Nesting, Tail Chaining, and Late Arrival

Copyright (c) 2023 - 2014 Young W. Lim.

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

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

This document was produced by using LibreOffice.

Based on

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

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

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

ARM assembler in Raspberry Pi
Roger Ferrer Ibáñez

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

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

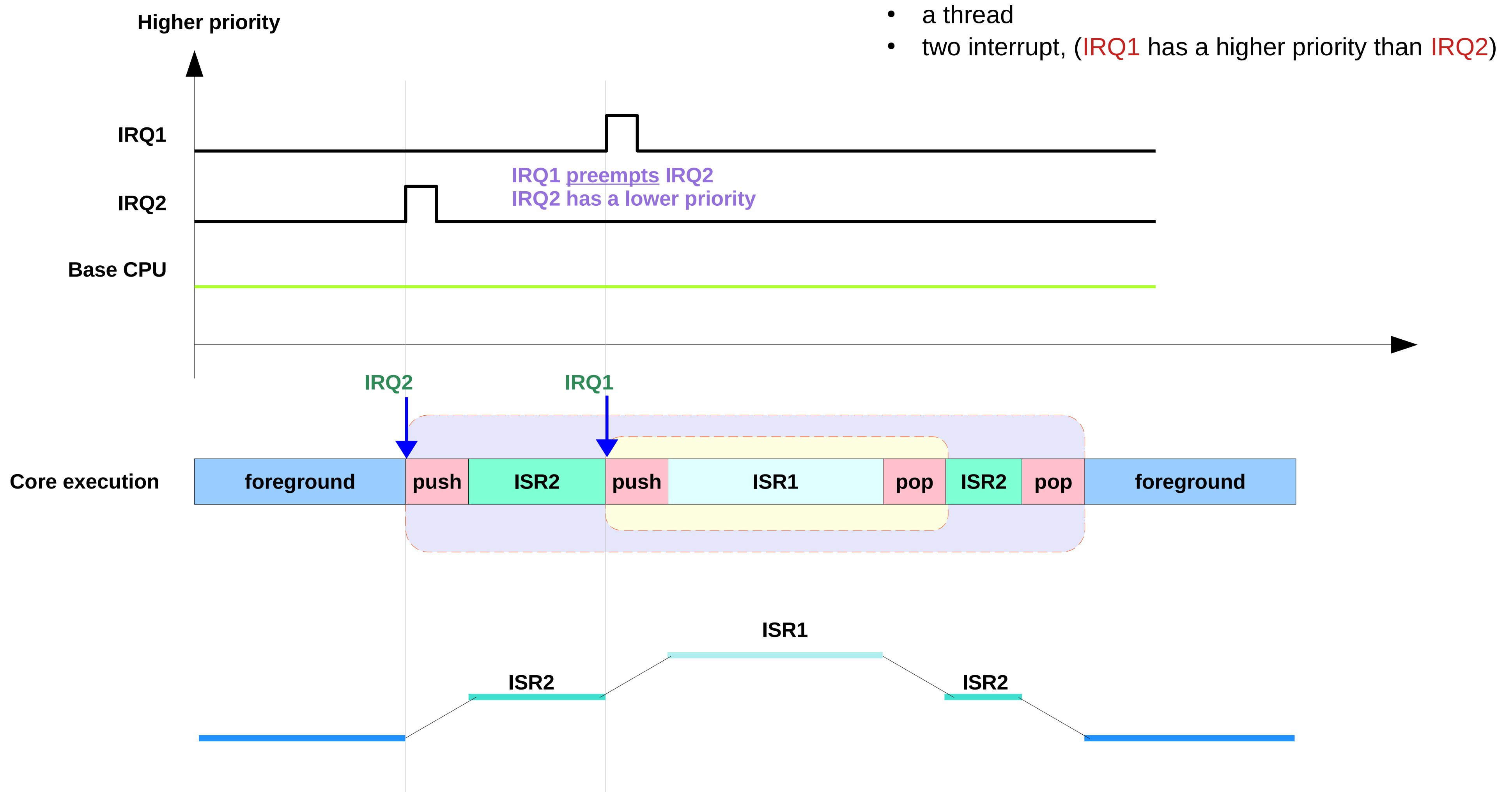
- **tail-chaining**
  allows additional handlers to be executed
  without additional pushing and popping of registers.

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)

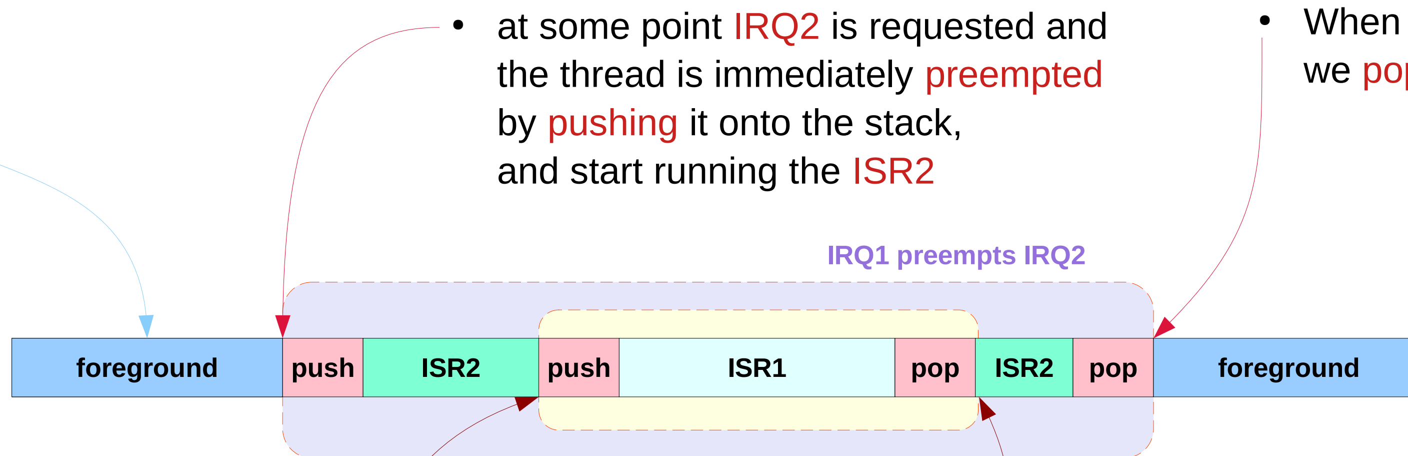
- a thread
- two interrupt, (IRQ1 has a higher priority than IRQ2)



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.

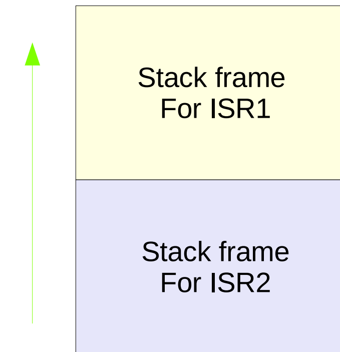
https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Nesting (3)

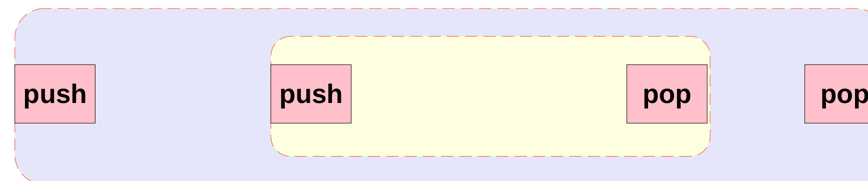
- The benefit
  - *distinct* levels of priority
  - *always* working on the most important task
  - *minimize* the interrupt latency
    for the highest priority interrupt at any time.

- 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

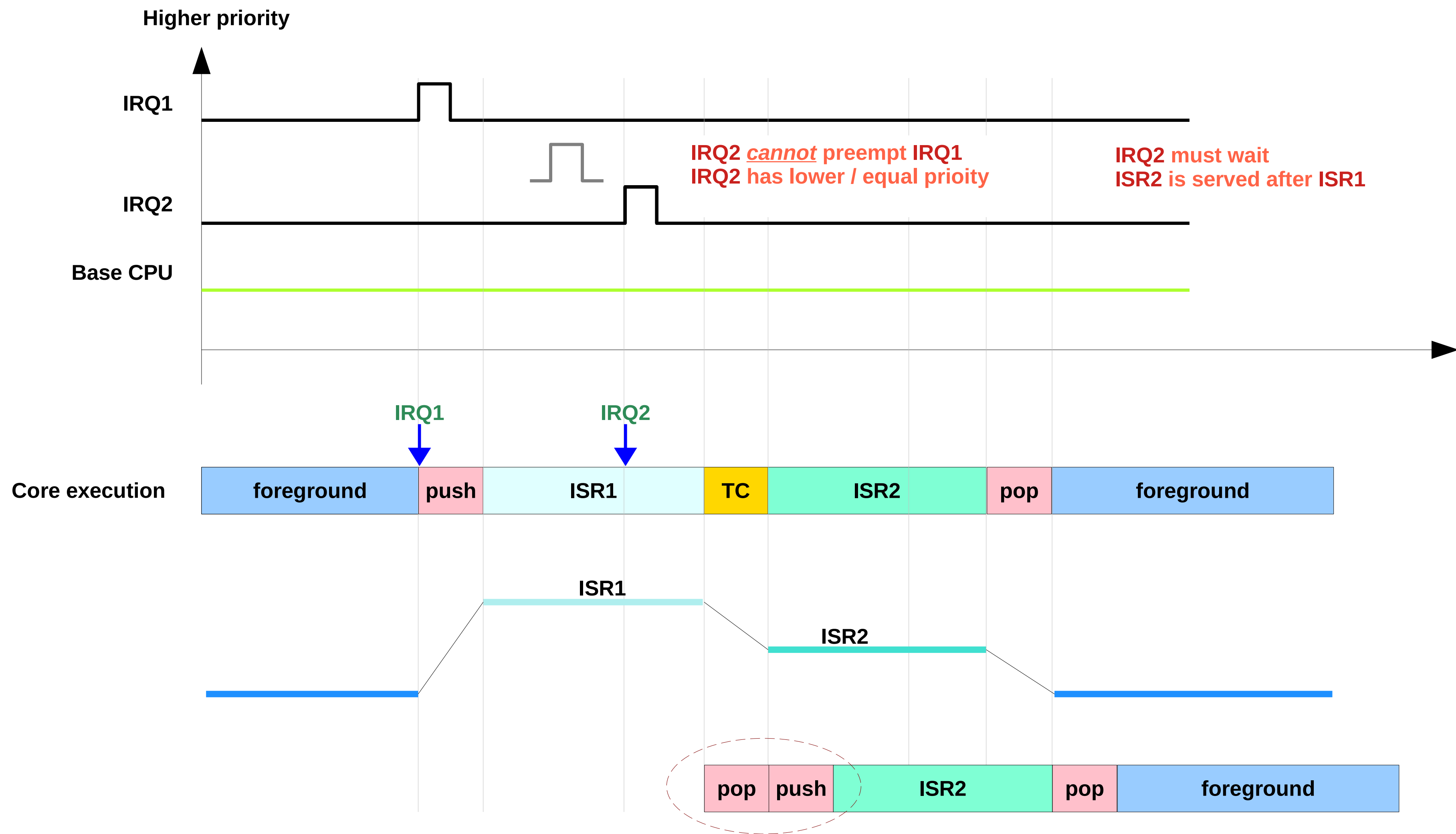


increasing stack



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

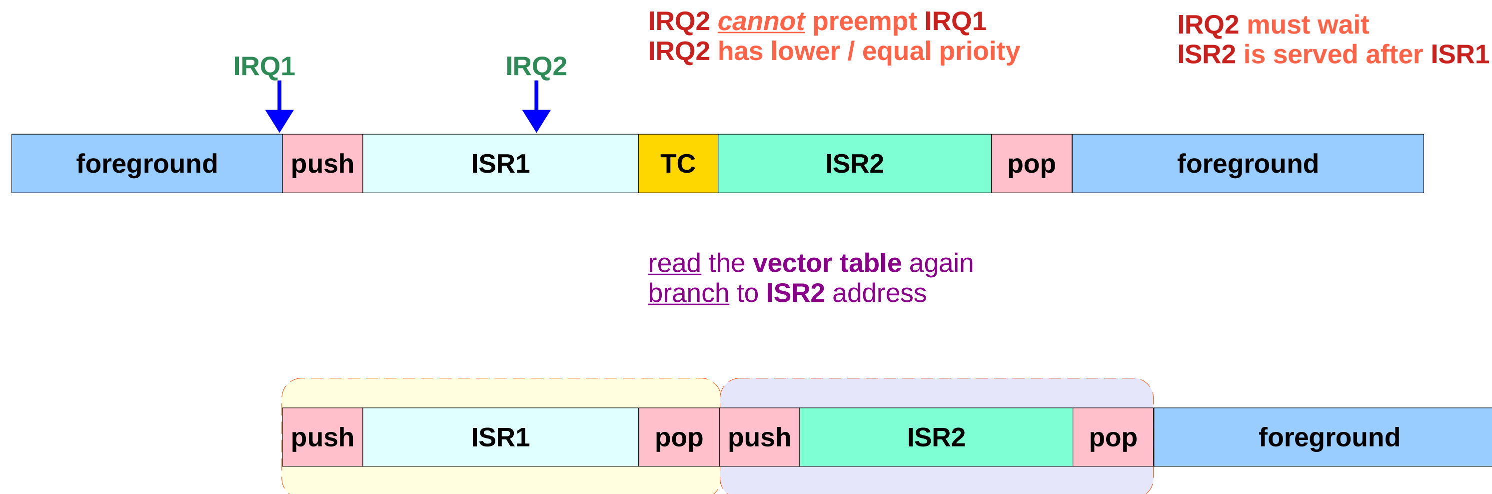
Tail chaining (2)

- 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.

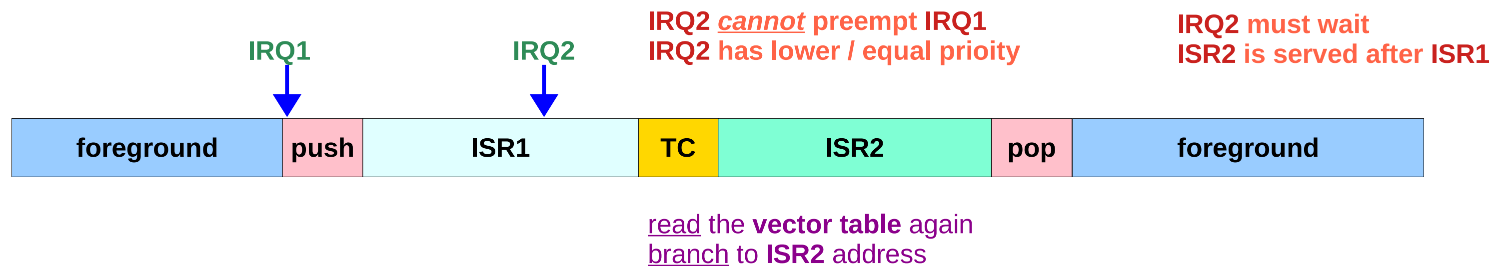


https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

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.



- 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

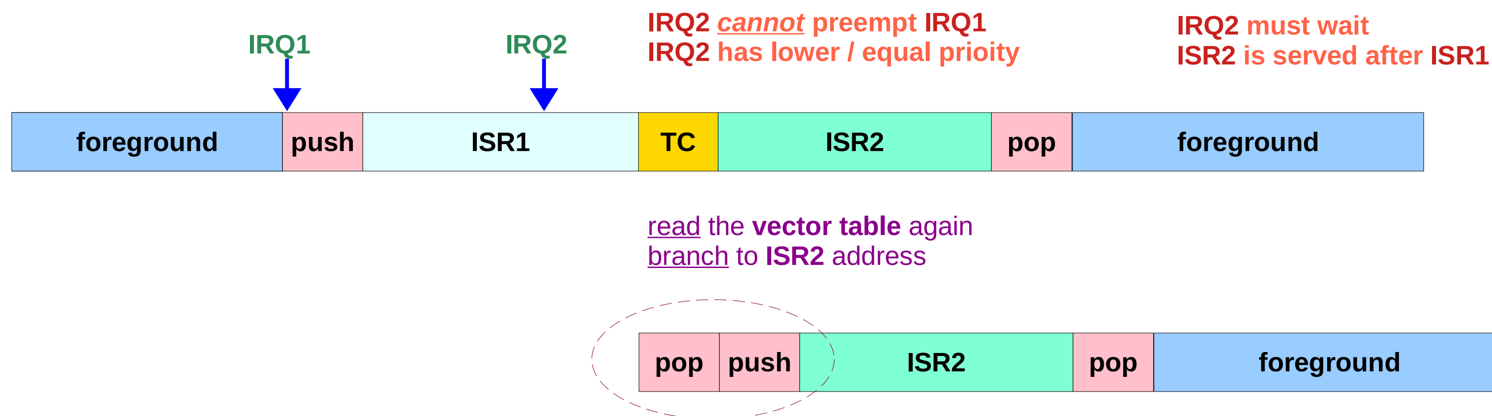
Tail chaining (4)

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*

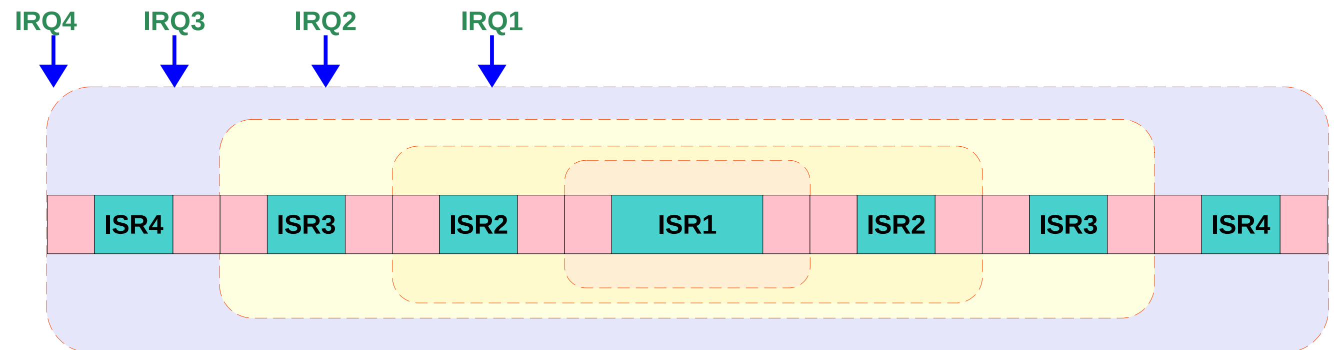


https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

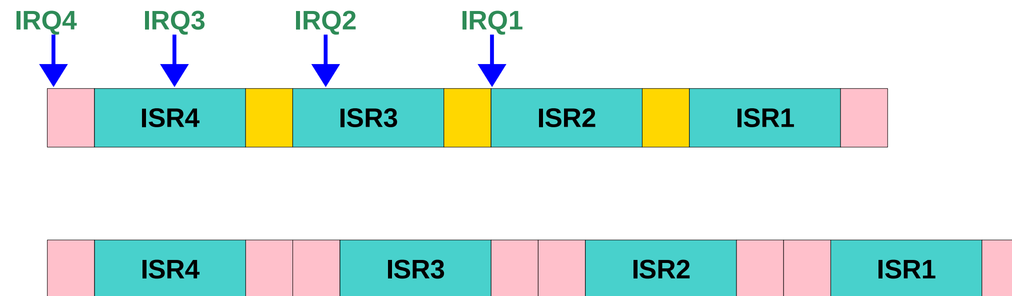
Tail chaining (5)

- 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

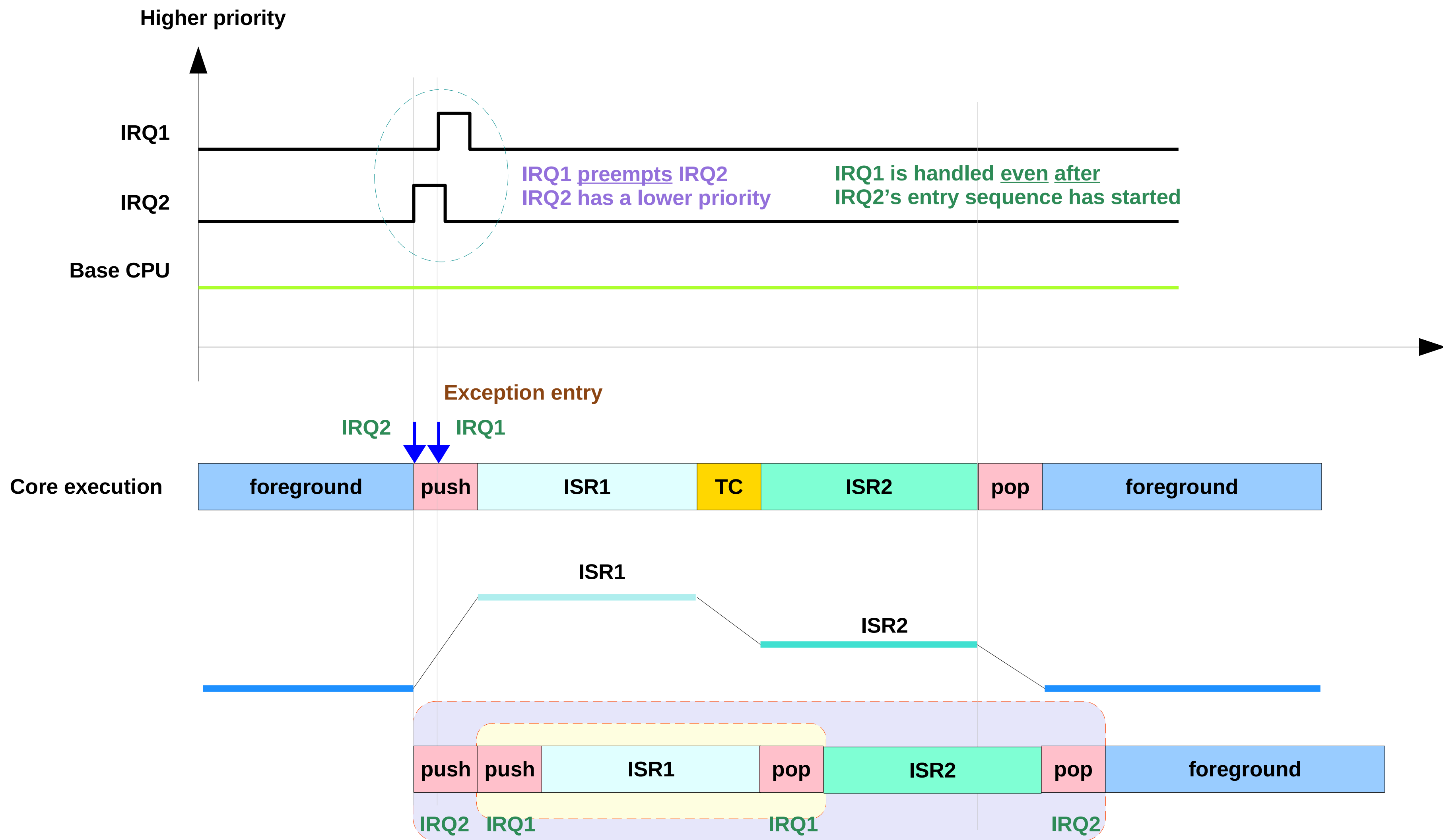


Priority (IRQ4)
= Priority (IRQ3)
= Priority (IRQ2)
= Priority (IRQ1)
the same priority level



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

Late arrival **A** (1)



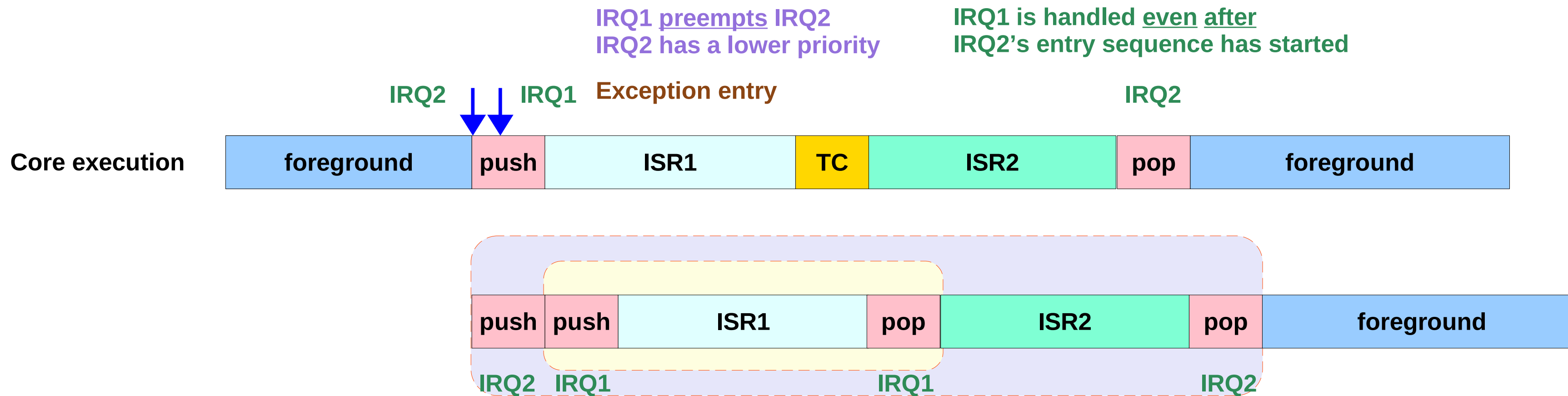
https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

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

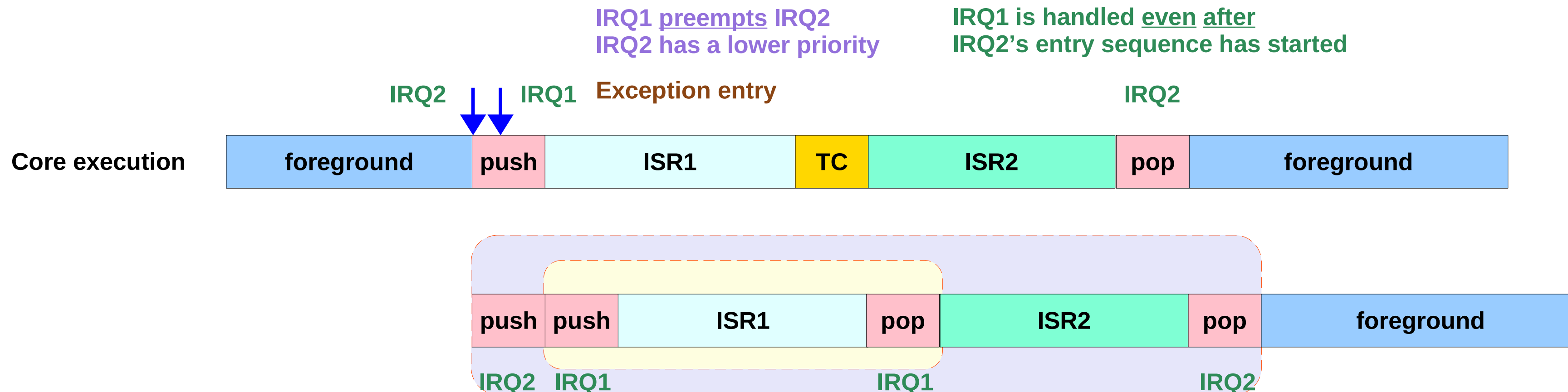
- the lower priority exception is handled
  after the higher priority exception is completed



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

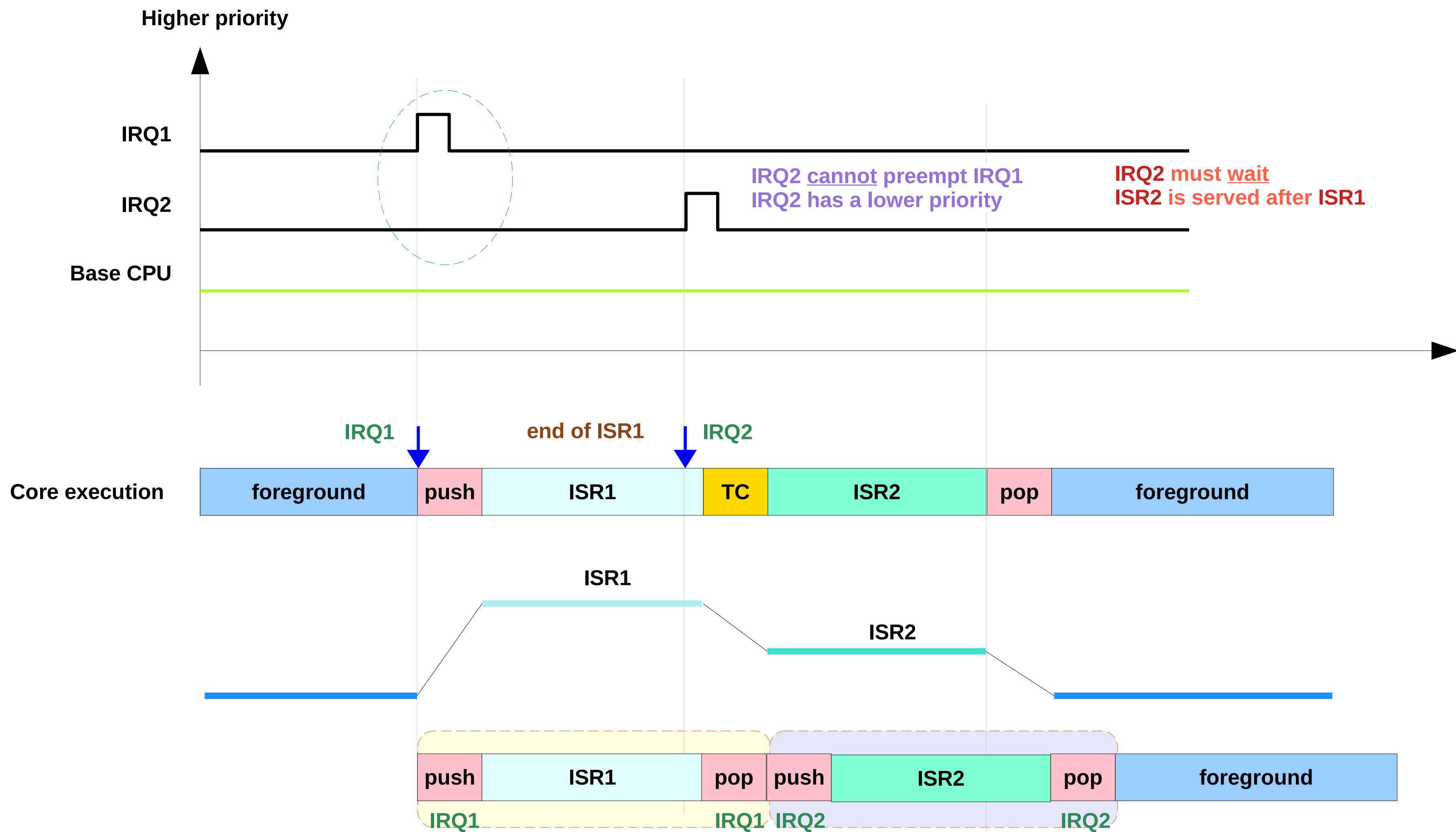
Late arrival **A** (3)

- also, in the case of the late-arriving interrupt, the processor might execute its ISR after fewer 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 not need to restart the stack push, so some cycles may be saved.



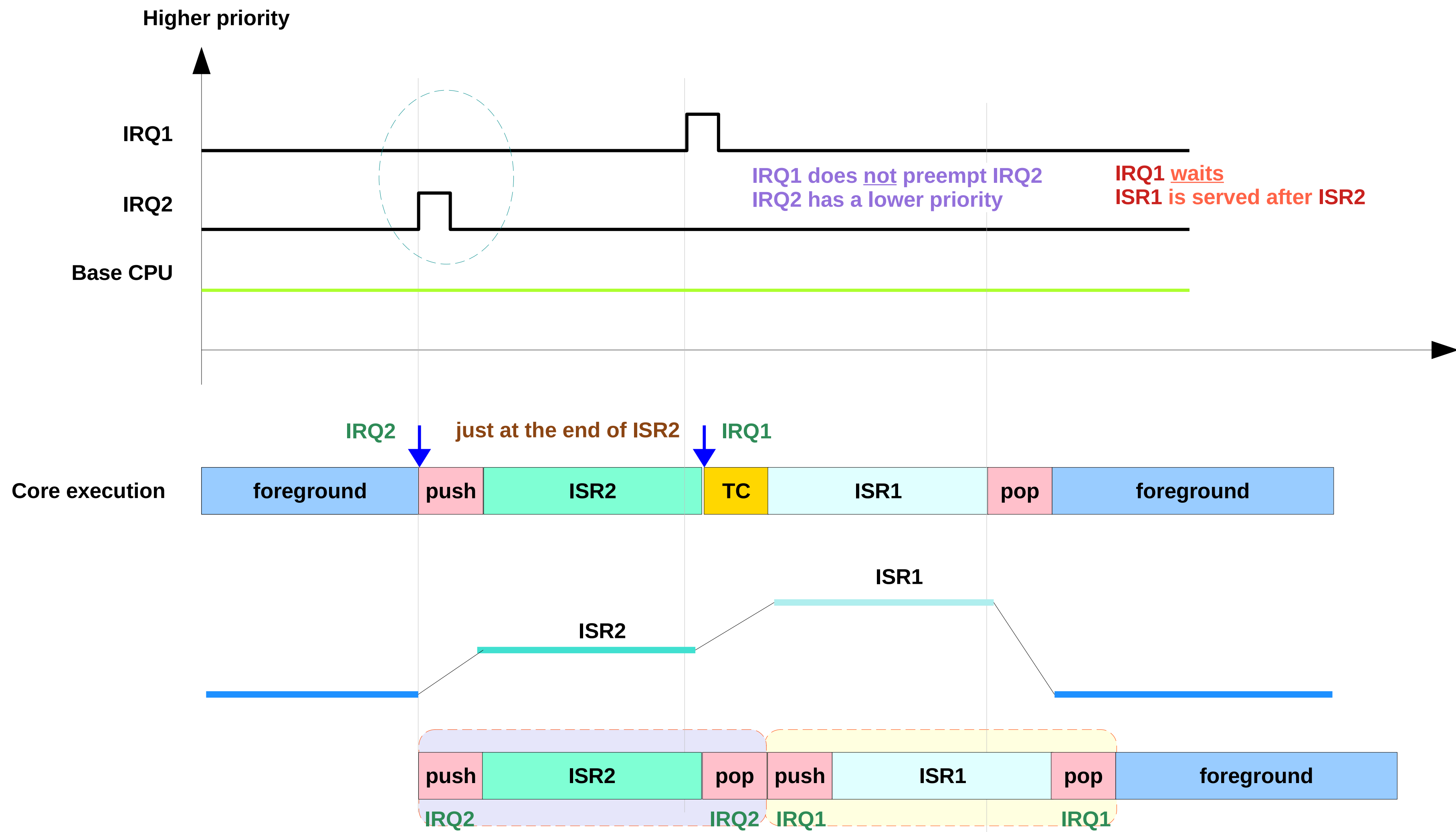
https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

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)

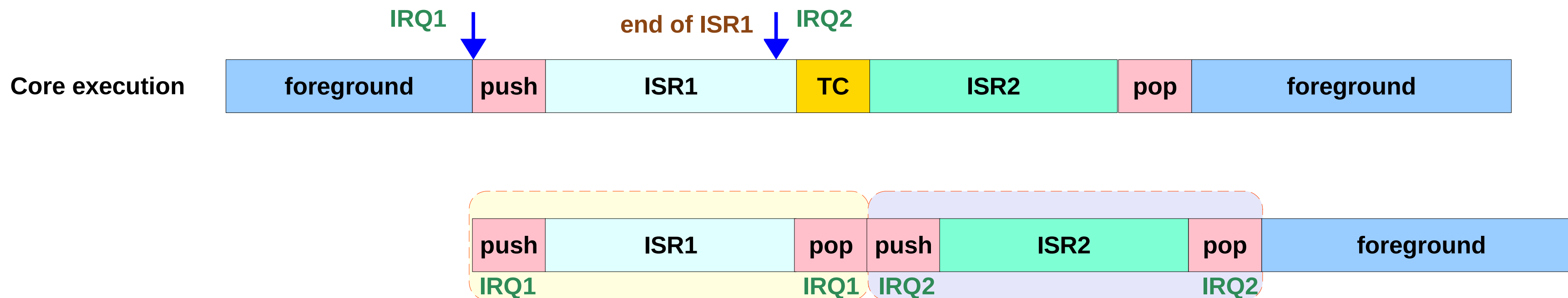
- A similar case arises
  if a new interrupt ISR2 arrives
  just before the end of an ISR1,

- 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.

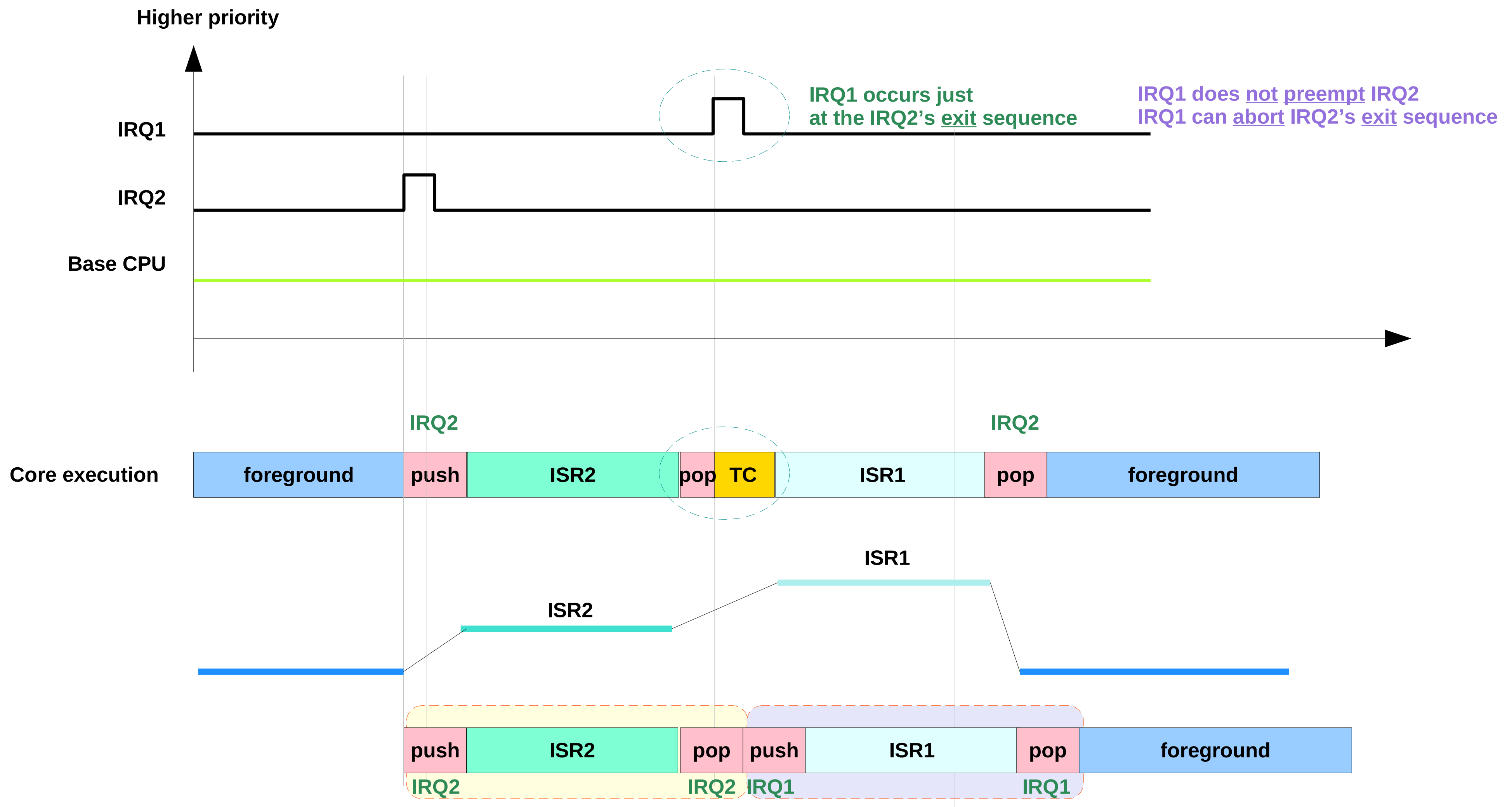
- Again, the vector table needs to be read
  to access the new ISR1,
  but tail-chaining does not require
  any stacking operation

- 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)



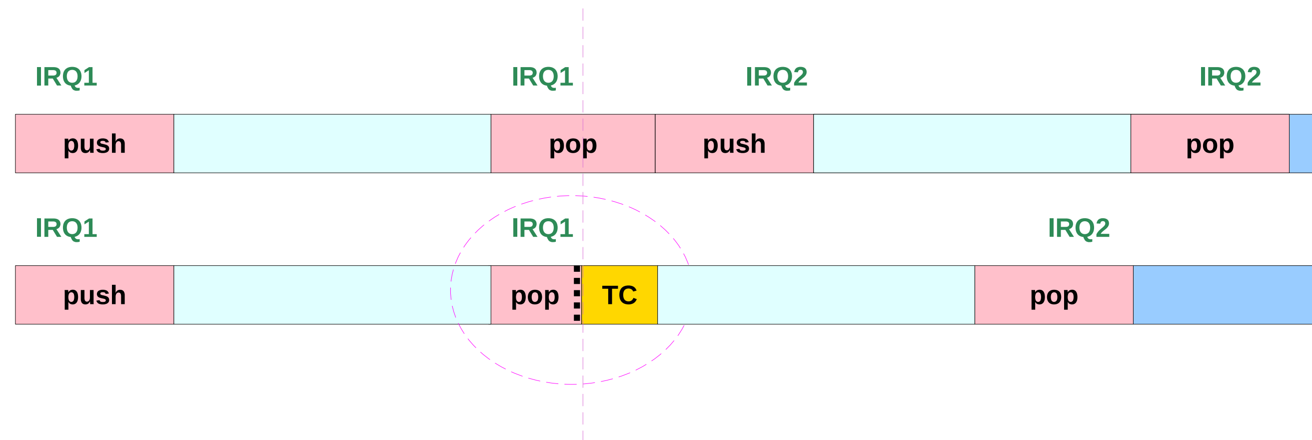
https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

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.

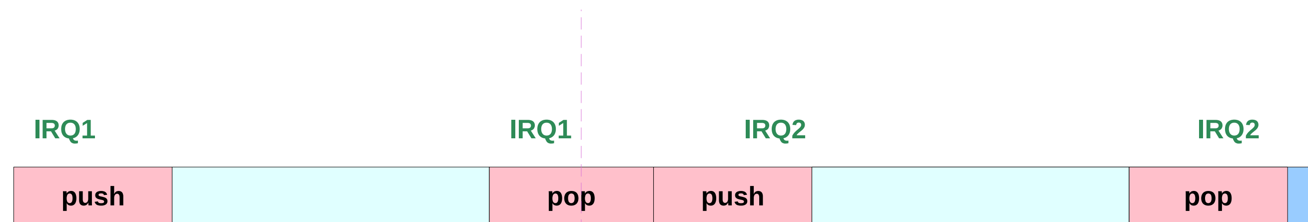
- In **Cortex M**,
  the stack pop can simply be abandoned,
  leaving the stack frame on the stack,
  and only a tail-chain is then needed
  to enter the new ISR.



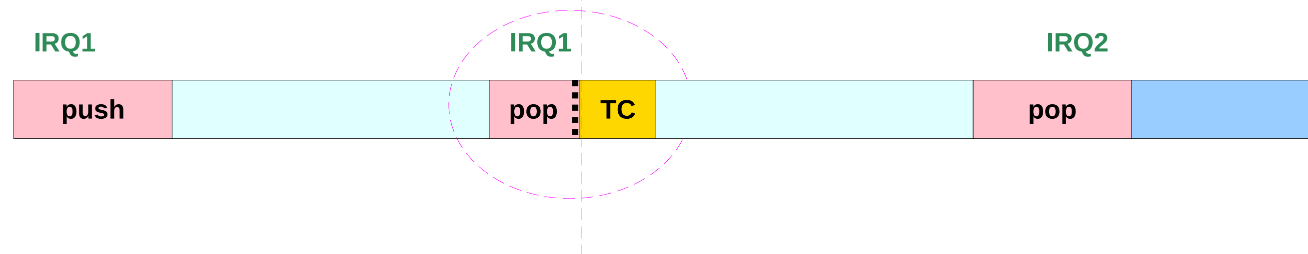
https://www.coursera.org/lecture/armv8-m-architecture-fundamentals/nesting-tail-chaining-and-late-arriving-examples-FmA6E

Late arrival **C** (3)

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

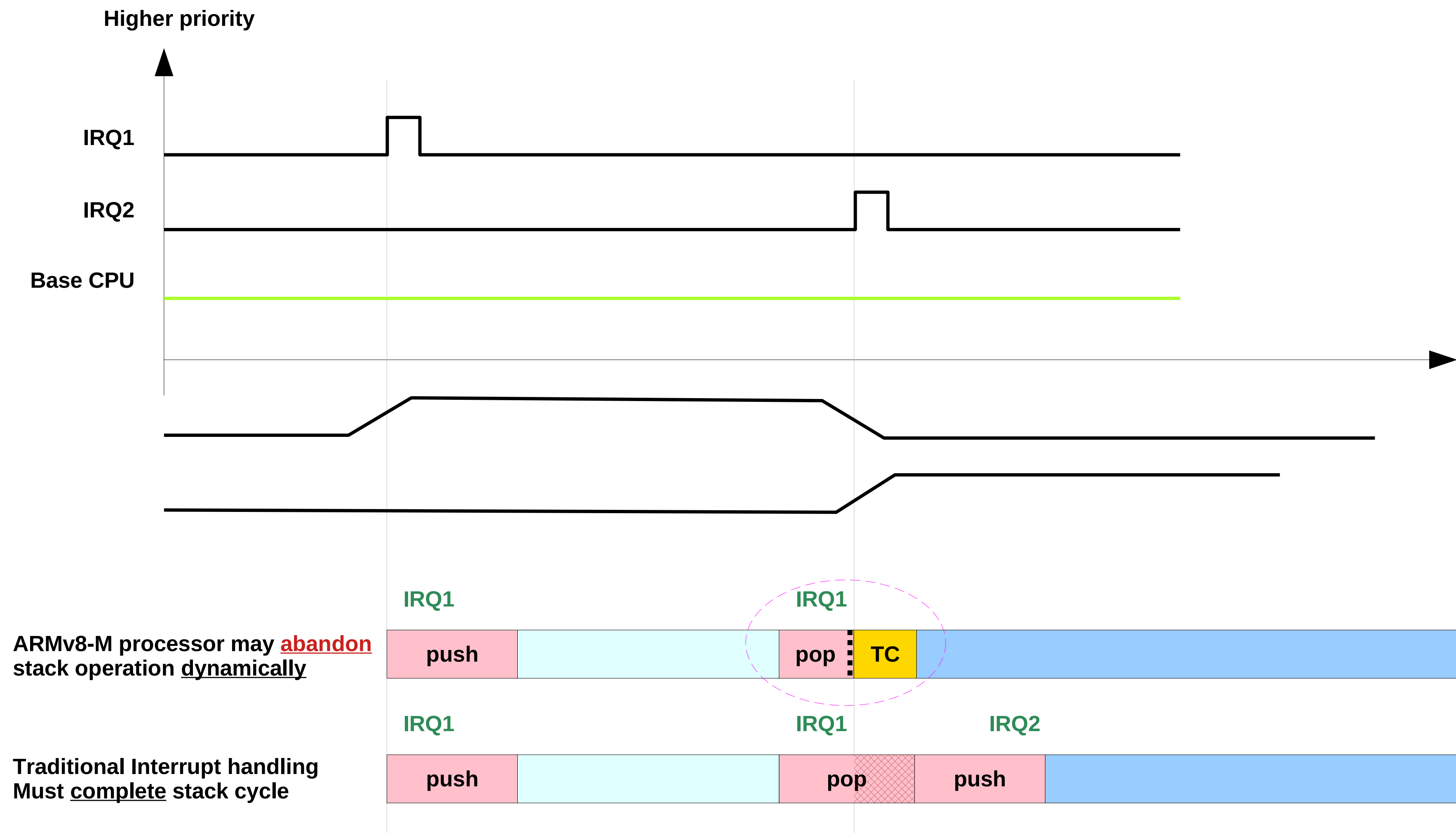


- **ARMv8-M** Processor
- POP may be abandoned early
- if another Interrupt arrives
- If POP is interrupted,
- the new handler can be fetched directly



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

Late arrival **C** (4)



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)

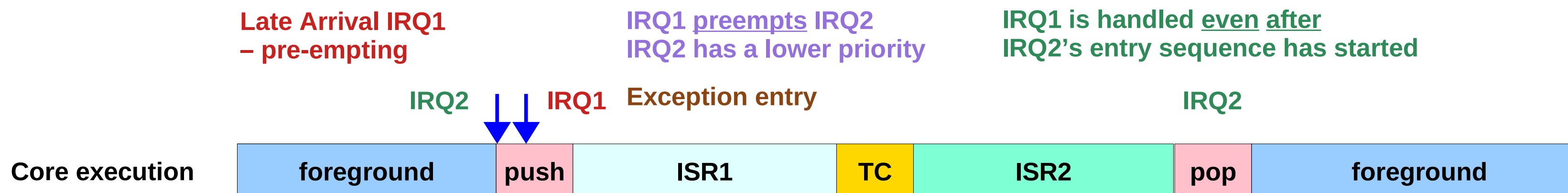
- A late-arriving interrupt is an interrupt which is recognized after the processor has started its exception entry procedure.

- 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)

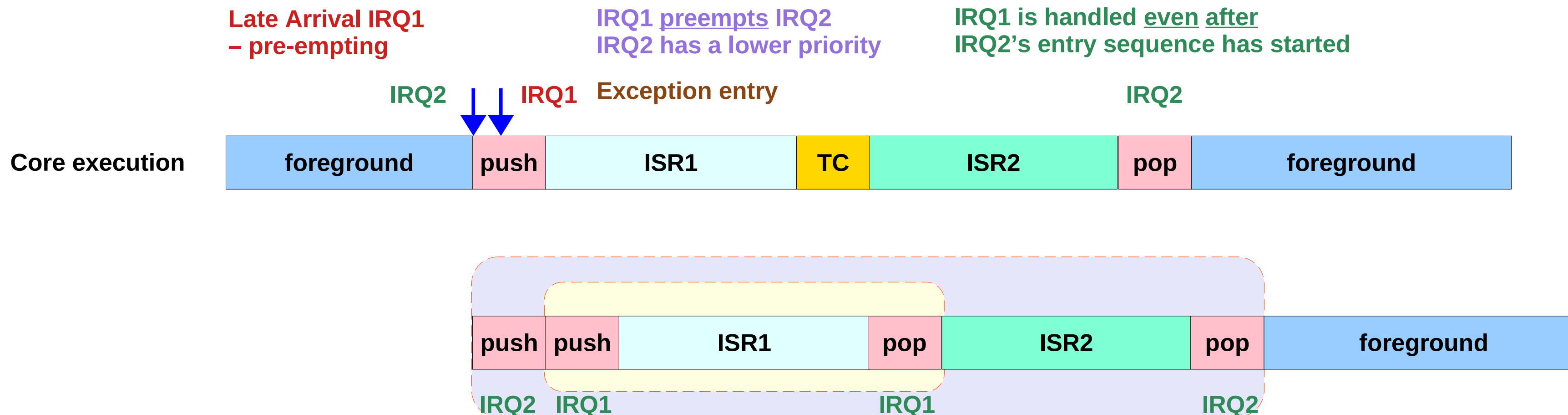
- This guarantees that the interrupt with the highest pre-empting priority will be serviced first, but in some circumstances this results in some wasted cycles from the original vector fetch which was abandoned.

after starting an exception entry, other interrupts are requested

current ***stack operation – utilized***

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

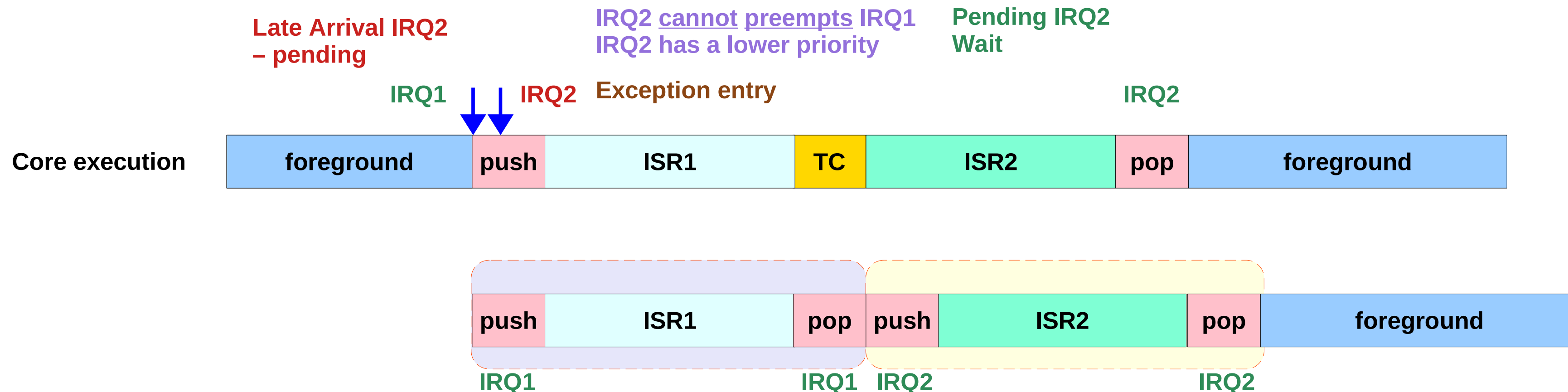
→ wasted cycles



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

Late Arrival (3)

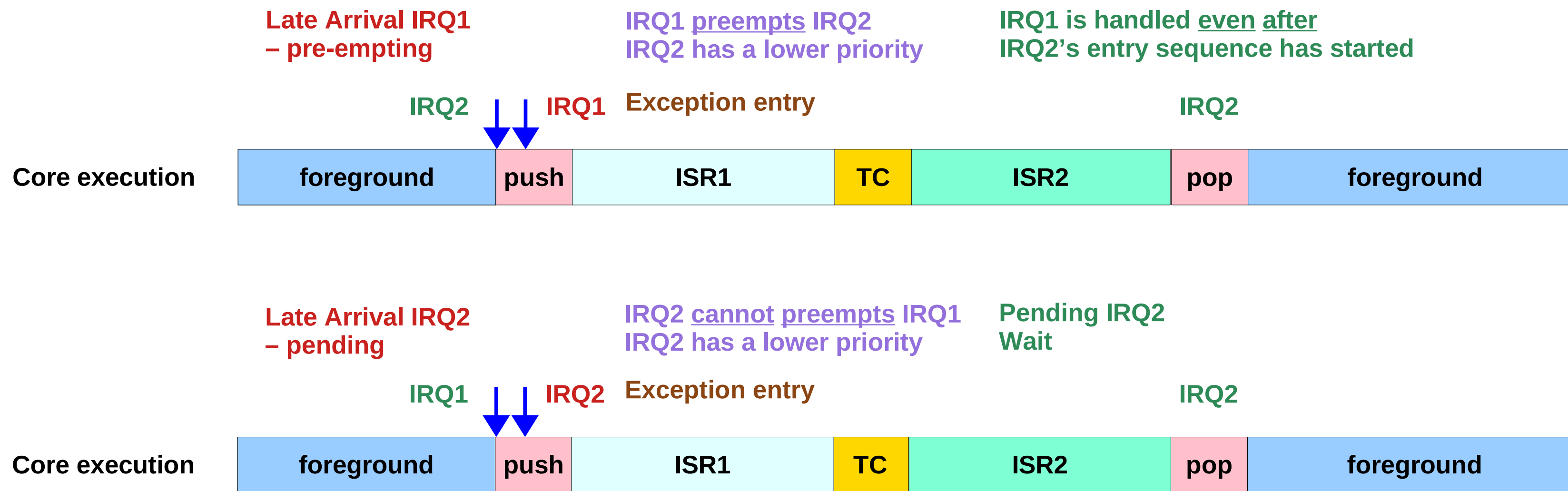
- If the late-arriving interrupt
  has only equal priority to (or lower priority than)
  the exception which the processor
  has already started to handle,
  then the late-arriving interrupt
  will remain pending until
  after the exception handler
  for the current exception has run



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.



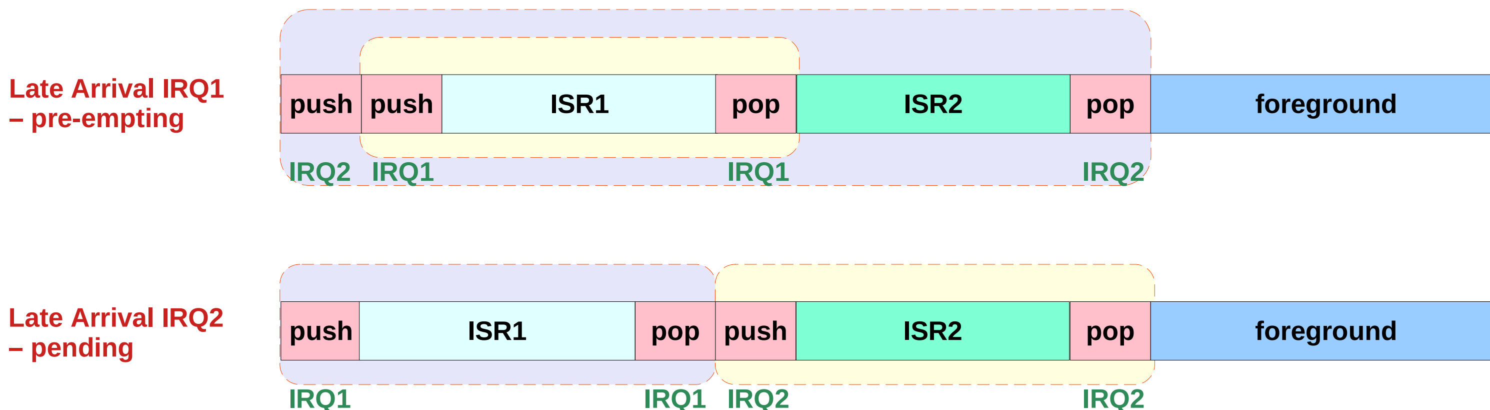
https://developer.arm.com/documentation/ka001190/latest

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.

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

Interruptible LDM, STM (4)

- 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 (6)

- **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 thumb 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

If-Then (IT) block

- The **If-Then** (**IT**) block contains
  up to four instructions following an **IT** instruction
- each instruction in the block is conditional.
- the conditions for the instructions are
  - either all the same, or — Then
  - some can be the inverse of others. — Else

```
ITTET EQ
ADD r0,r0,r0
ADD r1,r0,r0
ADD r2,r0,r0
ADD r3,r0,r0
```

IT block size = 4

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

If-Then (IT) block examples

First note that most instruction can specify a condition code
in ARM instruction, not in Thumb.

With **IT** instruction, you can specify condition code for up to 4 instructions.
For each instruction, you specify if it's part of the If (T) or Else (E).

For example:

```
ITTET EQ                Then, Then, Else, Then → EQ, EQ, NE, EQ
ADD r0,r0,r0
ADD r1,r0,r0
ADD r2,r0,r0
ADD r3,r0,r0
```

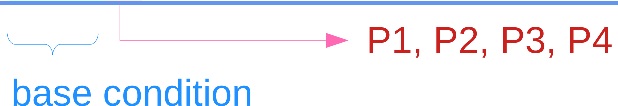
Will actually translate to:

```
ADDEQ r0,r0,r0  (Always if for 1st one)
ADDEQ r1,r0,r0  (T for 2nd one)
ADDNE r2,r0,r0  (E for 3rd one)
ADDEQ r3,r0,r0  (T for 4th one)
```

https://stackoverflow.com/questions/36558926/what-does-the-arm7-it-if-then-instruction-really-do

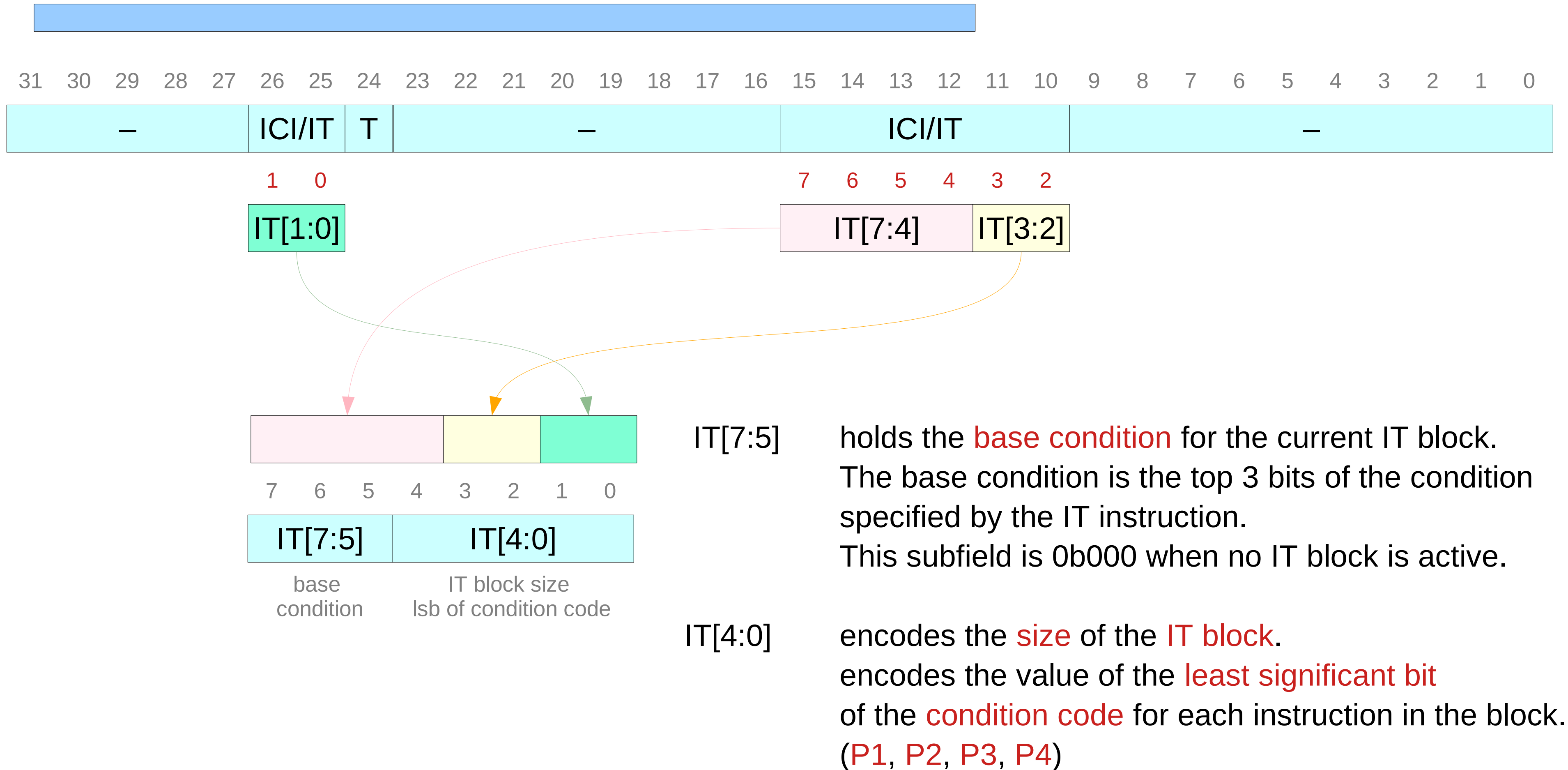
Condition Code Encoding

Code	Suffix	Flags	Meaning
0 0 0 0	EQ	Z set	equal
0 0 0 1	NE	Z clear	not equal
0 0 1 0	CS	C set	unsigned higher or same
0 0 1 1	CC	C clear	unsigned lower
0 1 0 0	MI	N set	negative
0 1 0 1	PL	N clear	positive or zero
0 1 1 0	VS	V set	overflow
0 1 1 1	VC	V clear	no overflow
1 0 0 0	HI	C set and Z clear	unsigned higher
1 0 0 1	LS	C clear or Z set	unsigned lower or same
1 0 1 0	GE	N equals V	greater or equal
1 0 1 1	LT	N not equal to V	less than
1 1 0 0	GT	Z clear AND (N equals V)	greater than
1 1 0 1	LE	Z set OR (N not equal to V)	less than or equal
1 1 1 0	AL	(ignored)	always
1 1 1 1	rsvd		



http://www.csbio.unc.edu/mcmillan/Media/arm-instructionset.pdf

ESPR fields for IT block (1)

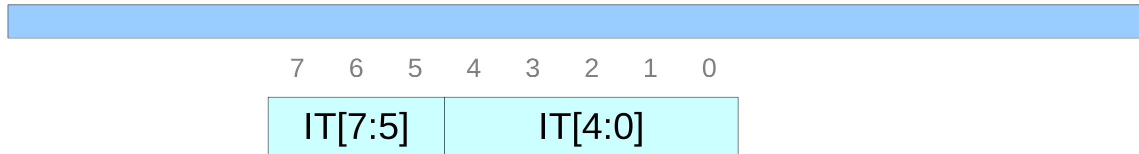


IT[7:5] holds the base condition for the current IT block.
The base condition is the top 3 bits of the condition specified by the IT instruction.
This subfield is 0b000 when no IT block is active.

IT[4:0] encodes the size of the IT block.
encodes the value of the least significant bit of the condition code for each instruction in the block.
(P1, P2, P3, P4)

https://developer.arm.com/documentation/ddi0337/e/ch02s03s02

ESPR fields for IT block (2)



ITTET EQ
ADD r0,r0,r0
ADD r1,r0,r0
ADD r2,r0,r0
ADD r3,r0,r0

IT block size = 4

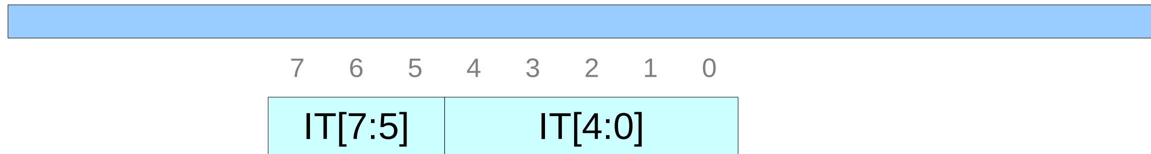
IT[4:0] encodes the size of the IT block.
This is the number of instructions
that are to be conditionally executed.
The size of the block is implied by the position
of the least significant **1** in this field

encodes the value of the least significant bit
of the condition code for each instruction in the block.
(P1, P2, P3, P4)

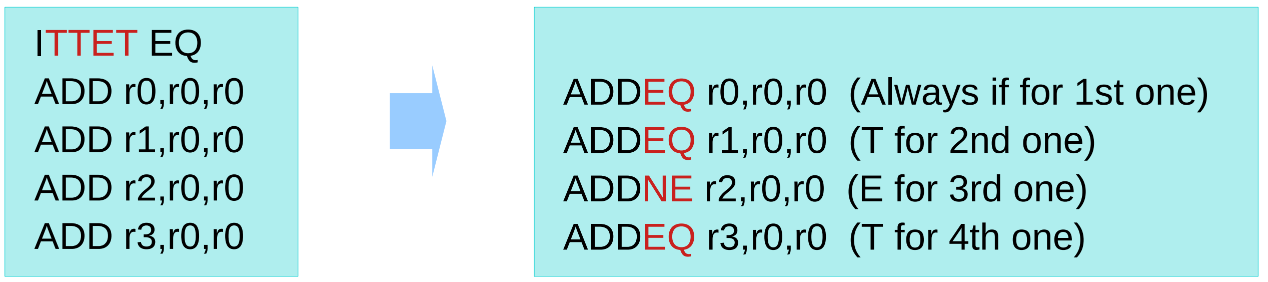
[7:5]	[4]	[3]	[2]	[1]	[0]	
cond_base	P1	P2	P3	P4	**1**	Entry point for 4-instruction IT block
cond_base	P1	P2	P3	**1**	0	Entry point for 3-instruction IT block
cond_base	P1	P2	**1**	0	0	Entry point for 2-instruction IT block
cond_base	P1	**1**	0	0	0	Entry point for 1-instruction IT block
000	**0**	0	0	0	0	Normal execution, not in an IT block

https://developer.arm.com/documentation/ddi0337/e/ch02s03s02

ESPR fields for IT block (3)



[7:5]	[4]	[3]	[2]	[1]	[0]	
cond_base	P1	P2	P3	P4	**1**	Entry point for 4-instruction IT block
cond_base	P1	P2	P3	**1**	0	Entry point for 3-instruction IT block
cond_base	P1	P2	**1**	0	0	Entry point for 2-instruction IT block
cond_base	P1	**1**	0	0	0	Entry point for 1-instruction IT block
000	**0**	0	0	0	0	Normal execution, not in an IT block



EQ condition code = 0000
NE condition codc = 0001

base code = 000
P1=0, P2=0, P3=1, P4=0 TTET

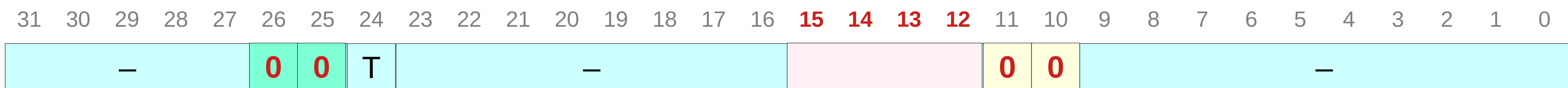
IT[7:5] = 000, IT[4:0] = 0010**1**

https://developer.arm.com/documentation/ddi0337/e/ch02s03s02

Interruptible-Continuable Instructions (ICI)

- When an interrupt occurs during the execution of an **LDM**, **STM**, **PUSH**, or **POP** instruction, the processor:
  - stops the load multiple or store multiple instruction operation temporarily
  - stores the next register operand in the multiple operation to **EPSR** bits[15:12].
- after servicing the interrupt, the processor:
  - returns to the register pointed to by **EPSR** bits[15:12]
  - resumes execution of the multiple load or store instruction.

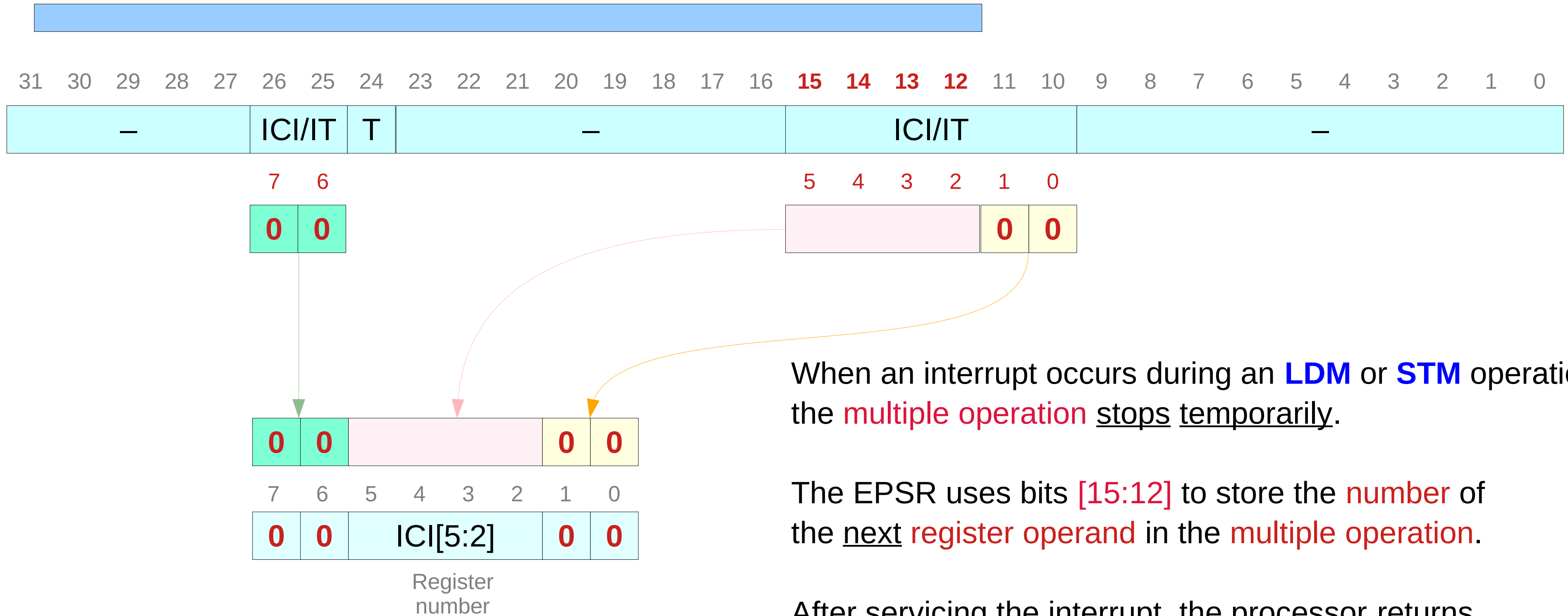
- When the **EPSR** holds ICI execution state bits[26:25,11:10] are zero.



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

7 6

ESPR fields for **ICI** bits (1)



When an interrupt occurs during an **LDM** or **STM** operation, the multiple operation stops temporarily.

The EPSR uses bits [15:12] to store the number of the next register operand in the multiple operation.

After servicing the interrupt, the processor returns to the register pointed to by [15:12] and resumes the multiple operation.

If the ICI field [5:2] points to a register that is not in the register list of the instruction, the processor continues with the next register in the list, if any.

https://developer.arm.com/documentation/ddi0337/e/ch02s03s02

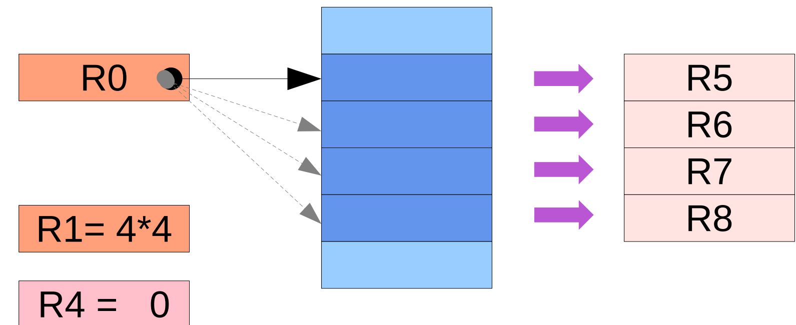
LDM / STM examples (1)

allowing several values to be loaded or stored

For example, **LDMIA** instruction allows
loading into multiple registers (R5, R6, R7, R8)
starting at an address named in another register (R0).

Consider the example of adding the integers of an array.
LDMIA can be used to processes four integers with each iteration of the loop.
In this way, fewer instructions can be used,
at the expense of more complexity.

LDMIA R0**!**, { R5-R8 }

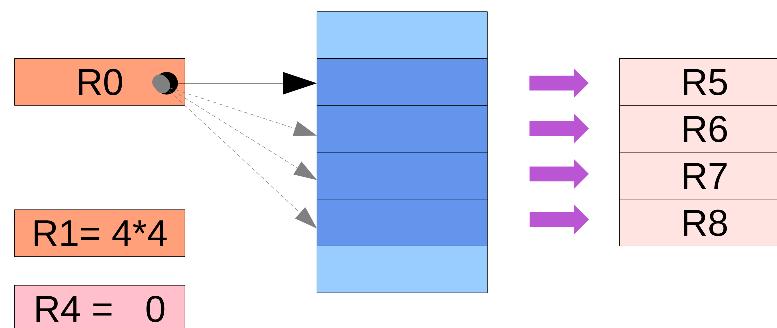


http://www.cburch.com/books/arm/

LDM / STM examples (2)

If the exclamation mark **!** following R0 is omitted,
then the address register R0 is not altered
R0 would continue pointing to the first integer in the array.

we want R0 to change so that it is pointing
to the next four integers for the next iteration,
the exclamation point should be included



LDMIA R0**!**, { R5-R8 }

equivalent instructions without ***!***
pre-indexed

```
LDR   R5,  =[R0, #0]
LDR   R6,  =[R0, #4]
LDR   R7,  =[R0, #8]
LDR   R8,  =[R0, #12]
ADD   R0,  R0, #12
```

equivalent instructions with ***!***
post-indexed

```
LDR   R5,  =[R0], #4
LDR   R6,  =[R0], #4
LDR   R7,  =[R0], #4
LDR   R8,  =[R0], #4
```

http://www.cburch.com/books/arm/

LDM / STM examples (3)

LDMIA R0**!**, { R5-R8 }

equivalent instructions with ***!***
post-indexed

```
LDR     R5,  =[R0], #4
LDR     R6,  =[R0], #4
LDR     R7,  =[R0], #4
LDR     R8,  =[R0], #4
```

the multiple operation

ESPR [15:12] (=**ICI [5:2]**)

could be R5, R6, R7, R8
In this example

https://developer.arm.com/documentation/ddi0337/e/ch02s03s02

Assume an interrupt happens in the middle of
LDR R6 =[R0], #4 operation

When an interrupt occurs during an **LDM** or **STM** operation,
the multiple operation stops temporarily.

```
  LDR R5  =[R0], #4
➡ LDR R6  =[R0], #4
  LDR R7  =[R0], #4      ; the stopped operation
  LDR R8  =[R0], #4
```

The EPSR uses bits [15:12] to store the number of
the next register operand in the multiple operation.
R6

After servicing the interrupt, the processor returns
to the register pointed to by [15:12] and
resumes the multiple operation.

```
  LDR R5  =[R0], #4
➡ LDR R6  =[R0], #4
  LDR R7  =[R0], #4      ; resume the stopped op
  LDR R8  =[R0], #4
```

Thumb state

- The **Cortex-M3** processor only supports execution of instructions in **Thumb state**.
- The following can clear the **T** bit to **0**:
  - instructions **BLX**, **BX** and **POP{PC}**
  - restoration from the stacked **xPSR** value on an exception return
  - bit[0] of the vector value on an exception entry or reset.
- Attempting to execute instructions when the **T** bit is **0** results in a fault or lockup. See Lockup for more information.

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

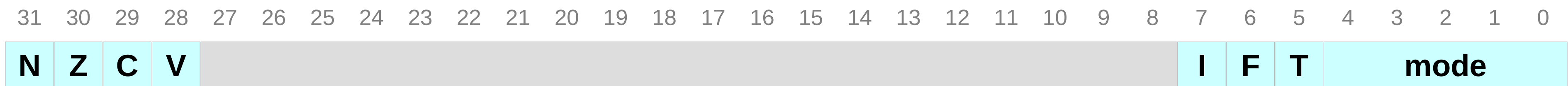
Current state bit in CPSR

- The CPSR register holds
  - processor mode bits (user or exception flag)
  - interrupt mask bits
  - condition codes and
  - Thumb status bit
- The Thumb status bit (**T**) *indicates* the processor's current state:
  - 0 for ARM state (default)
  - 1 for Thumb.
- Although other bits in the CPSR may be modified in software, it's *dangerous to write to **T** directly*;
  - the results of an improper state change are *unpredictable*.

N Negative flag
Z Zero flag
C Carry flag
V Overflow flag

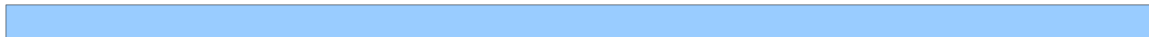
To disable Interrupt (**IRQ**), set **I**
To disable Fast Interrupt (**FIQ**), set **F**

USR	User mode
FIQ	Fast Interrupt mode
SVC	Supervisor mode
ABT	Abort mode
UND	Undefined mode
SYS	System mode



https://www.embedded.com/introduction-to-arm-thumb/

ESPR **T**-bit field



Field	Name	Definition
[24]	T	

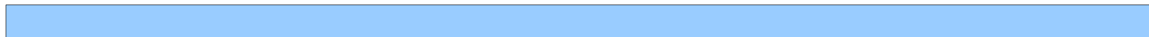
The T-bit can be cleared
using an interworking instruction
where bit [0] of the written PC is 0.

It can also be cleared
by unstacking from an exception
where the stacked T bit is 0.

Executing an instruction
while the T bit is clear causes
an INVSTATE exception.

https://developer.arm.com/documentation/ddi0337/e/ch02s03s02

Interruptible-Continuable Instruction (ICI) bits



Field	Name	Definition
[31:27]	-	Reserved.
[26:25], [15:10]	ICI	**Interruptible-continuable instruction** bits. When an interrupt occurs during an LDM or STM operation, the multiple operation stops temporarily. The EPSR uses bits [15:12] to store the number of the next register operand in the multiple operation. After servicing the interrupt, the processor returns to the register pointed to by [15:12] and resumes the multiple operation. If the ICI field points to a register that is not in the register list of the instruction, the processor continues with the next register in the list, if any.

https://developer.arm.com/documentation/ddi0337/e/ch02s03s02

If-Then (IT) block

Field	Name	Definition
[26:25], [15:10]	IT	If-Then bits. These are the execution state bits of the If-Then instruction. They contain the number of instructions in the if-then block and the conditions for their execution.
[24]	T	The T-bit can be cleared using an interworking instruction where bit [0] of the written PC is 0. It can also be cleared by unstacking from an exception where the stacked T bit is 0. Executing an instruction while the T bit is clear causes an INVSTATE exception.
[23:16]	-	Reserved.
[9:0]	-	Reserved.

https://developer.arm.com/documentation/ddi0337/e/ch02s03s02

Interruptible LDM, STM (5)

- 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 (7)

- 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 (8)

- 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 (9)

- 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

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