Application Note 211

Interrupt Behaviour of Cortex-M1

Document number: ARM DAI 211A
Issued: 25th June, 2008
Copyright ARM Limited 2008

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

Application Note 211
Interrupt Behaviour of Cortex-M1

Copyright © 2008 ARM Limited. All rights reserved.

Release information

Change history

Date	Issue	Change
June 2008	A	First release

Proprietary notice

Words and logos marked with © and ™ are registered trademarks owned by ARM Limited, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

`http://www.arm.com`

Table of Contents

1. Introduction 1-1

2. Exception Handling 2-1

3. Interrupt Handling 3-1

3.1 Nested Vectored Interrupt Controller (NVIC) 3-1
3.2 Interrupt Registers 3-1
3.3 Interrupt Prioritization 3-2
3.4 Typical Interrupt Entry/Exit 3-2
3.5 Pre-emption 3-2
3.6 Stacking 3-3
3.7 Return (Un-stacking) 3-3
3.8 Late-arriving 3-3
3.9 Pre-emption examples 3-3

4. Interrupt Latency Examples 4-1

4.1 Hardware Configuration 4-1
4.2 Software Configuration 4-2
4.3 Examples 4-4

5. References 5-1

6. Appendix A 6-1

6.1 Startup Code 6-1

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

1. Introduction

The ARM Cortex-M1 processor was developed for the usage with FPGAs (Field Programmable Gate Arrays) and it is targeting low-cost applications in which costs, ease of use and low interrupt latency are critical. Such low interrupt latency applications can include real-time control systems.

The processor implements the Thumb instruction set, with several additional Thumb-2 instructions to enable interrupt handling in Thumb state and upwards compatibility with ARM Cortex-M3. It includes 13 general purpose registers (R0 – R12), a Link Register (LR), a Stack Pointer (SP), a Program Counter (PC) and an xPSR (Program Status Register). The processor has 2 Tightly Coupled Memories (TCMs) interfaces, one for the instruction side (ITCM) and another for the data side (DTCM) of the processor, which could be connected to internal FPGA on-chip RAM blocks. Further the processor contains an AHB-Lite interface which allows the processor to fetch instructions or load/store data as a master in an AHB-Lite system.

This Application Note focuses on the interrupt behaviour of the Cortex-M1 processor, which allows the processor to be used in timing critical applications. It is recommended to use this Application Note in conjunction with the Technical Reference Manual (TRM) for Cortex-M1, as it describes all parts of the processor and gives the reader a good understanding of the complete processor and a base for using this Application Note.

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

2. Exception Handling

The processor implements an exception model which defines the following exception types with the associated priority levels:

Table 1 Exception types with associated priorities

Exception Number	Priority Level	Exception type	Vector Address
-	-	SP at reset	0x00
1	-3 (highest)	Reset	0x04
2	-2	Non-Maskable Interrupt (NMI)	0x08
3	-1	HardFault	0x0C
4 – 10	-	Reserved	-
11	Configurable	SuperVisor Call (SVC)	0x2C
12 – 13	-	Reserved	-
14	Configurable	PendSV	0x38
15	Configurable	SysTick	0x3C
16	Configurable	External Interrupt #0 (IRQ[0])	0x40
16 + N	Configurable	External Interrupt #N (IRQ[1])	0x40 + N*0x04
	…	…	
47	Configurable	External Interrupt #31 (IRQ[31])	0xBC

Table 1 show all exceptions which can occur in Cortex-M1 with the associated vector addresses. On an exception event the core reads the address stored in the vector table and branches to this address.

The first 3 exceptions, Reset, NMI and the HardFault have fixed priority levels, where the other exceptions are configurable in the priority.

The Reset is the highest priority exception, and no other exception can be higher priority. It is asserted with the SYSRESETn input of the Cortex-M1 processor and results in resetting the core logic.

The Non-Maskable Interrupt (NMI) is 1 interrupt line which connects to the Nested Vector Interrupt Controller (NVIC) over the NMI input of the processor. It has, after the Reset, the second highest priority and cannot be masked by software (i.e. it cannot be disabled by software).

The HardFault, SVC, PendSV and SysTick are internal processor exceptions and are not discussed in this paper. The SVC, PendSV and SysTick exceptions are only used when the OS Extensions are implemented. Please have a look in the TRM for further description of these exceptions.

The External Interrupts (#0 – #31) are up to 32 interrupt lines which are directly connected to the IRQ[0] – IRQ[31] inputs to the NVIC, described later in this document.

This paper describes the interrupt behaviour of Cortex-M1, so it shows the behaviour of the core on a NMI and/or External Interrupt exception.

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

3. Interrupt Handling

A typical system requires the processor to react to external events. Therefore Cortex-M1 provides 1 – 32 prioritizable, maskable interrupt lines (IRQ[0] – IRQ[31]), as well as one non-maskable interrupt line (NMI).

3.1 Nested Vectored Interrupt Controller (NVIC)

The NVIC is part of the Cortex-M1 processor and is tightly coupled to the core. Figure 1 shows the processor with the core and the NVIC with the associated interrupt lines.

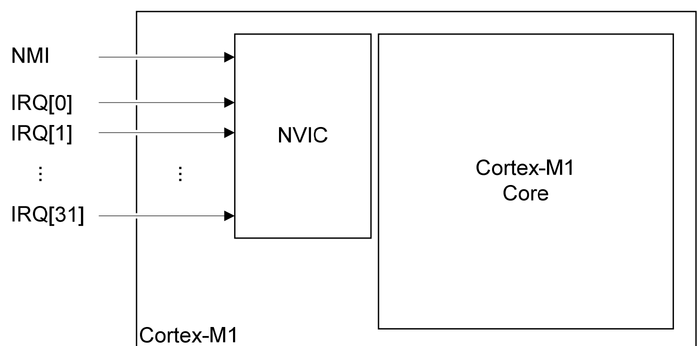


Figure 1 Cortex-M1 processor with the core and NVIC

The NVIC handles all exception of Cortex-M1 (also internal exceptions) and is able to prioritize all sources dependent on their priorities (see Table 1). Further it deals with late arriving interrupts (Interrupts with higher priority than the actual processed one) very efficiently.

Which interrupts are processed and which priorities are linked to these interrupts can be controlled with the Interrupt Registers.

3.2 Interrupt Registers

The NVIC has memory mapped registers which can be accessed by software running on the core. The registers are 32-bit wide, each bit represents one interrupt input, and are used for controlling the interrupts:

- Set-enable and clear-enable registers
- Set-pending and clear-pending registers
- Priority registers

The set-enable (SETENA) and clear-enable (CLRENA) registers are used to enable the interrupts. The set-pending (SETPEND) and clear-pending (CLRPEND) registers are used to handle incoming interrupts.

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

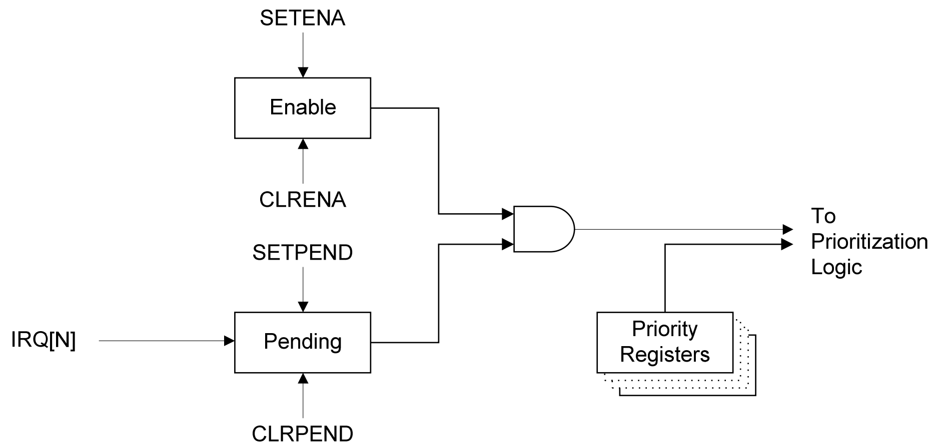


Figure 2 Cortex-M1 Interrupt registers

Figure 2 shows how the interrupt registers works together. The pending register can also be set by software. This can be very useful for debugging interrupt handlers. Please note that the pending register can also be accessed by the core directly, when exiting from an exception handler.

If the appropriate interrupt is enabled and pending, it is forwarded and checked against the priorities, which are stored in 8 priority registers, in the prioritization logic of the NVIC.

3.3 Interrupt Prioritization

The 8 priority registers contains priorities for 4 interrupts each where each priority uses 2 bits, which gives 4 priority levels. Lower numbers in the priority levels are higher priority, as well as lower hardware interrupt number are higher priority if the priority levels are the same.

3.4 Typical Interrupt Entry/Exit

Cortex-M1 has a “micro-coded” interrupt mechanism which automatically saves/restore the state of the core on an interrupt entry or return. This context saving is compliant to the ARM Architecture Procedure Calling Standard (AAPCS).
Using this micro-coded mechanism allows using interrupts with no instruction overhead and it allows Interrupt Service Routines (ISRs) to be written entirely in C.

3.5 Pre-emption

When an exception arrives, the core automatically performs the following steps:

- Stacking (Except on a Reset)
- Read the appropriate vector address from the vector table
- The SP is updated with the first entry of the vector table (Only on a Reset)
- The LR is updated
- The PC is updated with the vector address from the vector table

The core is fetching then the first instruction from the exception vector. It takes 3 cycles from the fetching of this instruction until it is executed (because of the 3 stage pipeline of the core).

Pre-emption would take place on an arriving exception if the priority is higher than the actual instruction stream. The following figures show typical pre-emption examples with one or two interrupts (with different priority, or different hardware interrupt number when configured with the same priority).

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

3.6 Stacking

Cortex-M1 uses a full-descending stack which is used to save the context of the core on an exception entry. The stack can be stored either the DTCM or the external AHB-Lite interface; this is dependent where the programmer has located the SP at reset (See Table 1, first entry). The core automatically restores the stack on an exception exit (See section 3.7).

Figure 3 shows the progress of stacking (assumes the address and data are presented at the same time, as this would be the case for TCMs), the stack in the memory, with the old/new Stack Pointer (SP) and the saved registers.

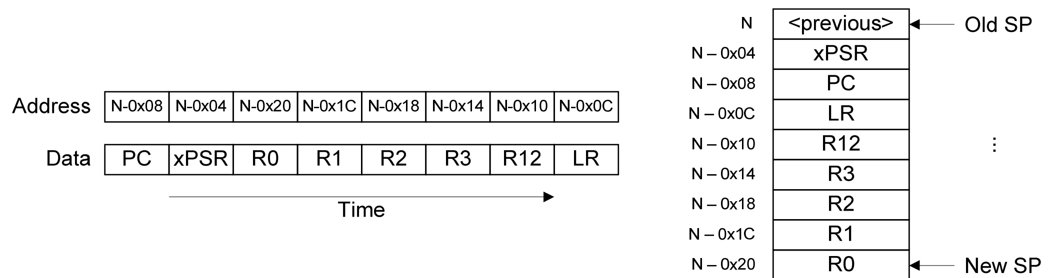


Figure 3 Stacking of the main stack of Cortex-M1

It shows the order of the storage in memory: PC, xPSR, R0, R1, R2, R3, R12 and LR (This order follows the AAPCS).

3.7 Return (Un-stacking)

A return from an exception is caused by one of two instructions: a POP to the PC or BX to any register instruction in the exception handler. The core automatically restores the information stored on the active stack (un-stacking), and updates the core registers with this information (Please note: The programmer has to save/restore other registers as mentioned in section 3.6 separately in the exception handler).

Please see the ARMv6-M Architecture Reference Manual for more information about these instructions.

3.8 Late-arriving

The NVIC has special features which deal with the case when an exception with higher priority or an exception with same priority but lower exception number arrives, when the pre-emption has been started already.

If this is the case, the core has the possibility to handle the new, higher priority, exception. This decision can be made until the point when the PC is updated with the vector address from the vector table.

3.9 Pre-emption examples

The following figures show typical examples how interrupts are handled, where the horizontal axis shows the time, and the vertical axis the priority associated to each interrupt. Each interrupt is marked appropriately.

The first example is one of the most obvious one. The running instruction stream is interrupted by an arriving interrupt:

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

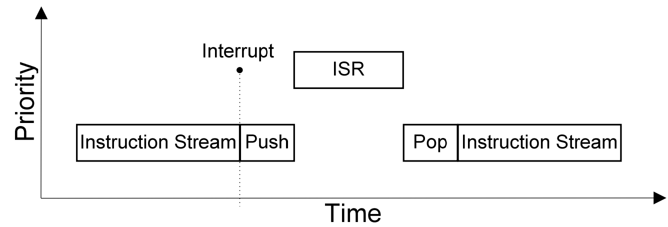


Figure 4 Pre-emption example with one interrupt

The next example shows what happens when 2 interrupts occur, the first when running normal in the instruction stream, and the second while handling the first interrupt:

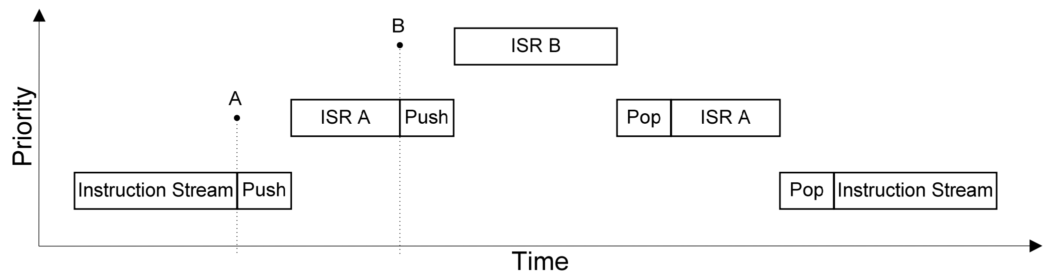


Figure 5 Pre-emption of 2 interrupts while servicing the first interrupt

Figure 5 shows that the instruction stream is interrupted by interrupt A, which results in execution of the ISR A. While ISR A is executed a second, higher priority interrupt, comes along and pre-empt the ISR A to run ISR B. This pre-emption is connected again to a stack push, which saves the context for the ISR A (Please note: With Cortex-M1 the programmer has no programming overhead for supporting nested interrupts besides the programming of the priority registers when needed).

The previous example showed how 2 interrupts are handled if they occur while another handling code is executed. The following example shows how "late arrival" interrupts are handled by the core (See section 3.8).

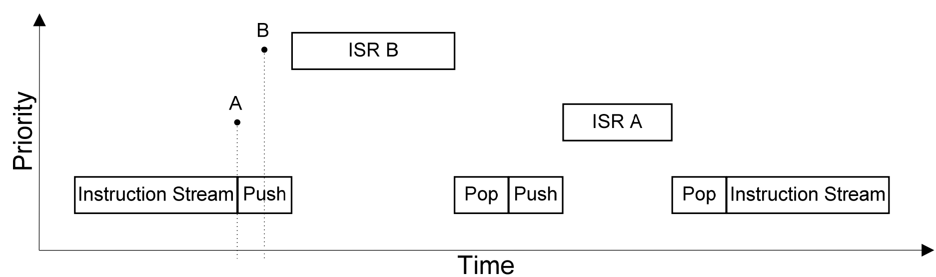


Figure 6 Pre-emption of 2 interrupts with one late-arriving interrupt

Figure 6 shows how Interrupt A is handled by the hardware in the typical way, but in this example a second interrupt, with higher priority, comes in and prevents the execution of ISR A completely. This can happen because Interrupt B happened before the PC is updated with the vector address. The Interrupt A is still pending, and ISR A is therefore executed after Interrupt B is service correctly with the ISR B.

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

4. Interrupt Latency Examples

The previous chapters introduced the main features which are responsible for the very good interrupt performance of Cortex-M1. The term “Interrupt latency” is usually the time which is needed from the arrival of an interrupt until the first instruction of the appropriate ISR is executed.

This chapter focuses on the interrupt latency of Cortex-M1. It shows the interrupt behaviour of the processor using a typical hardware/software configuration.

4.1 Hardware Configuration

The system uses a Cortex-M1 with 2 interrupt lines, the NMI and IRQ[0]. These interrupts are connected to an Interrupt Generation block which asserts the interrupt inputs. The core uses 2 TCM memories, ITCM and DTCM, and a clock/reset generation block.

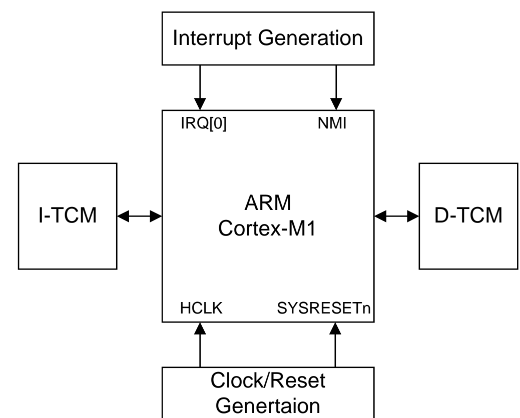


Figure 7 Hardware configuration of the testbench

Figure 7 shows the example hardware configuration. It uses the ITCM as instruction memory and the DTCM as data memory. The ITCM is preloaded with the application code, where the DTCM is not initialized.
Please note: The system doesn’t use the external AHB-Lite interface.

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

4.1.1 Memory Map

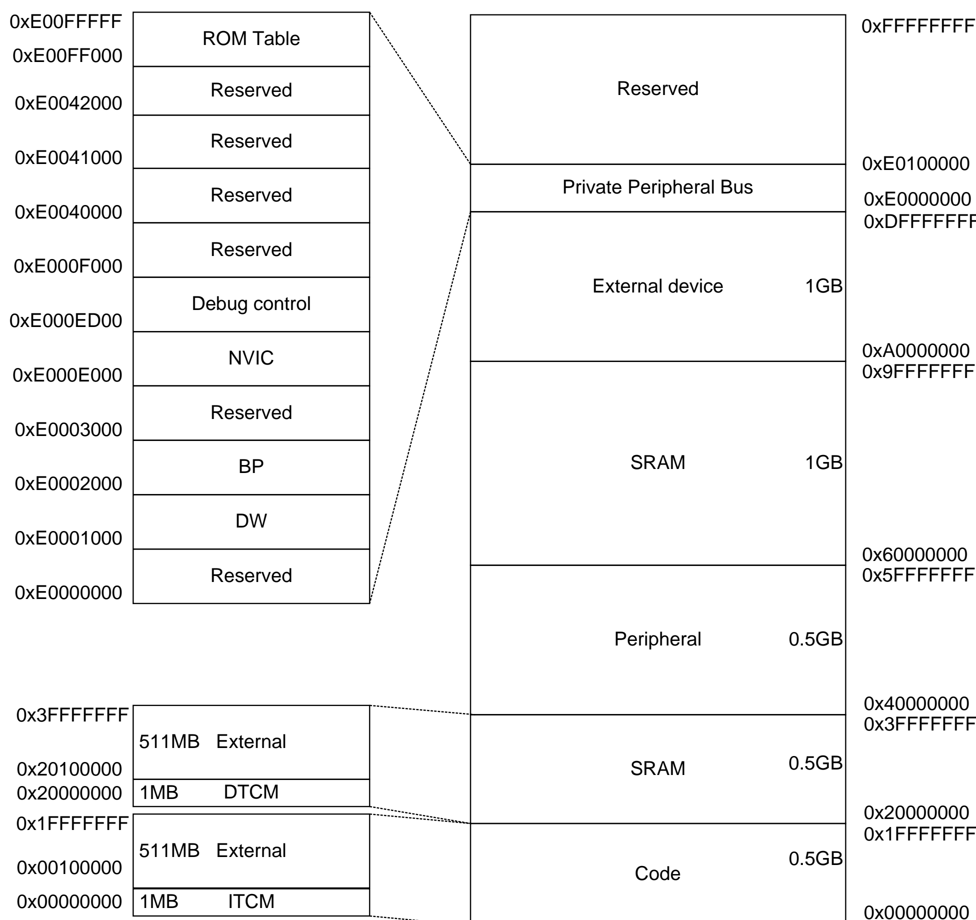


Figure 8 Cortex-M1 Memory Map

Figure 8 shows the memory map of the Cortex-M1 core. The core boots from the ITCM and stores data, e.g. the stack, in the DTCM. The NVIC can be accessed on the Private Peripheral Bus. Please refer to the TRM of Cortex-M1 for a description of the rest of the map.

4.2 Software Configuration

The exception handling of Cortex-M1 allows the programmer to program the processor completely in C. Nevertheless this software example uses Thumb-2 assembler code for the start-up code because it provides better visibility of the interrupt functionality.
The code (Start-up and main code) runs out of a preloaded ITCM.

4.2.1 Start-up Code

The start-up code provides the vector table, the exception handlers and the branch to the main code. Appendix A shows the start-up code.

The Vector Table

Table 2 Vector table for the example system

Exception type	Vector Address	Vector Data

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

SP at reset	0x00	0x20000200
Reset	0x04	Reset_Handler
Non-Maskable Interrupt (NMI)	0x08	NMI_Handler
HardFault	0x0C	HardFault_Handler
…	-	-
External Interrupt #0 (IRQ[0])	0x40	IRQ0_Handler

The vector table shown in Table 2 defines constant addresses for the handlers, which are calculated at assembly time. The first data value defines the SP which points to the top of the stack - This example allocates 200 bytes in the DTCM.

The Handler

Reset_Handler – The reset handler initializes all used registers, enable the used IRQ lines of the NVIC and branch to the main application code.

NMI_Handler – The NMI handler sets the main counter used in the main code to the value 0x1.

HardFault_Handler – The hard fault handler is a forever loop which has no functionality.

IRQ0_Handler – The IRQ0 handler sets the main counter used in the main code to the value 0x2.

4.2.2 Main Code

The main application is a 32-bit upcounter, which is initialized to 0x0 and counts forever in a while loop:

```
void __main (void)
{
  unsigned int n = 0;
  /* The main upcounter */
  while (1)
  {
    n += 1;
  }
}
```

The value of the upcounter (Integer value of n) is set to a specific value, dependent on which external exception occurred and which handler is executed by the core (0x1 on NMI and 0x2 on IRQ0).

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

4.3 Examples

4.3.1 Stack push on a NMI

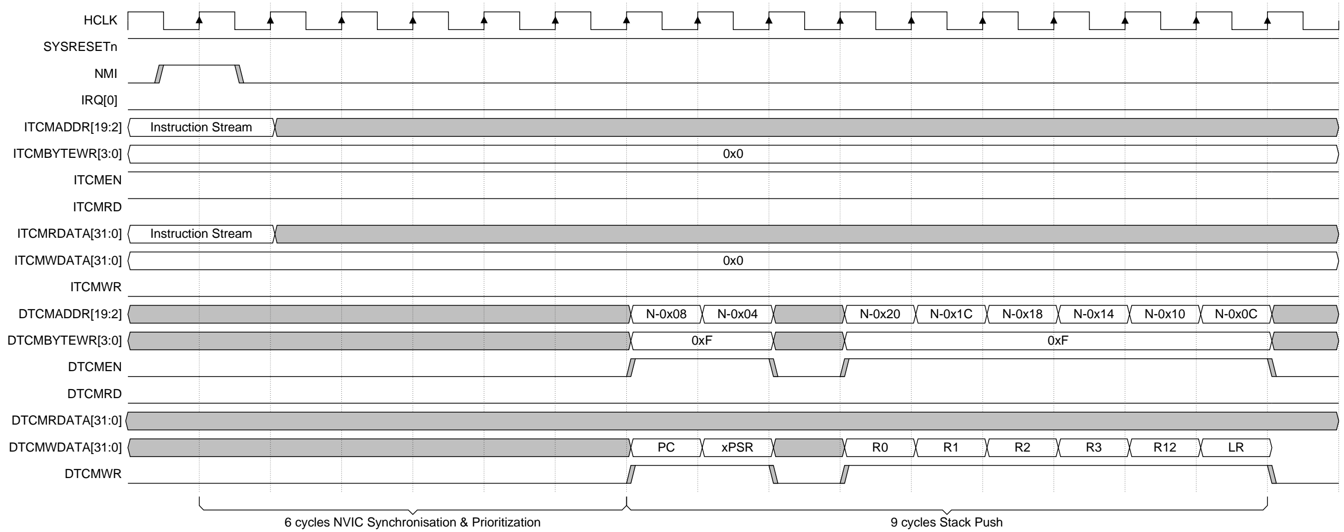


Figure 9 Example of a stack push on a NMI

On a NMI the core starts pushing the current context on the stack after 6 clock cycles. The stack push itself takes 9 cycles. This gives a total of 15 clock cycles for completing the stack operation on a NMI.

4.3.2 Stack push on an IRQ

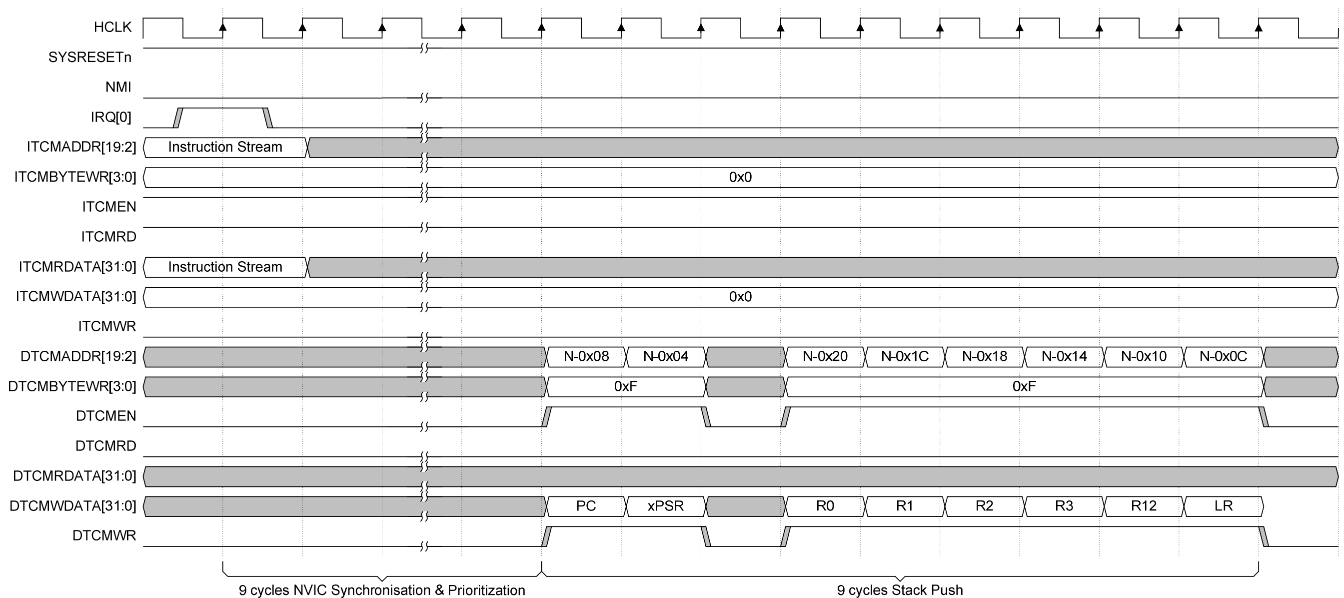


Figure 10 Example of a stack push on an IRQ

Compared to the NMI, an IRQ takes 3 cycles more to start pushing the context onto the stack, so it takes 9 cycles.

 Copyright © 2008 ARM Limited. All rights reserved.
Open Access

4.3.3 Stack pop

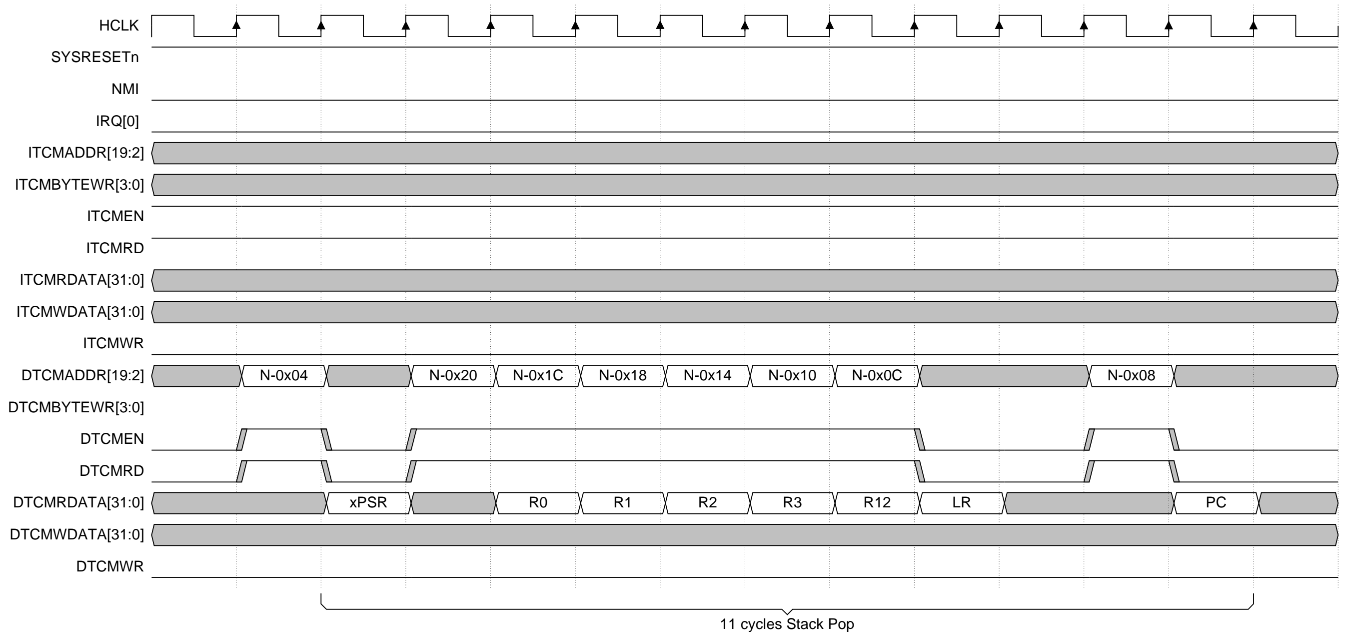
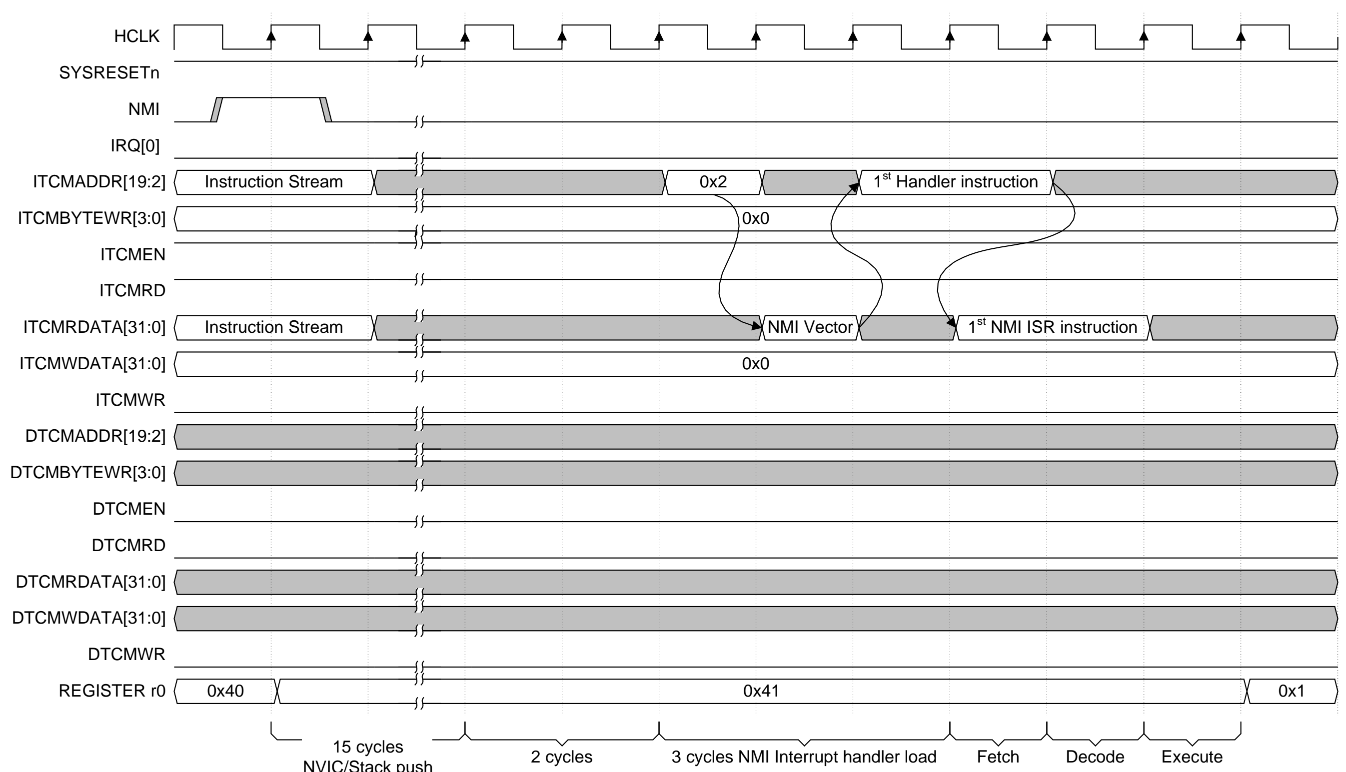


Figure 11 Example of a stack pop on a NMI/IRQ

A stack pop takes 11 clock cycles (until data is on the read bus) where the PC is popped at the end of the operation. Such a pop is similar for the return of a NMI or IRQ.

4.3.4 Example 1: A NMI interrupt arrives at Cortex-M1

This first example shows the behaviour of Cortex-M1 when a NMI interrupt arrives and the processor interrupts the instruction stream to handle the interrupt.



Copyright © 2008 ARM Limited. All rights reserved.
Open Access

Figure 12 A NMI interrupt is handled by the NMI handler

As mentioned in section 4.3.1 it takes 15 clock cycles to finish the NVIC synchronisation and the stack push. After that it takes 2 clock cycles to start loading the appropriate vector address. It takes 3 clock cycles to load the first instruction from the exception handler and another 3 cycles to fill up the pipeline and execute this first instruction.
This gives a total of 23 cycles to execute the first instruction of the NMI exception handler. After executing all instructions of the exception handler, a return instruction would initialize a stack pop (See section 4.3.3)

4.3.5 Example 2: An IRQ[0] interrupt arrives at Cortex-M1

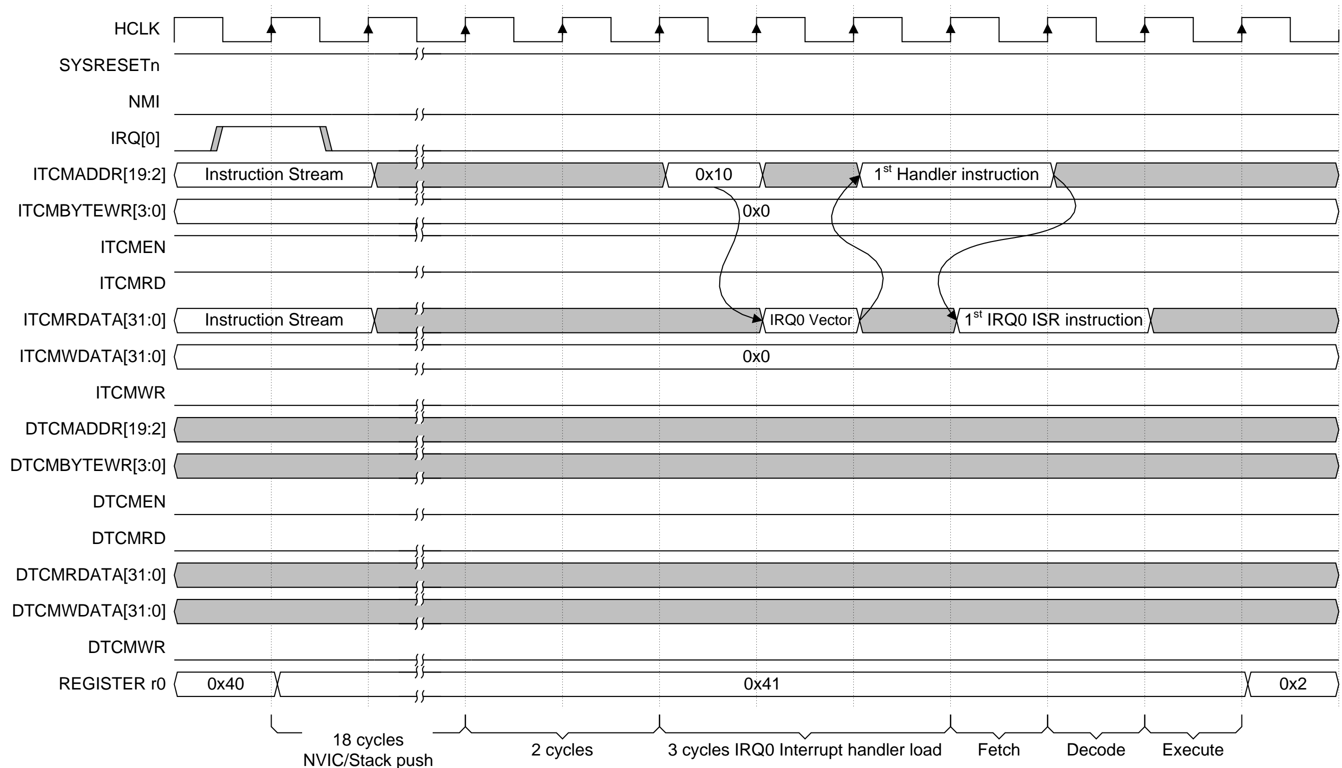


Figure 13 An IRQ[0] interrupt is handled by the IRQ0 handler

Figure 13 shows the execution of the IRQ0 handler, similar to the previous example.

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

5. References

Table 3 Document References

Ref	Author(s)	Title
1	ARM	Cortex-M1 Technical Reference Manual
2	ARM	Application Binary Interface for the ARM Architecture

Copyright © 2008 ARM Limited. All rights reserved.
Open Access

6. Appendix A

This appendix provides the startup code which was used to generate the examples.

6.1 Startup Code

```
                THUMB

; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors

__Vectors       DCD     0x20000200          ; Top of Stack
                DCD     Reset_Handler       ; Reset Handler
                DCD     NMI_Handler         ; NMI Handler
                DCD     HardFault_Handler   ; Hard Fault Handler
                DCD     0                   ; Reserved
                DCD     0                   ; Reserved
                DCD     0                   ; Reserved
                DCD     0                   ; Reserved
                DCD     0                   ; Reserved
                DCD     0                   ; Reserved
                DCD     0                   ; Reserved
                DCD     0                   ; Undefined
                DCD     0                   ; Reserved
                DCD     0                   ; Reserved
                DCD     0                   ; Undefined
                DCD     0                   ; Undefined
                DCD     IRQ0_Handler        ; IRQ0 Handler
                AREA    |.text|, CODE, READONLY

; Exception Handlers
Reset_Handler   PROC
                EXPORT  Reset_Handler

                ;Initialize used registers
                MOVS r0,#0x0
                MOV r1,r0
                MOV r2,r0
                MOV r3,r0
                MOV r12,r0

                ;Enable interrupt for IRQ[0]
                LDR  r0,=0xE000E100 ; SETEN register
                MOVS r1,#0x1        ; IRQ[0]
                STR  r1,[r0]        ; enable IRQ

                IMPORT  __main
                LDR     R0, =__main
                BX      R0
                ENDP

NMI_Handler     PROC
                MOVS r0,#0x1
                BX LR              ; return from NMI exception
                ENDP

HardFault_Handler\
                PROC
                EXPORT  HardFault_Handler
                B       .          ; loop forever
```


Copyright © 2008 ARM Limited. All rights reserved.
Open Access

```
                ENDP

IRQ0_Handler    PROC
                MOVS r0,#0x2
                BX LR              ; return from IRQ[0] exception
                ENDP

                ALIGN

                END
```


Copyright © 2008 ARM Limited. All rights reserved.
Open Access