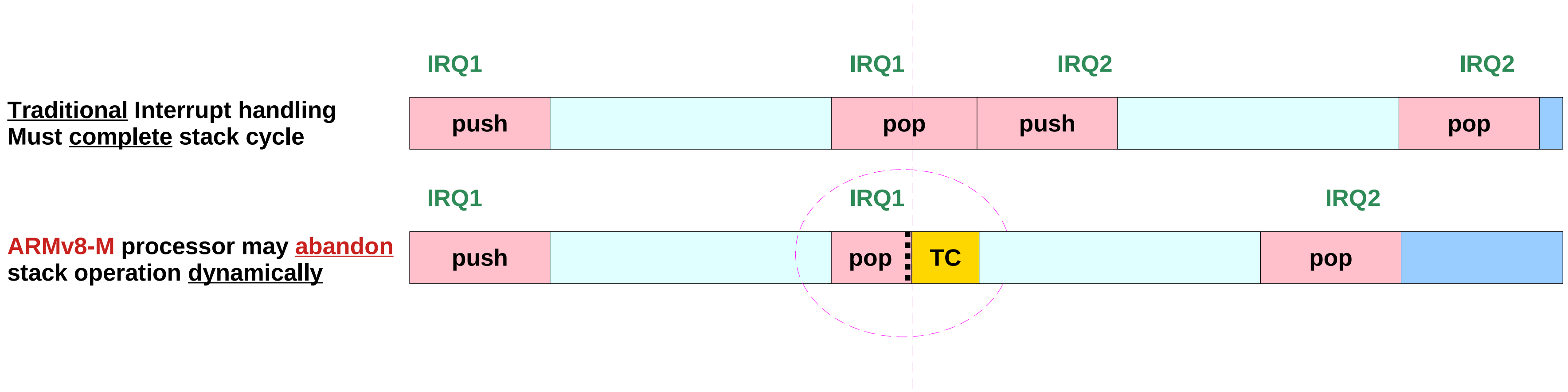
https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Late arrival **C** (2)

- In the case where
  the exception exit has already started,
  a similar situation arises.

- In the traditional model,
  the stack pop would have to complete,
  and then those same registers
  would need to be pushed again
  as part of the new exception handler.

- In **Cortex M**,
  the stack pop can simply be abandoned,
  leaving the stack frame on the stack,
  and only a tail-chain is then needed
  to enter the new ISR.

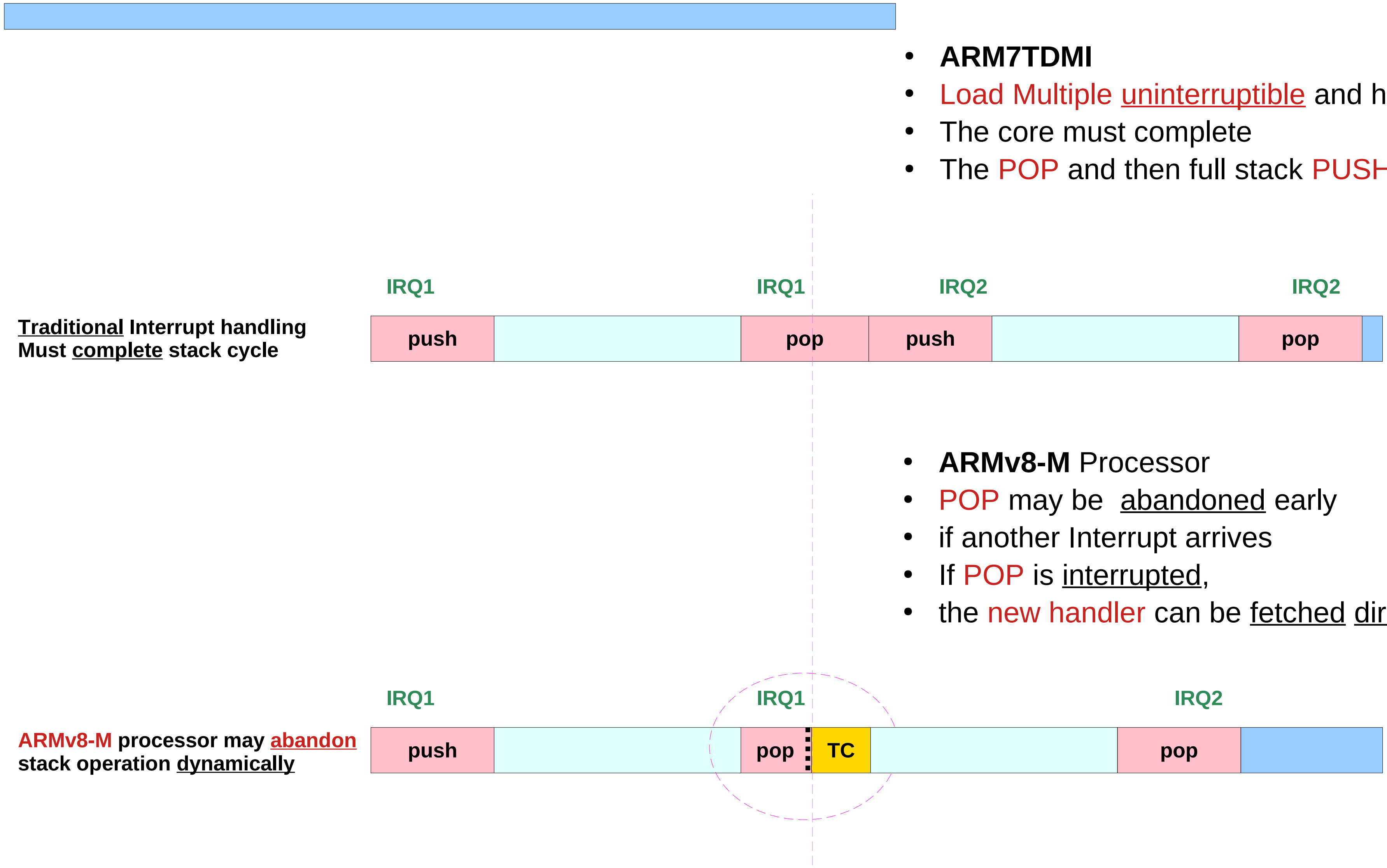


IRQ1
IRQ1
IRQ2
IRQ2
Traditional Interrupt handling
Must complete stack cycle
push
pop
push
pop
IRQ1
IRQ1
IRQ2
ARMv8-M processor may abandon
stack operation dynamically
push
pop
TC
pop

https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Late arrival **C** (3)

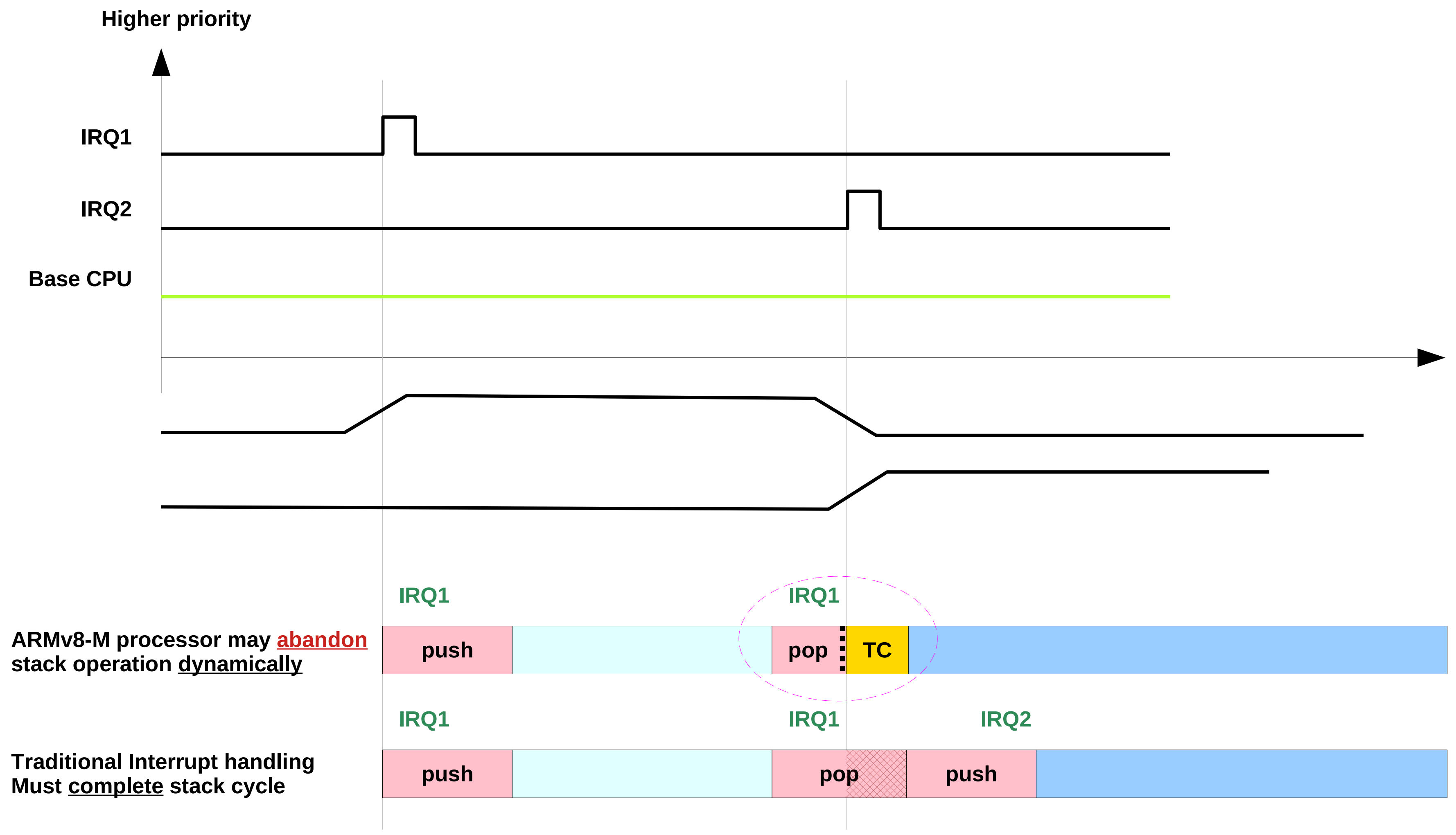
- **ARM7TDMI**
- Load Multiple uninterruptible and hence
- The core must complete
- The POP and then full stack PUSH


IRQ1
IRQ1
IRQ2
IRQ2
Traditional Interrupt handling
Must complete stack cycle
push
pop
push
pop
IRQ1
IRQ1
IRQ2
ARMv8-M processor may abandon
stack operation dynamically
push
pop
TC
pop

- **ARMv8-M** Processor
- POP may be abandoned early
- if another Interrupt arrives
- If POP is interrupted,
- the new handler can be fetched directly

https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Late arrival **C** (4)



ARMv8-M processor may abandon stack operation dynamically

Traditional Interrupt handling Must complete stack cycle

https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Nesting, Tail Chaining, and Late Arrival (2)

- The ARM-Architecture Reference Manual mentions
  three design options that can be implemented for **CortexM**.

- In the Instruction Set Attribute Register 2 (ID_ISAR2), bits[11:8]:

  - None supported.
    This means the **LDM** and **STM** instructions are
    not interruptible. ARMv7-M reserved.

  - **LDM** and **STM** instructions are restartable.
  -
  - **LDM** and **STM** instructions are continuable.

https://stackoverflow.com/questions/52924118/interrupted-load-multiple-store-multiple-on-cortexm

Late Arrival (1)

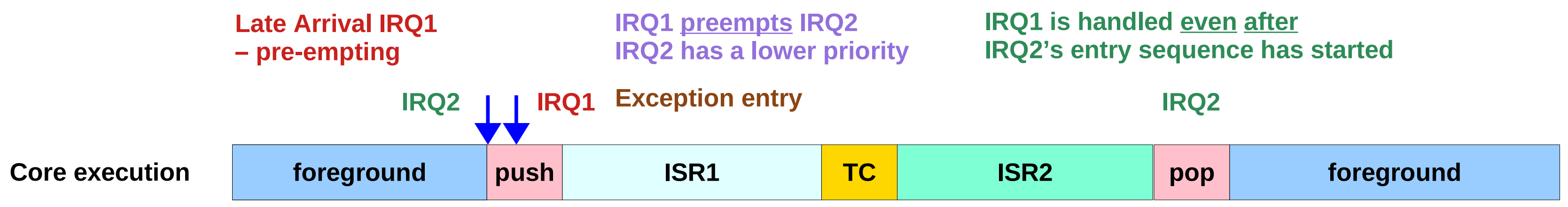
- A late-arriving interrupt is an interrupt which is recognized after the processor has started its exception entry procedure.

- If the late-arriving interrupt has higher pre-empting priority than the exception which the processor has already started to handle, then the existing stack push will continue but the vector fetch will be *re-started* using the vector for the late-arriving interrupt.

after starting an exception entry, other interrupts are requested

current ***stack operation – utilized***

current ***vector fetch – not used abandoned, restarted***



https://developer.arm.com/documentation/ka001190/latest

Late Arrival (2)

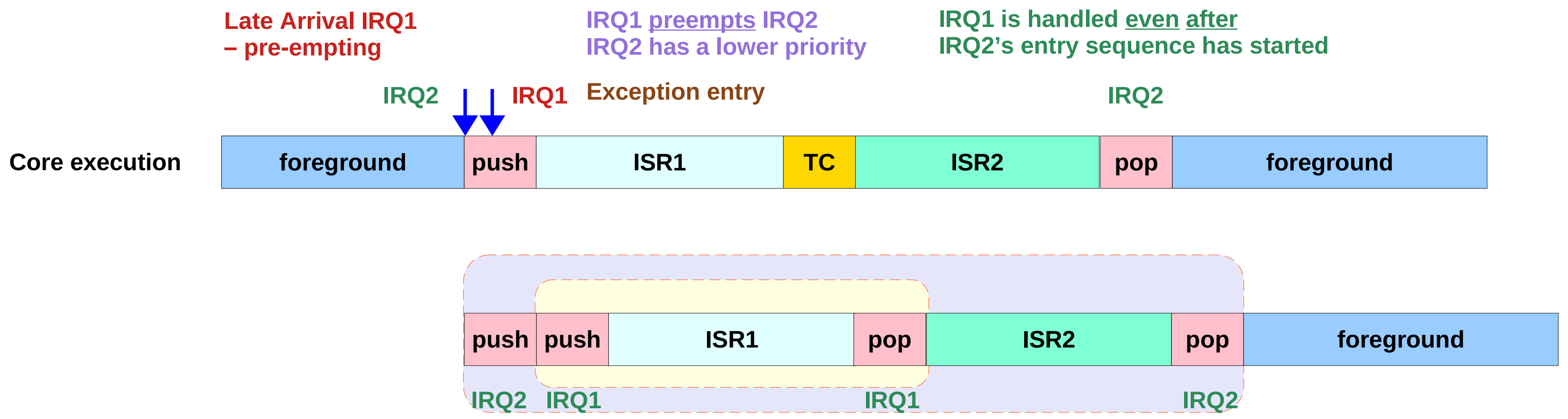
- This guarantees that the interrupt with the highest pre-empting priority will be serviced first, but in some circumstances this results in some wasted cycles from the original vector fetch which was abandoned.

after starting an exception entry, other interrupts are requested

current ***stack operation – utilized***

current ***vector fetch – not used abandoned, restarted***

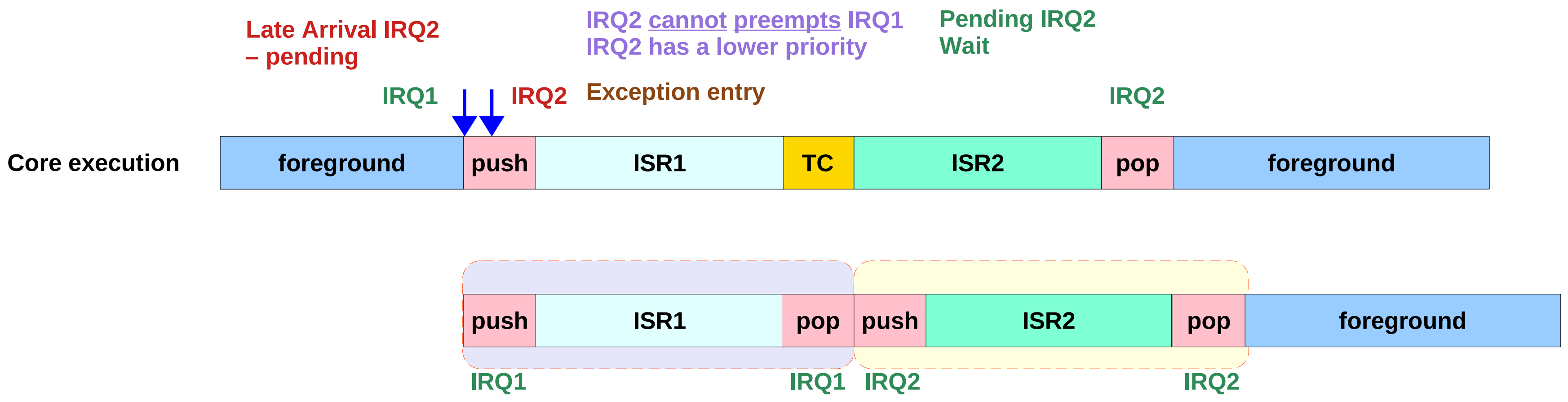
→ wasted cycles



https://developer.arm.com/documentation/ka001190/latest

Late Arrival (3)

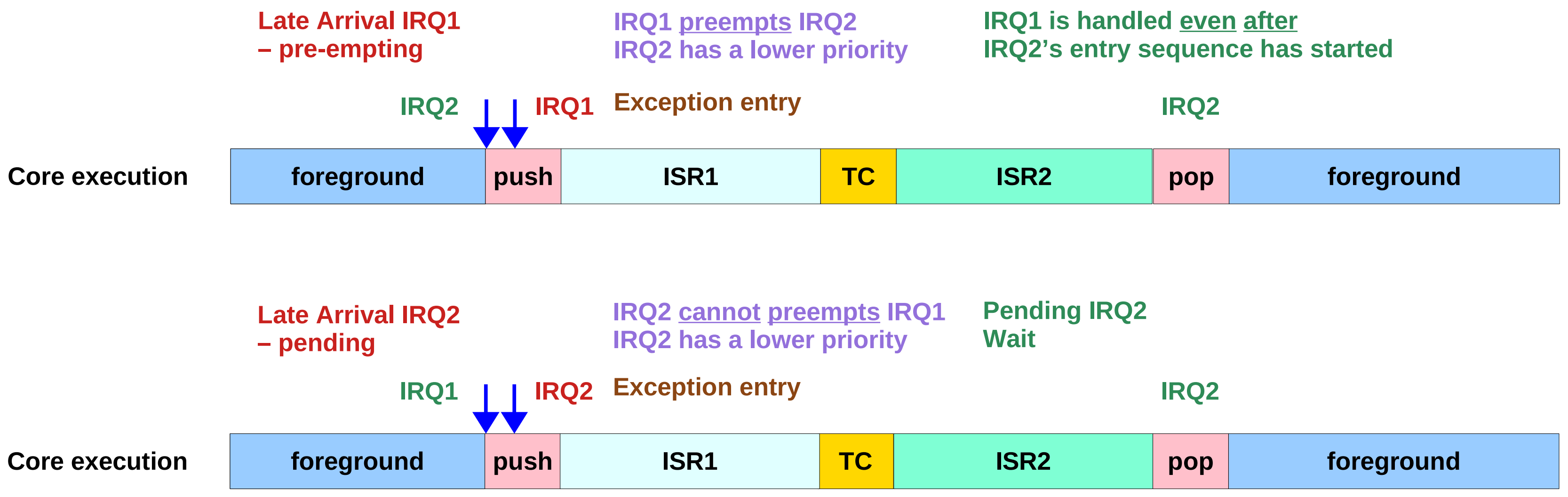
- If the late-arriving interrupt
  has only equal priority to (or lower priority than)
  the exception which the processor
  has already started to handle,
  then the late-arriving interrupt
  will remain pending until
  after the exception handler
  for the current exception has run



https://developer.arm.com/documentation/ka001190/latest

Late Arrival (4)

- This is because the late-arriving behaviour is classed as a pre-empting behaviour, and is therefore dependent only upon the pre-empting priority levels of the interrupts and exceptions.



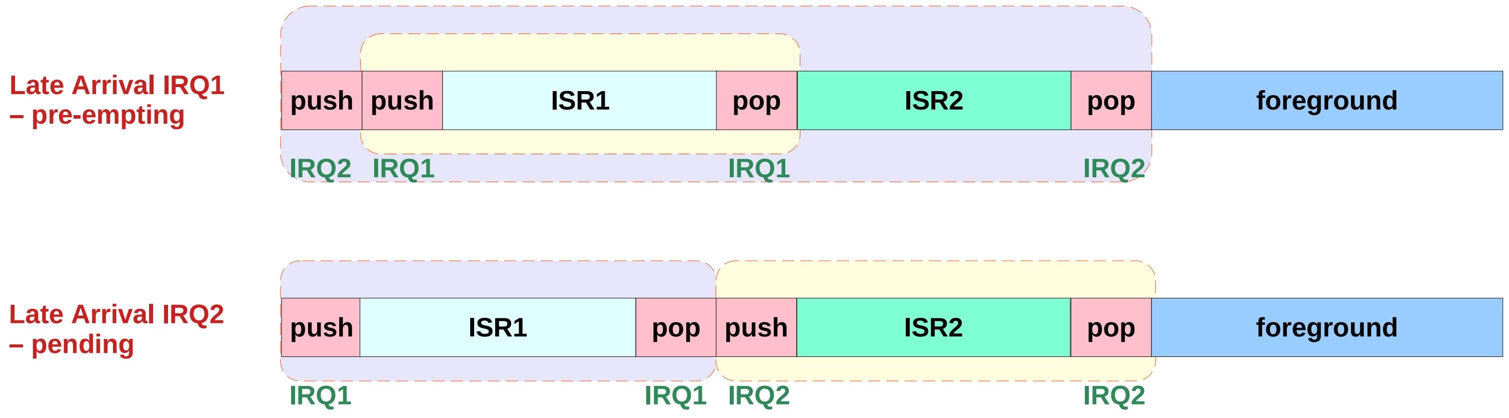
https://developer.arm.com/documentation/ka001190/latest

Late Arrival (5)

- Because the stack push has already been initiated, the interrupt latency
  (meaning the number of cycles between the arrival of the interrupt request and execution of the first instruction of its handler)

  might be less than the standard interrupt latency for the particular processor and system.

Standard Stack Operations



https://developer.arm.com/documentation/ka001190/latest

Late Arrival (6)

- Some (but not all) **Cortex-M** processors provide an *implementation-time* option for the chip designer to specify a minimum value for the interrupt latency, reducing or removing the uncertainty in interrupt latency by adding stall cycles in such cases.
- Documentation of the specific chip should provide details of this setting, if applicable.

Interrupt latency > min value

min value
: set at the implementation time

add stall cycles
to small interrupt latency
to meet the min value

https://developer.arm.com/documentation/ka001190/latest

Single copy atomicity in ARM (1)

- a **read** or **write** operation is single-copy atomic
  if the following conditions are both true:

- after *any number* of **write** operations to a memory location,
  the value of the memory location is
  the value written by one of the **write** operations.

- It is *impossible* for part of the value of the memory location
  to come from one **write** operation
  and another part of the value to come
  from a different **write** operation

https://stackoverflow.com/questions/24010989/arm-single-copy-atomicity

Single copy atomicity in ARM (2)

- When a **read** operation and a **write** operation
  are made to the same memory location,
  the value obtained by the **read** operation is one of:

  - the value of the memory location
    before the **write** operation

  - the value of the memory location
    after the **write** operation.

- It is never the case that
  the value of the **read** operation is
  partly the value of the memory location
  before the **write** operation
  and partly the value of the memory location
  after the **write** operation.

https://stackoverflow.com/questions/24010989/arm-single-copy-atomicity

Single copy atomicity in ARM (3)

- So your understanding is right - the defining point of a single-copy atomic operation is that at any given time you can only ever see either all of it, or none of it.
-
- There is a case in v7 whereby (if I'm interpreting it right) two normally single-copy atomic stores that occur to the same location at the same time but with different sizes break any guarantee of atomicity, so in theory you could observe some unexpected mix of bytes there - this looks to have been removed in v8.

https://stackoverflow.com/questions/24010989/arm-single-copy-atomicity

Interruptible LDM, STM (1)

- the load multiple (**LDM**) instructions are explicitly not atomic.

- section A3.5.3 of the ARM V7C architecture reference manual.

- **LDM**, LDC, LDC2, LDRD, **STM**, STC, STC2, STRD, PUSH, POP, RFE, SRS, VLDM, VLDR, VSTM, and VSTR instructions

  - are executed as a sequence of word-aligned word accesses.

  - *each* 32-bit word access is guaranteed to be single-copy atomic.

  - the architecture does not require subsequences of *two or more* word accesses from the sequence to be single-copy atomic.

-

https://stackoverflow.com/questions/9857760/can-an-arm-interrupt-occur-in-mid-instruction

Interruptible LDM, STM (2)

- the **LDM**/**STM** instructions
  can be aborted by an interrupt
  and restarted from the beginning on interrupt return

- **LDM** and **STM** instructions
  can *always* be interrupted by a data abort,
  so they're non atomic in that sense.

- Otherwise, the ARMv7-A architecture
  does its best to help you out.

- for interrupts, they can ***only*** be interrupted
  - if low interrupt latency is enabled,
  - AND normal memory is being accessed.

  - So at the very least, you won't get
    repeated accesses to device memory.
  - You don't want to do anything
    that expects atomic read/writes of normal memory
    though.

https://stackoverflow.com/questions/9857760/can-an-arm-interrupt-occur-in-mid-instruction

Interruptible LDM, STM (3)

- On v7-M, LDM and STM can be interrupted at any time (see section B1.5.10 of the ARMv7-M Architecture Reference Manual). It's implementation defined whether or not the instruction is restarted from the beginning of the list of loads/stores, or whether it's restarted from where it left off. As the ARM says:
-
- The ARMv7-M architecture supports continuation of, or restarting from the beginning, an abandoned LDM or STM instruction as outlined below. Where an LDM or STM is abandoned and restarted (ICI bits are not supported), the instructions should not be used with volatile memory.
-
- In other words, don't rely on LDM or STM being atomic if you're trying to write portable code.

https://stackoverflow.com/questions/9857760/can-an-arm-interrupt-occur-in-mid-instruction

Interruptible LDM, STM (4)

- If an **STM** or **LDM** instruction is interrupted,
  EPSR is set to indicate the point
  from which the execution can continue,
  and then exception entry is triggered.

- the stacked PSR value that contains this information,
  just as it contains the Thumb bit from the interrupted code.

- If your new context has
  zero in the ISI bits of the stacked PSR,
  you should not see a usage fault exception
  for the reasons you give.

https://stackoverflow.com/questions/52924118/interrupted-load-multiple-store-multiple-on-cortexm

Interruptible LDM, STM (5)

- **Application Program Status Register (APSR)**
  - The **APSR** contains the current state of the condition flags from previous instruction executions.
- **Interrupt Program Status Register (IPSR)**
  - The **IPSR** contains the exception type number of the current Interrupt Service Routine (ISR)
- **Execution Program Status Register (EPSR)**
  - The **EPSR** contains the
    - thuicoimb state bit, and
    - the execution state bits

    - for either the:
    - If-Then (IT) instruction
    - Interruptible-Continuable Instruction (ICI) field for an interrupted load multiple or store multiple instruction.

https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/programmers-model/core-registers

Interruptible LDM, STM (6)

- The ICI/IT field is part of EPSR, not IPSR,
  not that it makes a huge amount of difference
  if you're interacting with xPSR.

- If an **STM** or **LDM** instruction is interrupted,
  EPSR is
  - set to indicate the point
    from which the execution can continue, and then
  - exception entry is triggered.

- It is therefore the stacked PSR value
  that contains this information,
  just as it contains the Thumb bit
  from the interrupted code.

- If your new context has zero
  in the ISI bits of the stacked PSR,
  you should not see a usage fault exception for the reasons
  you give. (In the absence of any code, I can't really be
  more specific than this.)

https://stackoverflow.com/questions/52924118/interrupted-load-multiple-store-multiple-on-cortexm

Interruptible LDM, STM (7)

- If **LDM** and **STM** are implemented
  as restartable or continuable,
  then no, the stack will not be corrupted by this process.
  (That would be a nightmare!)

- If **LDM** and **STM** are restartable
  then the stack pointer is simply reset to the value
  it had at the start of the LDM/STM
  and the instruction is executed anew;

- if they are continuable
  then the stack pointer is not modified
  but a partial **STM**/**LDM** is performed
  to complete the instruction.

https://stackoverflow.com/questions/52924118/interrupted-load-multiple-store-multiple-on-cortexm

Interruptible LDM, STM (8)

- You don't mention exactly how you're achieving a context switch,
  but I assume you are manually pushing r4-r11 to the process stack,
  then saving the PSP somewhere
  and updating it to point to the new context on a different stack,
  before popping r4-r11 and triggering an exception return
  - that's certainly the usual way to go about it.

https://stackoverflow.com/questions/52924118/interrupted-load-multiple-store-multiple-on-cortexm

Nest VIC (1)

- In a microcontroller, such as those at the heart of industrial motion controllers, interrupts serve as a way to immediately divert the CPU from its current task to another, more important task.

- An interrupt can be triggered internally from the microcontroller (MCU) or externally, by a peripheral.

- the interrupt alerts the CPU to an occurrence such as a time-based event
  - a specified amount of time has elapsed or
  - a specific time is reached, for example,
- a change of state, or
- the start or end of a process.

https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/

Nest VIC (2-1)

- Another method of monitoring a timed event or change of state is referred to as “polling.”
- With polling, the status of a timer or state change is periodically checked.
- The downsides of polling are the risk of excessive latency (delay) between the actual change and its detection, the possibility of missing a change altogether, and the increased processing time and power it requires.
-
-
-

https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/

Nest VIC (2-2)

- When an interrupt occurs,
  an interrupt signal is generated,
  which causes the CPU to stop its current operation,
  save its current state,
  and begin the processing program
  — referred to as an interrupt service routine (ISR) or interrupt handler
  — associated with the interrupt.
-
- When the interrupt processing is complete,
- the CPU restores its previous state and resumes where it left off.

https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/

Nest VIC (3)

- Nested vector interrupt control (NVIC) is
  a method of prioritizing interrupts,
  improving the MCU’s performance and
  reducing interrupt latency.
- NVIC also provides implementation schemes
  for handling interrupts that occur
  when other interrupts are being executed or
  when the CPU is in the process of restoring its previous state
  and resuming its suspended process.
-
- The term “nested” refers to the fact that in NVIC,
  a number of interrupts can be defined
  (up to several hundred in some processors), and
  each interrupt is assigned a priority,
  with “0” being the highest priority.
- In addition, the most critical interrupt
  can be made non-maskable,
  meaning it cannot be disabled (masked).

https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/

Nest VIC (4)

- One function of NVIC is to ensure that higher priority interrupts are completed before lower-priority interrupts, even if the lower-priority interrupt is triggered first.
- For example, if a lower-priority interrupt is being registered* or executed and a higher-priority interrupt occurs, the CPU will stop the lower-priority interrupt and process the higher-priority one first.
-
- * A register is a special, dedicated memory circuit within the CPU that can be written and read much more quickly than regular memory.
-
- The register is used to store information such as calculation results, CPU execution states, or other critical program information.

https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/

Nest VIC (5)

- Similarly, a handling scheme referred to as “tail-chaining” specifies that if an interrupt is pending while the ISR for another, higher-priority another interrupt completes, the processor will immediately begin the ISR for the next interrupt, without restoring its previous state.
-
- The term “vector” in nested vector interrupt control refers to the way in which the CPU finds the program, or ISR, to be executed when an interrupt occurs.
-
-

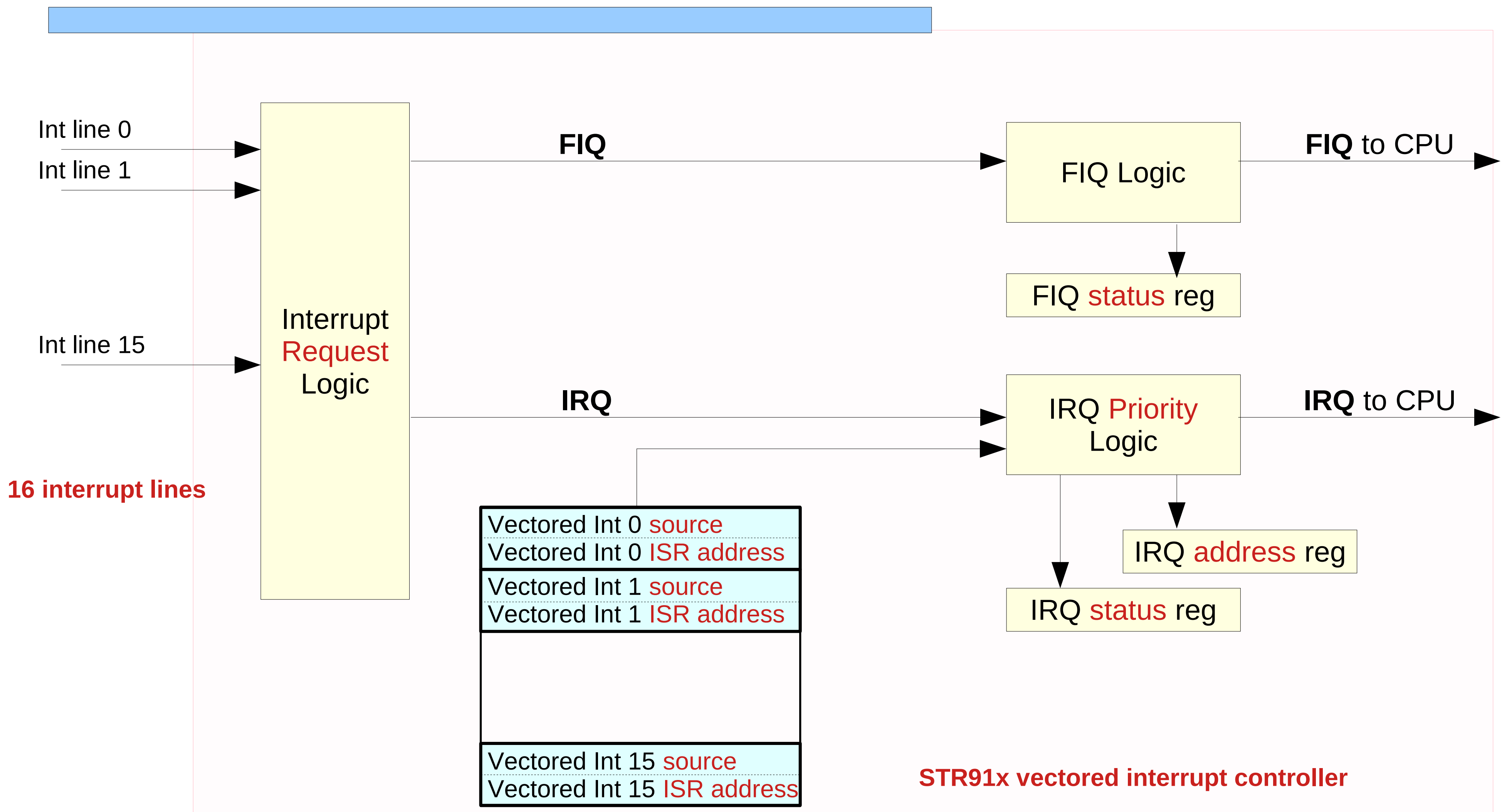
https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/

Nest VIC (6)

- Nested vector interrupt control uses a vector table
  that contains the addresses of the ISRs for each interrupt.
- When an interrupt is triggered,
  the processor gets the address from the vector table.
-
- The prioritization and handling schemes
  of nested vector interrupt control
  reduce the latency and overhead
  that interrupts typically introduce and
- ensure low power consumption,
  even with high interrupt loading on the controller.
-

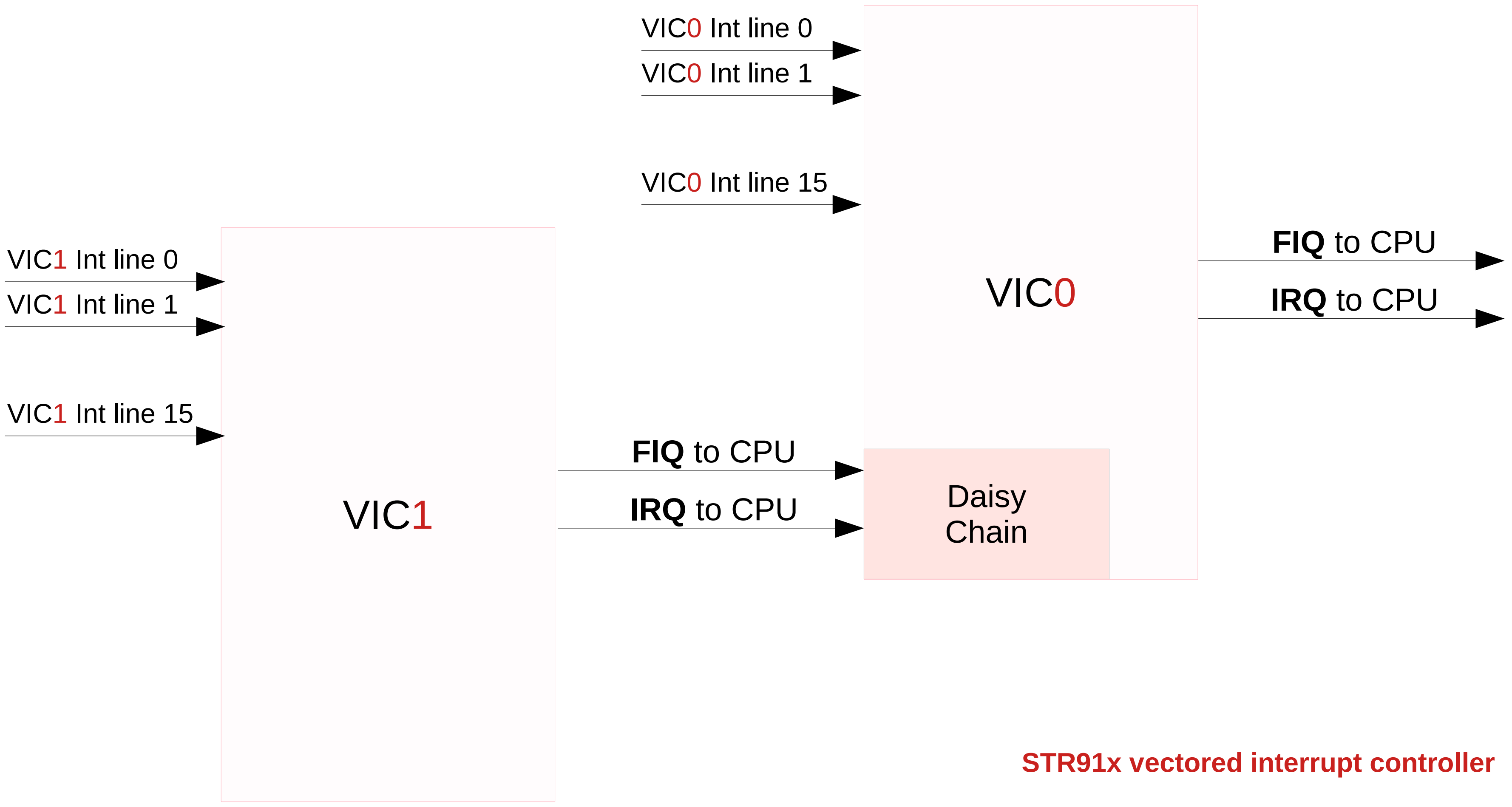
https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/

Single VIC diagram in STR91x



https://www.st.com/resource/en/application_note/an2593-str91x-interrupt-management-stmicroelectronics.pdf

Single VIC diagram in STR91x

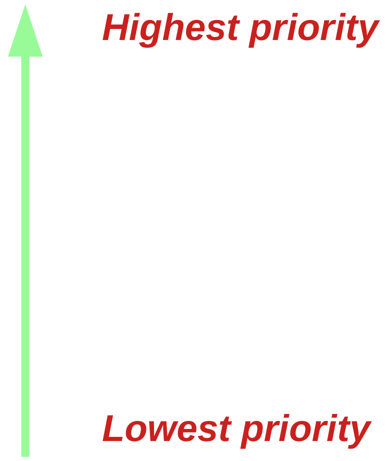


STR91x vectored interrupt controller

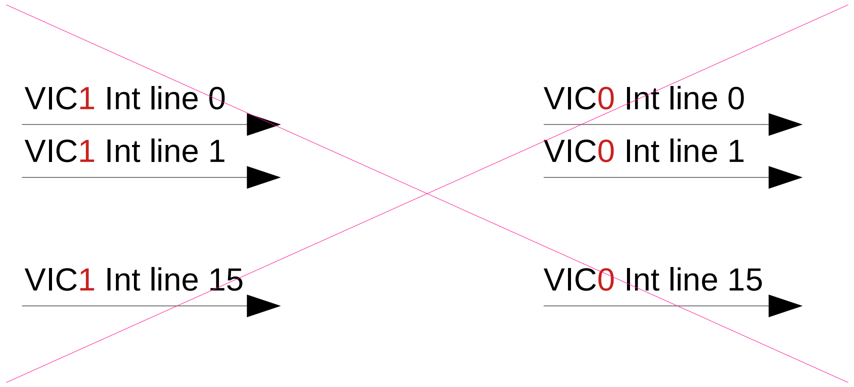
https://www.st.com/resource/en/application_note/an2593-str91x-interrupt-management-stmicroelectronics.pdf

VIC interrupt priority level in STR91x

Interrupt	Configured Priority
VIC0 FIQ / VIC1 FIQ	NA
VIC0 IRQ	0
VIC0 IRQ	1
VIC0 IRQ	15
VIC1 IRQ	0
VIC1 IRQ	1
VIC1 IRQ	15



Vectored Int 0 source
Vectored Int 0 ISR address
Vectored Int 1 source
Vectored Int 1 ISR address
Vectored Int 15 source
Vectored Int 15 ISR address



http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

FIQ interrupt management in STR91x

- for FIQ there are no priority levels.
- when an enabled FIQ interrupt occurs, the VIC signals it *directly* to the ARM core by asserting the FIQ interrupt line.
- then the ARM core switches to FIQ mode, and goes to address 0x1C where the FIQ interrupt handler resides

no priority levels

directly assert FIQ interrupt line

FIQ mode

FIQ interrupt handler at 0x1C

Interrupt	Configured Priority
VIC0 FIQ / VIC1 FIQ	NA
VIC0 IRQ	0
VIC0 IRQ	1
VIC0 IRQ	15
VIC1 IRQ	0
VIC1 IRQ	1
VIC1 IRQ	15

Highest priority

Lowest priority

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

FIQ interrupt management in STR91x

- normally in order to minimize FIQ interrupt latency
  only one interrupt should be configured as FIQ.
- But it is possible to configure several interrupts as FIQ,
  and in this case the application software
  must read the FIQ status registers of both VIC0 and VIC1
  in order to determine the FIQ interrupt source
- when the interrupt flag is cleared
  in the peripheral(s) that generated the interrupt,
  the VIC then will stop asserting the FIQ interrupt to CPU
  and the flag will be cleared in the VIC FIQ status register

one interrupt source

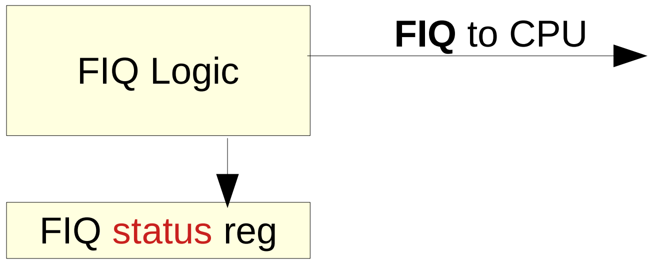
multiple interrupt sources

read the FIQ status reg

determine the FIQ source

interrupt flag in the peripherals

interrupt flag in the VIC status reg



https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

IRQ interrupt management in STR91x

1. Vectored IRQ handling
2. Simple (Non vectored) IRQ handling

Vectored handling ensures the *best* interrupt latency

the hardware priority management of the VIC
 - small latency

the software priority management of the VIC
 - simple handling

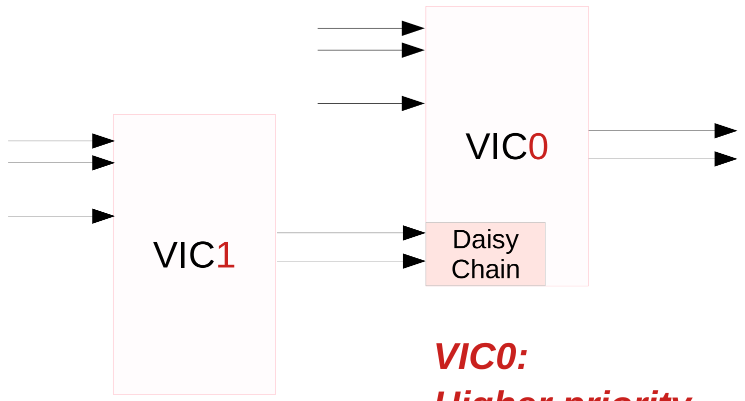
https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

IRQ interrupt management in STR91x

Although a software priority management *increases* the interrupt latency,
it can be useful in special cases
where a VIC1 interrupt has to be configured
with a higher priority level than a VIC0 interrupt,

- this is not possible when using
the hardware priority management
due to the hardwired priority between VIC0 and VIC1

Interrupt	Configured Priority
VIC0 FIQ / VIC1 FIQ	NA
VIC0 IRQ	0
VIC0 IRQ	1
VIC0 IRQ	15
VIC1 IRQ	0
VIC1 IRQ	1
VIC1 IRQ	15



VIC0:
Higher priority

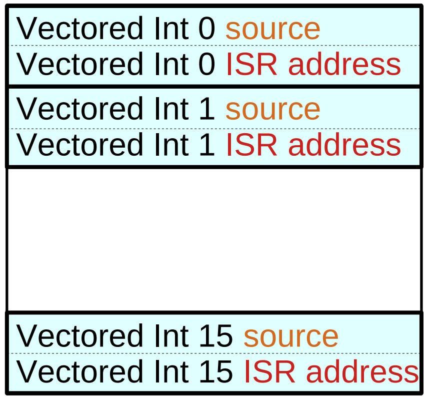
https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

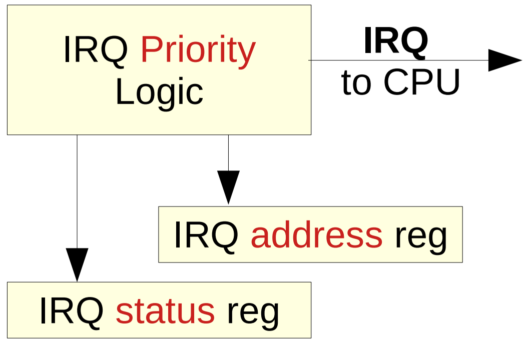
Vectored handling of IRQ in STR91x (1)

When an IRQ interrupt from VIC0 or from VIC1 occurs,

- If the interrupt has a lower priority than the current interrupt being processed, then it remains pending in the VIC until it becomes the higher priority interrupt.
- If the interrupt has the highest priority level then the VIC0 Vector Address register VIC0_VAR will be loaded with the ISR address and an IRQ interrupt will be signalled to CPU.

  Note: The VIC0_VAR will be loaded with the ISR address of the interrupt independently from the the interrupt source either from VIC0 or from VIC1.


Vectored Int 0 source
Vectored Int 0 ISR address
Vectored Int 1 source
Vectored Int 1 ISR address
Vectored Int 15 source
Vectored Int 15 ISR address



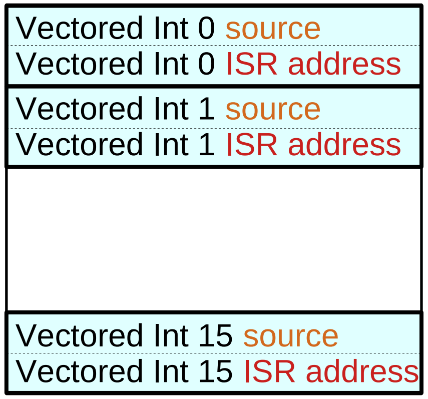
https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

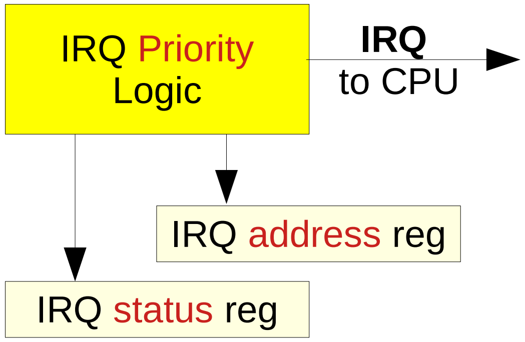
Vectored handling of IRQ in STR91x (2)

1. In the IRQ interrupt handler, the software should read the VIC0_VAR (vector address register) to determine the ISR address and <u>jump</u> to it.

2. If the interrupt originates <u>from VIC0</u>, then reading the VIC0_VAR in step 1 will <u>update</u> the priority logic of VIC0: so interrupts with the <u>same</u> or <u>lower</u> <u>priority levels</u> will be <u>*masked*</u> by the VIC.

   But if an interrupt originates <u>from VIC1</u> then you must also read the VIC1_VAR in order to <u>update</u> the <u>priority logic</u> in VIC1.


Vectored Int 0 source
Vectored Int 0 ISR address
Vectored Int 1 source
Vectored Int 1 ISR address
Vectored Int 15 source
Vectored Int 15 ISR address



https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

Vectored handling of IRQ in STR91x (3)

3. After handling the interrupt
   including the clearing of the interrupt flags,
   you must write any value in the VIC0_VAR
   if the interrupt originates from VIC0,
   or in the VIC1_VAR
   if the interrupt is from VIC1,
   in order to indicate to the VIC
   that interrupt processing has finished,
   so it can update the priority logic:
   then a same or lower level interrupt
   will be able to interrupt the CPU

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

Non-vectored handling of IRQ in STR91x (1)

Non-vector handling method of IRQ interrupts
does not use the VIC hardware priority management,

so this means you do not have to read or write
the VIC0_VAR or VIC1_VAR registers
to update the hardware priority logic.

This method can be used
when there is a need to give higher priority
to a VIC1 interrupt over a VIC0 interrupt.

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

Non-vectored handling of IRQ in STR91x (2)

The flow for simple (non vectored) IRQ handling is the following:

1. An IRQ interrupt occurs.
2. Branch to the interrupt handler.
3. Read the VICs IRQ Status registers
   to determine the source that generated the interrupt,
   and prioritize the interrupts if there are multiple active interrupt sources.
4. Branch to the *corresponding* ISR.
5. Execute the ISR.
6. Clear the interrupt. If a software interrupt generated the request,
   you must write to the VICx_SWINTCR register.
7. Check the IRQ Status registers of both VICs
   to ensure that no other interrupt is active.
   If there is an active request, go to Step 4.
8. Return from the interrupt.

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

Vectored Interrupt Controller

the sequence for the vectored interrupt flow:

- **VICVectAddr** Register
  read to branch to the ISR (interrupt service routine),
  which is currently active.
  write to clear the respective interrupt

- **VICSoftIntClear** Register
  to clear the software interrupt request triggered by VICSoftInt

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

Vectored Interrupt Controller

the sequence for the vectored interrupt flow:

- When an interrupt occurs, The ARM processor branches to either the **IRQ** or **FIQ interrupt vector**.
- If the interrupt is an **IRQ**, read the **VICVectAddr** Register and branch to the ISR (interrupt service routine).
- Stack the workspace so that you can re-enable IRQ interrupts.
- Enable the IRQ interrupts so that a higher priority can be serviced.
- Execute the Interrupt Service Routine (ISR).
- Clear the requesting interrupt in the peripheral, or write to the **VICSoftIntClear** Register if the request was generated by a software interrupt.
- Disable the interrupts and restore the workspace.
- Write to the **VICVectAddr** Register. This clears the respective interrupt in the internal interrupt priority hardware.
- Return from the interrupt. This re-enables the interrupts.

https://embetronicx.com/tutorials/microcontrollers/stm32/vectored-interrupt-controller-nested-vectored-interrupt-controller-vic-nvic/

Interrupt Vector Table with Priority

Interrupt Source		*Service Routine*		
VicVecCntl0	*IRQ source 0*	**VicVecAddr0**	*ISR 0 address*	***Highest priority***
VicVecCntl1	*IRQ source 1*	**VicVecAddr1**	*ISR 1 address*	
VicVecCntl2	*IRQ source 2*	**VicVecAddr2**	*ISR 2 address*	
VicVecCntl3	*IRQ source 3*	**VicVecAddr3**	*ISR 3 address*	
VicVecCntl4	*IRQ source 4*	**VicVecAddr4**	*ISR 4 address*	
VicVecCntl5	*IRQ source 5*	**VicVecAddr5**	*ISR 5 address*	
VicVecCntl6	*IRQ source 6*	**VicVecAddr6**	*ISR 6 address*	
VicVecCntl7	*IRQ source 7*	**VicVecAddr7**	*ISR 7 address*	
VicVecCntl8	*IRQ source 8*	**VicVecAddr8**	*ISR 8 address*	
VicVecCntl9	*IRQ source 9*	**VicVecAddr9**	*ISR 9 address*	
VicVecCntl10	*IRQ source 10*	**VicVecAddr10**	*ISR 10 address*	
VicVecCntl11	*IRQ source 11*	**VicVecAddr11**	*ISR 11 address*	
VicVecCntl12	*IRQ source 12*	**VicVecAddr12**	*ISR 12 address*	
VicVecCntl13	*IRQ source 13*	**VicVecAddr13**	*ISR 13 address*	
VicVecCntl14	*IRQ source 14*	**VicVecAddr14**	*ISR 14 address*	
VicVecCntl15	*IRQ source 15*	**VicVecAddr15**	*ISR 15 address*	***Lowest priority***

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Vectored meaning (2)

the '*magnitude*' : the interrupt source ID
the 'source' of the currently pending IRQ

VicVecCntl0~15

the '*direction*' : the corresponding ISR
vectored IRQ '*points to*' its own unique ISR

VicVecAddr0~15

Non-Vectored IRQs does not point to a unique ISR
instead, **default / common** ISR

In LPC214x, **VICDefVectAddr** register is used
The user must assign the address of the default ISR

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Vectored meaning (3)

VIC (in ARM CPUs & MCUs), as per its design,
can take 32 interrupt request inputs
but only 16 requests can be assigned
to Vectored IRQ interrupts
in its LCP2148 ARM7 Implementation.

VicVecCntl0~15 **VicVecAddr0~15**

We are given a set of 16 vectored IRQ **slots**
to which we can assign any of the 22 **requests**
that are available in LPC2148.

The slot numbering goes from 0 to 15
with slot no. 0 having highest priority and
slot no. 15 having lowest priority.

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Vectored meaning (5)

Bit 0 : WDT	Bit11 : SPI1/SSP
Bit 1 : N/A	Bit12 : PLL
Bit 2 : ARMC0	Bit13 : RTC
Bit 3 : ARMC1	Bit14 : EINT0
Bit 4 : TIMR0	Bit15 : EINT1
Bit 5 : TIMR1	Bit16 : EINT2
Bit 6 : UART0	Bit17 : EINT3
Bit 7 : UART1	Bit18 : AD0
Bit 8 : PWM	Bit19 : I2C1
Bit 9 : I2C0	Bit20 : BOD
Bit10 : I2C0	Bit21 : AD1
	Bit22 : USB

Interrupt Source Encoding

22 requests

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Vectored meaning (4)

For example if you working with 2 interrupt sources
UART0 and TIMER0.

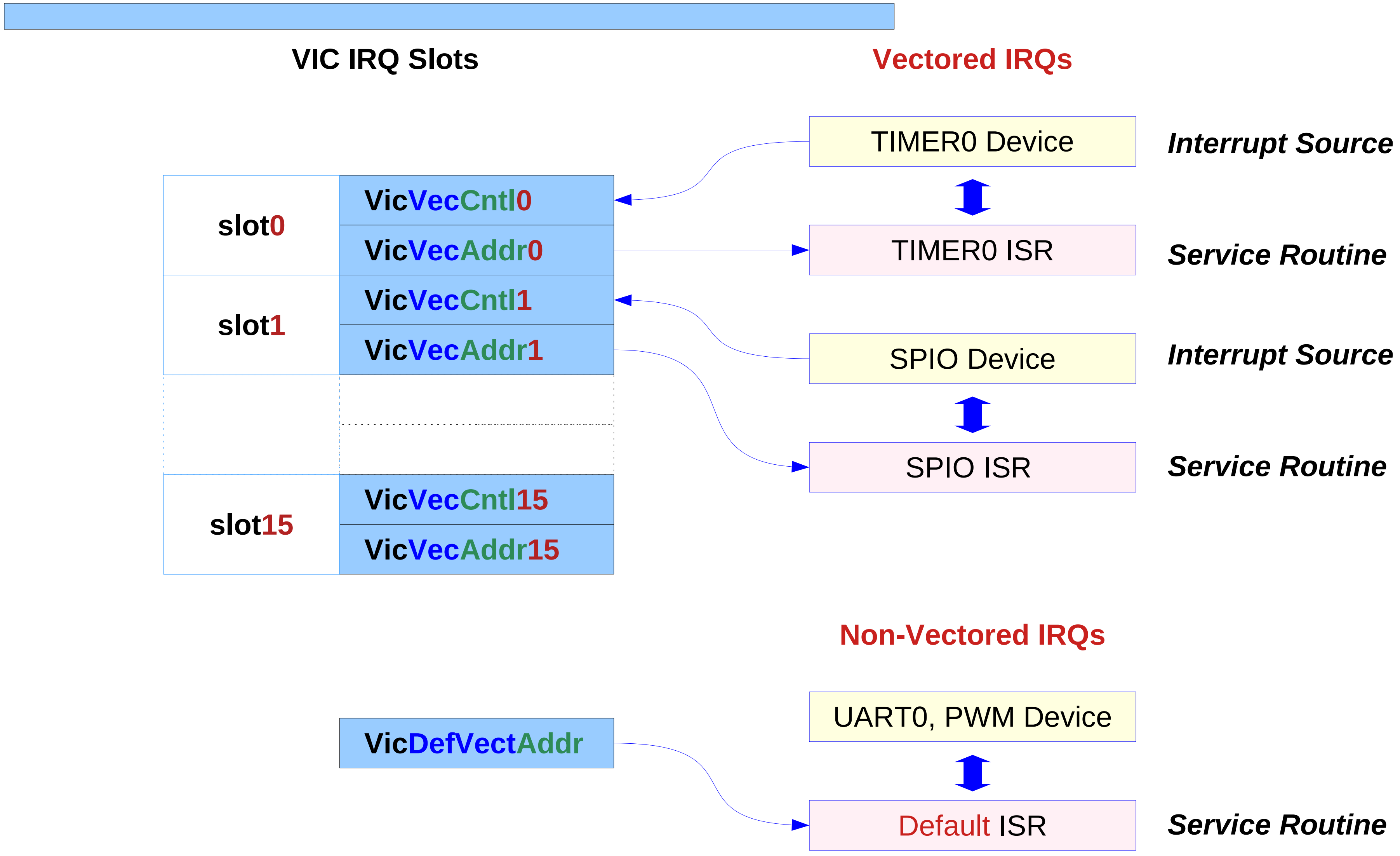
Now if you want to give TIMER0 a higher priority than UART0
then assign TIMER0 interrupt a lower number slot than UART0 .

eg. TIMER0 to slot 0 and UART0 to slot 1 or
TIMER0 to slot 4 and UART to slot 9 and so on.

The number of the slot doesn’t matter
as long TIMER0 slot is lower than UART0 slot.

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Defining the ISR for Timers



http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Vectored meaning (5)

VIC has plenty of registers.

Most of the registers that are used
to configure interrupts or read status

each bit corresponds to a particular interrupt source
and this correspondence is same for all of these registers.

For example
  bit 0 in these registers corresponds to Watch dog timer interrupt,
  bit 4 corresponds to TIMER0 interrupt ,
  bit 6 corresponds to UART0 interrupt .. and so on.

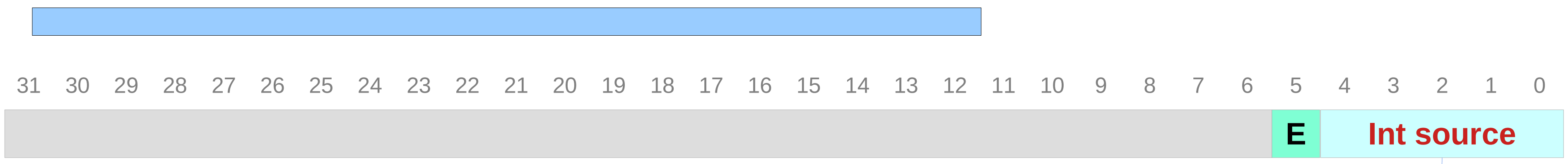
http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Vectored meaning (5)

1) **VICIntSelect** (R/W) : used to select an interrupt as IRQ or as FIQ
2) **VICIntEnable** (R/W): used to enable interrupts
3) **VICIntEnClr** (R/W) : used to disable interrupts
4) **VICIRQStatus** (R) : used for reading the current status of the enabled IRQ interrupts.
5) **VICFIQStatus** (R) : used for reading the current status of the enabled FIQ interrupts
6) **VICSoftInt** : used to generate interrupts using software i.e the program itself
7) **VICSoftIntClear** : used to clear the interrupt request that was triggered(forced) using VICSoftInt.
8) **VICVectCntl0 ~15** : used to assign a particular interrupt source to a particular slot.
9) **VICVectAddr0 ~15** : store the address of the function that must be called when an interrupt occurs
10) **VICVectAddr** : holds the address of the associated ISR i.e the one which is currently active.
11) **VICDefVectAddr** : stores the address of the "default/common" ISR for a **Non-Vectored IRQ** occurs

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

VICVectCntl Registers



VICVectCntl0 ~ 15 : used to assign a particular interrupt source to a particular slot.

VICVectCntl0 - the highest priority VICVectCntl15 - the lowest priority

Bit4 ~ Bit0 contain the number of the interrupt request which is assigned to this slot.

Bit5 is used to enable the vectored IRQ slot by writing a 1

WDT	: 0	SPI1/SSP	: 11
N/A	: 1	PLL	: 12
ARMC0	: 2	RTC	: 13
ARMC1	: 3	EINT0	: 14
TIMR0	: 4	EINT1	: 15
TIMR1	: 5	EINT2	: 16
UART0	: 6	EINT3	: 17
UART1	: 7	AD0	: 18
PWM	: 8	I2C1	: 19
I2C0	: 9	BOD	: 20
I2C0	: 10	AD1	: 21
		USB	: 22

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Defining the ISR for Timers

defining the ISR

explicitly tell the compiler that the function
is not a normal function but an ISR

a special keyword called "__irq"
: a function qualifier.

use this keyword with the function definition

an example of defining an ISR in Keil :

```
__irq void myISR (void)
{
 ...
}

// or equivalently

void myISR (void) __irq
{
 ...
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Setup the interrupt for Timers

for ARM based microcontrollers like lpc2148.

in order to assign TIMER0 IRQ and ISR to slot X.

Assign TIMER0 Interrupt to Slot number 0

```
// Enable TIMER0 IRQ
// 5th bit must 1 to enable the slot
// Vectored-IRQ for TIMER0 has been configured

VICIntEnable    |= (1<<4) ;
VICVectCntl0    = (1<<5) | 4 ;
VICVectAddr0 = (unsigned) myISR;
```

Bit 0	: WDT
Bit 1	: N/A
Bit 2	: ARMC0
Bit 3	: ARMC1
Bit 4	: TIMR0
Bit 5	: TIMR1
Bit 6	: UART0
Bit 7	: UART1
Bit 8	: PWM
Bit 9	: I2C0
Bit10	: I2C0

2) VICIntEnable (R/W) : used to enable interrupts
8) VICVectCntl0 ~15 : used to assign a particular interrupt source to a particular slot.
9) VICVectAddr0 ~15 : store the address of the function that must be called when an interrupt occurs

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

T0IR, U0IIR

IR (Interrupt Register)
The **IR** can be written to clear interrupts.
The **IR** can be read to identify
which of eight possible interrupt sources are pending

T0IR (TIMER0 Interrupt Register)
4 bits for the *timer* match interrupts
4 bits for the *timer* capture interrupts

The high bit in the IR signifies that
an interrupt is generated

Writing a logic one
to the corresponding IR bit
will reset the interrupt.
Writing a zero has no effect.

U0IIR (UART0 Interrupt Identification Register)
The U0IIR provides a status code that denotes
the priority and source of a pending interrupt.

The interrupts are frozen during an U0IIR access.

If an interrupt occurs during an U0IIR access,
the interrupt is recorded for the next U0IIR access

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

T0IR (T0 Timer Interrupt Register)

The **IR** can be read to identify which of
8 possible interrupt sources are pending.

The **IR** can be written to clear interrupts.

TIMER/ COUNTER0	**T0IR**
TIMER/ COUNTER1	**T1IR**

The Interrupt Register consists of
four bits for the match interrupts and
four bits for the capture interrupts.

If an interrupt is generated
then the corresponding bit in the **IR** will be high.
Otherwise, the bit will be low.

Writing a logic one to the corresponding IR bit
will reset the interrupt.
Writing a zero has no effect

Bit 0	: MR0 Interrupt	flag for match channel 0
Bit 1	: MR1 Interrupt	flag for match channel 1
Bit 2	: MR2 Interrupt	flag for match channel 2
Bit 3	: MR3 Interrupt	flag for match channel 3
Bit 4	: CR0 Interrupt	flag for capture channel 0 event
Bit 5	: CR1 Interrupt	flag for capture channel 1 event
Bit 6	: CR2 Interrupt	flag for capture channel 2 event
Bit 7	: CR3 Interrupt	flag for capture channel 3 event

A high bit signifies the interrupt is generated

```
#define MR0I_FLAG (1<<0)          // 0x00000001
#define MR1I_FLAG (1<<1)          // 0x00000002
#define MR2I_FLAG (1<<2)          // 0x00000004
        ***
        regVal = T0IR;

        if( T0IR & MR0I_FLAG )        {
                * * * MR0 match * * *
        }  else if ( T0IR & MR1I_FLAG ) {
                * * * MR1 match * * *
        }  else if ( T0IR & MR2I_FLAG ) {
                * * * MR2 match * * *
        }

        T0IR = regval;
        ***
```

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

U0IIR (UART0 Interrupt Identification Register)

Note than UART0’s Interrupt Register (**U0IIR**)
is a lot different than TIMER0’s (**T0IR**).

The first Bit UART0[0] in **U0IIR** indicates
whether any interrupt is pending or not
and its Active LOW!

The next 3 bits UART0[3:1] give the Identification
for any of the 4 Interrupts if enabled.

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

U0IIR (UART0 Interrupt Identification Register)

Bit0	**Interrupt Pending** the U0IIR[0] is active low the pending interrupt can be determined by evaluating U0IIR[3:1]
Bit3:1	**Interrupt Identification** U0IIR[3:1] identifies an interrupt corresponding to the UART0 Rx FIFO All other combinations of U0IIR[3:1] not list are reserved (000, 100, 101, 111)

011	1	RLS (Receive Line Status)
010	2a	RDA (Receive Data Available)
110	2b	CTI (Character Time-Out Indicator
001	3	THRE (Transmitter Holding Register Empty)

```
__irq void myDefault_ISR(void)
{
        U0RegVal = U0IIR;     // read the current value
        …
        if( ! (U0RegVal & 0x1) )          // active low
        {
        }
        …
        VICVectAddr = 0x0;    // The ISR has finished!
}
```

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

GPIO Register (1) legacy GPIO

IOPIN GPIO Port Pin value register.
the current state of the GPIO configured port pins can
always be read from this register, regardless of pin direction

IODIR GPIO Port Direction control register.
This register individually controls the direction of
each port pin.

IOSET GPIO Port Output Set register.
This register controls the state of output pins in
conjunction with the **IOCLR** register.
Writing ones produces highs at the corresponding
port pins. Writing zeroes has no effect.

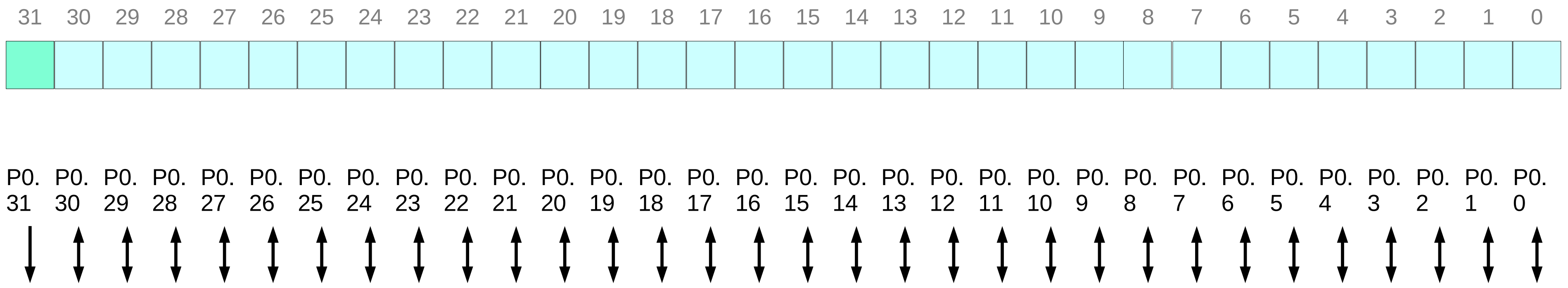
IOCLR GPIO Port Output Clear register.
This register controls the state of output pins.
Writing ones produces lows at the corresponding port pins and clears the corresponding bits in the **IOSET** register.
Writing zeroes has no effect.

the legacy GPIO referred as "the slow" GPIO
the enhanced GPIO referred as "the fast" GPIO.

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

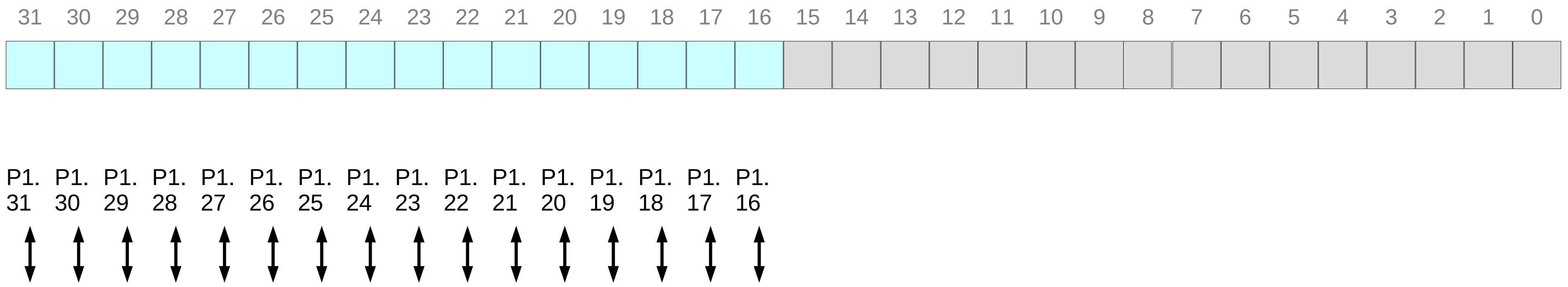
GPIO Register (2) **IODIR**

IODIR0 for Port 0 **1 for output / 0 for input**



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
P0.31 P0.30 P0.29 P0.28 P0.27 P0.26 P0.25 P0.24 P0.23 P0.22 P0.21 P0.20 P0.19 P0.18 P0.17 P0.16 P0.15 P0.14 P0.13 P0.12 P0.11 P0.10 P0.9 P0.8 P0.7 P0.6 P0.5 P0.4 P0.3 P0.2 P0.1 P0.0

IODIR1 for Port 1 **1 for output / 0 for input**



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
P1.31 P1.30 P1.29 P1.28 P1.27 P1.26 P1.25 P1.24 P1.23 P1.22 P1.21 P1.20 P1.19 P1.18 P1.17 P1.16

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

GPIO Register (3) enhanced GPIO

FIODIR Fast GPIO Port Direction control register.
This register individually controls the
direction of each port pin.

FIOMASK Fast Mask register for port.
Writes, sets, clears, and reads to port
alter or return only the bits enabled
by zeros in this register.
(done via writes to **FIOPIN**, **FIOSET**, and **FIOCLR**,
and reads of **FIOPIN**)

FIOPIN Fast Port Pin value register using **FIOMASK**.
The current state of digital port pins can be
read from this register, regardless of pin direction
or alternate function selection
(as long as pins is not configured as an input to ADC).
The value read is masked by ANDing with **FIOMASK**.
Writing to this register places corresponding values
in all bits enabled by ones in **FIOMASK**.

FIOSET Fast Port Output Set register using **FIOMASK**.
This register controls the state of output pins.
Writing 1s produces highs at the corresponding port pins.
Writing 0s has no effect.
Reading this register returns the current contents
of the port output register.
Only bits enabled by ones in **FIOMASK** can be altered

FIOCLR Fast Port Output Clear register using **FIOMASK**.
This register controls the state of output pins.
Writing 1s produces lows at the corresponding port pins.
Writing 0s has no effect.
Only bits enabled by ones in **FIOMASK** can be altered.

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

GPIO Register (4) examples

```
IO0DIR = 0xFFFFFFFF;    // Configure all pins on Port 0 as Output
IO0PIN = 0x0;


IO0PIN = ~IO0PIN;       // Toggle all pins in Port 0


IO0PIN ^= (1<<0);       // xor 2^0    Toggle GPIO0 PIN0 .. P0.0
IO0PIN ^= (1<<1);       // xor 2^1    Toggle GPIO0 PIN1 .. P0.1
IO0PIN ^= (1<<2);       // xor 2^2    Toggle GPIO0 PIN2 .. P0.2


IO0PIN ^= (1<<2);       // Toggle 3rd Pin (PIN2) in GPIO0 .. P0.2
IO0PIN ^= (1<<3);       // Toggle 4th Pin (PIN3) in GPIO0 .. P0.3
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

VICVectAddr

This must not be confused with
the set of 16 VICVecAddr0 ~15 registers.

When an interrupt is Triggered this register holds
the address of the associated ISR i.e
the one which is currently active.

Writing a value i.e dummy write to this register
indicates to the VIC that current Interrupt has finished execution.

In this tutorial the only place we'll use this register ..
is at the end of the ISR to signal end of ISR execution.

```
__irq void myDefault_ISR(void)
{
          U0RegVal = U0IIR;      // read the current value
          …
          if( ! (U0RegVal & 0x1) )          // active low
          {
          }
          …
          VICVectAddr = 0x0;     // The ISR has finished!
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 1 & 2 overview

consider two simple cases for coding an ISR

Use **TIMER0** for generating IRQs

Case #1)

only one 'internal' source of interrupt in **TIMER0**
i.e an MR0 match event which raises an IRQ.

Case #2)

multiple 'internal' source of interrupt in **TIMER0**
i.e. say a match event for MR0 , MR1 & MR2
which raise an IRQ.

T0IR for TIMER0
T0's Interrupt Register

```
regVal = T0IR;
        * * * MR0  * * *
T0IR = regval;
```

```
regVal = T0IR;

if( T0IR & MR0I_FLAG )   {
        * * * MR0 match * * *
}  else if ( T0IR & MR1I_FLAG ) {
        * * * MR1 match * * *
}  else if ( T0IR & MR2I_FLAG ) {
        * * * MR2 match * * *
}

T0IR = regval;
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Only one interrupt source

Since only one source is triggering an interrupt
we don't need to identify it
– though its a good practice to explicitly identify it.

```
__irq void myISR(void)
{
        long int regVal;

        // read the current value in
        // TIMER0's Interrupt Register
        regVal = T0IR;

        //... MR0 match event has occured
        // .. do something here

        // write back to clear the interrupt flag
        T0IR = regval;

        VICVectAddr = 0x0; // The ISR has finished!
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 2: Multiple interrupt sources

Even in case #2 things are simple unless we need to identify the 'actual' source of interrupt.

```
#define MR0I_FLAG (1<<0)
#define MR1I_FLAG (1<<1)
#define MR2I_FLAG (1<<2)

__irq void myISR(void)
{
        long int regVal;

        // read the current value
        // in TIMER0's Interrupt Register
        regVal = T0IR;



        // write back to clear the interrupt flag
        T0IR = regVal;

        // Acknowledge that ISR has finished execution
        VICVectAddr = 0x0;
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

```
if( T0IR & MR0I_FLAG )      {
        //do something for MR0 match
}  else if ( T0IR & MR1I_FLAG ) {
        //do something for MR1 match
}  else if ( T0IR & MR2I_FLAG ) {
        //do something for MR2 match
}
```

Only one interrupt source

Case #2 actually provides a general method of
using Timers as PWM generators!

You can use any one of the match registers as PWM Cycle generator
and then use other 3 match registers to generate 3 PWM signals!

Since LPC214x already has PWM generator blocks on chip
I don’t see any use of Timers being used as PWM generators.

But for MCUs which don’t have PWM generator blocks this is very useful.

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 3 & 4 overview

Both of them deal with IRQs from different blocks
: TIMER0 and UART0.

Case #3)
Multiple Vectored IRQs from different devices.
Hence Priority comes into picture here.

myTimer0_ISR()
myUart0_ISR

Case #4)
Multiple Non-Vectored IRQs from different devices.

myDefault_ISR

T0IR for TIMER0
T0's Interrupt Register

U0IIR for UART0
U0's Interrupt ID Register

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Multiple Vectored IRQ from different devices

Case #3
TIMER0 and UART0 generating interrupts with TIMER0 having higher priority.

2 different Vectored ISRs
– one for TIMER0 and one for UART0.
- **myTimer0_ISR()**
- **myUart0_ISR**

assume only 1 internal source inside both TIMER0 and UART0

```
__irq void myTimer0_ISR(void)
{
        long int regVal;

        regVal = T0IR;          // read the current value

        T0IR = regval;          // write back to clear
                                // the interrupt flag
        VICVectAddr = 0x0;

}
```

```
__irq void myUart0_ISR(void)
{
        long int regVal;

        regVal = U0IIR;         // read the current value

        //Something inside UART0 has raised an IRQ

        VICVectAddr = 0x0;
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Multiple Non-Vectored IRQ from different devices

Case #4
TIMER0 and UART0 generating interrupts

But here both of them are Non-Vectored
and hence will be serviced
by a common Non-Vectored ISR.

Hence, here we will need to check
the actual source i.e device
which triggered the interrupt and
proceed accordingly.

This is quite similar to Case #2.

T0's Interrupt Register
U0's(Uart 0) Interrupt Identification Register

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

```
__irq void myDefault_ISR(void)
{
        long int T0RegVal , U0RegVal;

        T0RegVal = T0IR;        // read the current value
        U0RegVal = U0IIR;       // read the current value

        if( T0RegVal )
        {
                //do something for TIMER0 Interrupt

                T0IR = T0RegVal;        // write back to clear
                                        // the interrupt flag
        }

        if( ! (U0RegVal & 0x1) )        // active low
        {
                // do something for UART0 Interrupt
                // No need to write back to U0IIR
                // since reading it clears it
        }

        VICVectAddr = 0x0;      // The ISR has finished!
}
```

Fast Interrupt Request

Well , you can think FIQ as a promoted version of a Vectored IRQ.

To promote or covert a Vectored IRQ to FIQ
just make the bit for corresponding IRQ
in VICIntSelect register to 1
and it will be become an FIQ.

Also Note that its recommended
that you only have one FIQ in your system.

FIQs have low latency than VIRQs
and usually used in System Critical Interrupt Handling.

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Code snippets of lpc214x.h for Keil tools (1)

```
/*************************************************************************/
/*  This file is part of the uVision/ARM development tools        */
/*  Copyright KEIL ELEKTRONIK GmbH 2002-2005                     */
/*************************************************************************/
/*                                                                */
/*  LPC214X.H:  Header file for Philips LPC2141/42/44/46/48  */
/*                                                                */
/*************************************************************************/

/* Vectored Interrupt Controller (VIC) */
#define VICIRQStatus      (*((volatile unsigned long *) 0xFFFFF000))
#define VICFIQStatus      (*((volatile unsigned long *) 0xFFFFF004))
#define VICRawIntr        (*((volatile unsigned long *) 0xFFFFF008))
#define VICIntSelect      (*((volatile unsigned long *) 0xFFFFF00C))
#define VICIntEnable      (*((volatile unsigned long *) 0xFFFFF010))
#define VICIntEnClr       (*((volatile unsigned long *) 0xFFFFF014))
#define VICSoftInt        (*((volatile unsigned long *) 0xFFFFF018))
#define VICSoftIntClr     (*((volatile unsigned long *) 0xFFFFF01C))
#define VICProtection     (*((volatile unsigned long *) 0xFFFFF020))
#define VICVectAddr       (*((volatile unsigned long *) 0xFFFFF030))
#define VICDefVectAddr    (*((volatile unsigned long *) 0xFFFFF034))
```

https://www.keil.com/dd/docs/arm/philips/lpc214x.h

Code snippets of lpc214x.h for Keil tools (2)

```
#define VICVectAddr0     (*((volatile unsigned long *) 0xFFFFF100))
#define VICVectAddr1     (*((volatile unsigned long *) 0xFFFFF104))
         …       …       …
#define VICVectAddr15    (*((volatile unsigned long *) 0xFFFFF13C))

#define VICVectCntl0     (*((volatile unsigned long *) 0xFFFFF200))
#define VICVectCntl1     (*((volatile unsigned long *) 0xFFFFF204))
         …       …       …
#define VICVectCntl15    (*((volatile unsigned long *) 0xFFFFF23C))
```

https://www.keil.com/dd/docs/arm/philips/lpc214x.h

Code snippets of lpc214x.h for Keil tools (3)

```
/* Timer 0 */
#define T0IR      (*((volatile unsigned long *) 0xE0004000))   // Interrupt Register (IR)
#define T0TCR     (*((volatile unsigned long *) 0xE0004004))   // Timer Control Register (TCR)
#define T0TC      (*((volatile unsigned long *) 0xE0004008))   // Timer Counter (TC)
#define T0PR      (*((volatile unsigned long *) 0xE000400C))   // Prescale Register (PR)
#define T0PC      (*((volatile unsigned long *) 0xE0004010))   // Prescale Counter Register (PC)
#define T0MCR     (*((volatile unsigned long *) 0xE0004014))   // Match Control Register (MCR)
#define T0MR0     (*((volatile unsigned long *) 0xE0004018))   // Match Register 0 (MR0)
#define T0MR1     (*((volatile unsigned long *) 0xE000401C))   // Match Register 1 (MR1)
#define T0MR2     (*((volatile unsigned long *) 0xE0004020))   // Match Register 2 (MR2)
#define T0MR3     (*((volatile unsigned long *) 0xE0004024))   // Match Register 3 (MR3)
#define T0CCR     (*((volatile unsigned long *) 0xE0004028))   // Capture Control Register (CCR)
#define T0CR0     (*((volatile unsigned long *) 0xE000402C))   // Capture Register 1 (CR1)
#define T0CR1     (*((volatile unsigned long *) 0xE0004030))   // Capture Register 2 (CR2)
#define T0CR2     (*((volatile unsigned long *) 0xE0004034))   // Capture Register 3 (CR3)
#define T0CR3     (*((volatile unsigned long *) 0xE0004038))   // Capture Register 4 (CR4)
#define T0EMR     (*((volatile unsigned long *) 0xE000403C))   // External Match Register (EMR)
#define T0CTCR    (*((volatile unsigned long *) 0xE0004070))   // Counter Control Register (CTCR)
```

https://www.keil.com/dd/docs/arm/philips/lpc214x.h

Code snippets of lpc214x.h for Keil tools (4)

```
/* Universal Asynchronous Receiver Transmitter 0 (UART0) */
#define U0RBR     (*((volatile unsigned char *) 0xE000C000))     // Receiver Buffer Register (RBR)
#define U0THR     (*((volatile unsigned char *) 0xE000C000))     // Transmit Holding Register (THR)
#define U0IER     (*((volatile unsigned long *) 0xE000C004))     // Interrupt Enable Register (IER)
#define U0IIR     (*((volatile unsigned long *) 0xE000C008))     // Interrupt Identification Register (IIR)
#define U0FCR     (*((volatile unsigned char *) 0xE000C008))     // FIFO Control Register (FCR)
#define U0LCR     (*((volatile unsigned char *) 0xE000C00C))     // Line Control Register (LCR)
#define U0MCR     (*((volatile unsigned char *) 0xE000C010))     // Modem Control Register (U1MCR)
#define U0LSR     (*((volatile unsigned char *) 0xE000C014))     // Line Status Register (LSR)
#define U0MSR     (*((volatile unsigned char *) 0xE000C018))     // Modem Status Register (U1MSR)
#define U0SCR     (*((volatile unsigned char *) 0xE000C01C))     // Scratch Pad Register (SCR)
#define U0DLL     (*((volatile unsigned char *) 0xE000C000))     // Divisor Latch LSB Register (DLL)
#define U0DLM     (*((volatile unsigned char *) 0xE000C004))     // Divisor Latch MSB Register (DLM)
#define U0ACR     (*((volatile unsigned long *) 0xE000C020))     // Auto-baud Control Register (ACR)
#define U0FDR     (*((volatile unsigned long *) 0xE000C028))     // Fraction Divisor Register (FDR)
#define U0TER     (*((volatile unsigned char *) 0xE000C030))     // Transmit Enable Register (TER)
```

https://www.keil.com/dd/docs/arm/philips/lpc214x.h

Code snippets of lpc214x.h for Keil tools (4)

```
/* General Purpose Input/Output (GPIO) */
#define IOPIN0        (*((volatile unsigned long *) 0xE0028000))
#define IOSET0        (*((volatile unsigned long *) 0xE0028004))
#define IODIR0        (*((volatile unsigned long *) 0xE0028008))
#define IOCLR0        (*((volatile unsigned long *) 0xE002800C))
#define IOPIN1        (*((volatile unsigned long *) 0xE0028010))
#define IOSET1        (*((volatile unsigned long *) 0xE0028014))
#define IODIR1        (*((volatile unsigned long *) 0xE0028018))
#define IOCLR1        (*((volatile unsigned long *) 0xE002801C))
#define IO0PIN        (*((volatile unsigned long *) 0xE0028000))   // alias
#define IO0SET        (*((volatile unsigned long *) 0xE0028004))   // alias
#define IO0DIR        (*((volatile unsigned long *) 0xE0028008))   // alias
#define IO0CLR        (*((volatile unsigned long *) 0xE002800C))   // alias
#define IO1PIN        (*((volatile unsigned long *) 0xE0028010))   // alias
#define IO1SET        (*((volatile unsigned long *) 0xE0028014))   // alias
#define IO1DIR        (*((volatile unsigned long *) 0xE0028018))   // alias
#define IO1CLR        (*((volatile unsigned long *) 0xE002801C))   // alias
#define FIO0DIR       (*((volatile unsigned long *) 0x3FFFC000))
#define FIO0MASK      (*((volatile unsigned long *) 0x3FFFC010))
#define FIO0PIN       (*((volatile unsigned long *) 0x3FFFC014))
#define FIO0SET       (*((volatile unsigned long *) 0x3FFFC018))
#define FIO0CLR       (*((volatile unsigned long *) 0x3FFFC01C))
#define FIO1DIR       (*((volatile unsigned long *) 0x3FFFC020))
#define FIO1MASK      (*((volatile unsigned long *) 0x3FFFC030))
#define FIO1PIN       (*((volatile unsigned long *) 0x3FFFC034))
#define FIO1SET       (*((volatile unsigned long *) 0x3FFFC038))
#define FIO1CLR       (*((volatile unsigned long *) 0x3FFFC03C))
```

https://www.keil.com/dd/docs/arm/philips/lpc214x.h

Case 1 Example code (1)

```
/*
(C) Umang Gajera | Power_user_EX - www.ocfreaks.com 2011-13.
More Embedded tutorials @ www.ocfreaks.com/cat/embedded

Soruce for Interrupt Tutorial Case #1.
License : GPL.
*/

#include <lpc214x.h>

#define MR0I            (1<<0)      // Interrupt When TC matches MR0
#define MR0R            (1<<1)      // Reset TC when TC matches MR0

#define DELAY_MS        500         // 0.5 Second(s) Delay
#define PRESCALE        60000       // 60000 PCLK clock cycles to increment TC by 1

void initClocks(void);
void initTimer0(void);
__irq void T0ISR(void);
void initClocks(void);
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 1 Example code (2)

```
int main(void)
{
	initClocks(); //Initialize CPU and Peripheral Clocks @ 60Mhz
	initTimer0(); //Initialize Timer0
	IO0DIR = 0xFFFFFFFF; //Configure all pins on Port 0 as Output
	IO0PIN = 0x0;

	T0TCR = 0x01; //Enable timer

	while(1); //Infinite Idle Loop

	//return 0; //normally this wont execute ever	:P
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 1 Example code (3)

```
void initTimer0(void)
{
        /*Assuming that PLL0 has been setup with CCLK = 60Mhz and PCLK also = 60Mhz.*/

        //----------Configure Timer0-------------
        T0CTCR = 0x0;

        T0PR = PRESCALE-1; //(Value in Decimal!) - Increment T0TC at every 60000 clock cycles
                //Count begins from zero hence subtracting 1
                //60000 clock cycles @60Mhz = 1 mS

        T0MR0 = DELAY_MS-1; //(Value in Decimal!) Zero Indexed Count - hence subtracting 1

        T0MCR = MR0I | MR0R; //Set bit0 & bit1 to High which is to : Interrupt & Reset TC on MR0

        //----------Setup Timer0 Interrupt-------------
        VICVectAddr4 = (unsigned )T0ISR; //Pointer Interrupt Function (ISR)

        VICVectCntl4 = 0x20 | 4; //0x20 (i.e bit5 = 1) -> to enable Vectored IRQ slot
                                    //0x4 (bit[4:0]) -> this the source number - here its timer0 which has VIC channel mask # as 4
                                    //You can get the VIC Channel number from Lpc214x manual R2 - pg 58 / sec 5.5

        VICIntEnable = 0x10; //Enable timer0 int

        T0TCR = 0x02; //Reset Timer
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 1 Example code (4)

```
__irq void T0ISR(void)
{
        long int regVal;
        regVal = T0IR;              // Read current IR value

        IO0PIN = ~IO0PIN;           // Toggle all pins in Port 0

        T0IR = regVal;              // Write back to IR to clear Interrupt Flag
        VICVectAddr = 0x0;          // This is to signal end of interrupt execution
}

void initClocks(void)
{
  // This function is used to config PPL0 and setup both
  // CPU and Peripheral clock @ 60Mhz
  // You can find its definition in the attached files or case #2 source
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 2 Example code (1)

```
/*
(C) Umang Gajera | Power_user_EX - www.ocfreaks.com 2011-13.
More Embedded tutorials @ www.ocfreaks.com/cat/embedded

LPC2148 Interrupt Example.
License : GPL.
*/

#include <lpc214x.h>

#define PLOCK              0x00000400
#define MR0I               (1<<0)     // Interrupt When TC matches MR0
#define MR1I               (1<<3)     // Interrupt When TC matches MR1
#define MR2I               (1<<6)     // Interrupt When TC matches MR2
#define MR2R               (1<<7)     // Reset TC when TC matches MR2

#define MR0I_FLAG          (1<<0)     // Interrupt Flag for MR0
#define MR1I_FLAG          (1<<1)     // Interrupt Flag for MR1
#define MR2I_FLAG          (1<<2)     // Interrupt Flag for MR2

#define MR0_DELAY_MS       500        // 0.5 Second(s) Delay
#define MR1_DELAY_MS       1000       // 1 Second Delay
#define MR2_DELAY_MS       1500       // 1.5 Second(s) Delay
#define PRESCALE           60000      // 60000 PCLK clock cycles to increment TC by 1
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 2 Example code (2)

```
void delayMS(unsigned int milliseconds);
void initClocks(void);
void initTimer0(void);
__irq void myTimer0_ISR(void);

void setupPLL0(void);
void feedSeq(void);
void connectPLL0(void);


int main(void)
{
        initClocks();              // Initialize CPU and Peripheral Clocks @ 60Mhz
        initTimer0();              // Initialize Timer0
        IO0DIR = 0xFFFFFFFF;       // Configure all pins on Port 0 as Output
        IO0PIN = 0x0;

        T0TCR = 0x01;              // Enable timer

        while(1);                  // Infinite Idle Loop

        //return 0;                // normally this wont execute ever        :P
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 2 Example code (3)

```
void initTimer0(void)
{
        /*Assuming that PLL0 has been setup with CCLK = 60Mhz and PCLK also = 60Mhz.*/

        //----------Configure Timer0-------------
        T0CTCR = 0x0;
        T0PR = PRESCALE-1;                      // 60000 clock cycles @60Mhz = 1 mS
        T0MR0 = MR0_DELAY_MS-1;                 //  0.5sec (Value in Decimal!) Zero Indexed Count - hence subtracting 1
        T0MR1 = MR1_DELAY_MS-1;                 // 1sec
        T0MR2 = MR2_DELAY_MS-1;                 // 1.5secs
        T0MCR = MR0I | MR1I | MR2I | MR2R;      // Set the Match control register

        //----------Setup Timer0 Interrupt-------------    // I've just randomly picked-up slot 4
        VICVectAddr4 = (unsigned) myTimer0_ISR; // Pointer Interrupt Function (ISR)
        VICVectCntl4 = 0x20 | 4;
        VICIntEnable = 0x10;                    // Enable timer0 int
        T0TCR = 0x02;                           // Reset Timer
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 2 Example code (4)

```
__irq void myTimer0_ISR(void)
{
        long int regVal;
        regVal = T0IR;                      // read the current value in T0's Interrupt Register

        if( T0IR & MR0I_FLAG )    {
                //do something for MR0 match

                IO0PIN ^= (1<<0);           // Toggle GPIO0 PIN0 .. P0.0
        }
        else if ( T0IR & MR1I_FLAG )     {
                //do something for MR1 match

                IO0PIN ^= (1<<1);           // Toggle GPIO0 PIN1 .. P0.1
        }
        else if ( T0IR & MR2I_FLAG )     {
                //do something for MR0 match

                IO0PIN ^= (1<<2);           // Toggle GPIO0 PIN2 .. P0.2
        }

        T0IR = regVal;                      // write back to clear the interrupt flag
        VICVectAddr = 0x0;                  // Acknowledge that ISR has finished execution
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 2 Example code (5)

```
void initClocks(void)
{
        setupPLL0();
        feedSeq();                // sequence for locking PLL to desired freq.
        connectPLL0();
        feedSeq();                // sequence for connecting the PLL as system clock

        // SysClock is now ticking @ 60Mhz!

        VPBDIV = 0x01;            // PCLK is same as CCLK i.e 60Mhz

        // PLL0 Now configured!
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 2 Example code (6)

```
//---------PLL Related Functions :---------------

// Using PLL settings as shown in : http://www.ocfreaks.com/lpc214x-pll-tutorial-for-cpu-and-peripheral-clock/

void setupPLL0(void)
{
        // Note : Assuming 12Mhz Xtal is connected to LPC2148.

        PLL0CON = 0x01;
        PLL0CFG = 0x24;
}

void feedSeq(void)
{
        PLL0FEED = 0xAA;
        PLL0FEED = 0x55;
}

void connectPLL0(void)
{
        while( !( PLL0STAT & PLOCK ));
        PLL0CON = 0x03;
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 3 Example code (1)

```
//----------Setup TIMER0 Interrupt-------------      // Using Slot 0 for TIMER0
VICVectAddr0 = (unsigned) myTimer0_ISR;   // Pointer Interrupt Function (ISR)

VICVectCntl0 = 0x20 | 4;

VICIntEnable |= (1<<4);                   // Enable timer0 int , 4th bit=1

//----------Setup UART0 Interrupt-------------       / /Any Slot with Lower Priority than TIMER0's slot will suffice
VICVectAddr1 = (unsigned) myUart0_ISR;    // Pointer Interrupt Function (ISR)

VICVectCntl1 = 0x20 | 6;

VICIntEnable |= (1<<6); //Enable Uart0 interrupt , 6th bit=1
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 3 Example code (2)

```
__irq void myTimer0_ISR(void)
{
        long int regVal;
        regVal = T0IR;              // read the current value in T0's Interrupt Register

        IO0PIN ^= (1<<2);           // Toggle 3rd Pin in GPIO0 .. P0.2

        T0IR = regVal;              // write back to clear the interrupt flag
        VICVectAddr = 0x0;          // Acknowledge that ISR has finished execution
}

__irq void myUart0_ISR(void)
{
        long int regVal;
        regVal = U0IIR;             // Reading U0IIR also clears it!

        //Recieve Data Available Interrupt has occured
        regVal = U0RBR;             // dummy read
        IO0PIN ^= (1<<3);           // Toggle 4th Pin in GPIO0 .. P0.3

        VICVectAddr = 0x0;          // Acknowledge that ISR has finished execution
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 4 Example code (1)

```
VICDefVectAddr = (unsigned) myDefault_ISR;         // Pointer to Default ISR

//----------Enable (Non-Vectored) TIMER0 Interrupt-------------
VICIntEnable |= (1<<4);                            // Enable timer0 int , 4th bit=1

//----------Enable (Non-Vectored) UART0 Interrupt-------------
VICIntEnable |= (1<<6);                            // Enable Uart0 interrupt , 6th bit=1
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Case 4 Example code (2)

```
__irq void myDefault_ISR(void)
{
        long int T0RegVal , U0RegVal;
        T0RegVal = T0IR;                      // read the current value in T0's Interrupt Register
        U0RegVal = U0IIR;

        if( T0IR )
        {
                IO0PIN ^= (1<<2);             // Toggle 3rd Pin in GPIO0 .. P0.2
                T0IR = T0RegVal;              // write back to clear the interrupt flag
        }

        if( !(U0RegVal & 0x1) )
        {
                //Recieve Data Available Interrupt has occured
                U0RegVal = U0RBR;             // dummy read
                IO0PIN ^= (1<<3);             // Toggle 4th Pin in GPIO0 .. P0.3
        }

        VICVectAddr = 0x0; // Acknowledge that ISR has finished execution
}
```

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Example code 1 (Case 1)

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

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