Cortex-M4(F) Lazy Stacking and Context Switching

Application Note 298

Released on: 16 March 2012

ARM®

ARM DAI0298A (ID032612)

Cortex-M4(F) Lazy Stacking and Context Switching
Application Note 298

Release Information

Table 1 Change history

Date	Issue	Confidentiality	Change
16 March 2012	A	Non-Confidential	First release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

1 Introduction

The Cortex-M4 processor is based on the ARMv7E-M architecture and is similar to the Cortex-M3. It has an integer register bank for general data processing, which is the same as the one in the ARM Cortex-M3, and a stack based exception model.

The Cortex-M4 products are available as:

- Cortex-M4 without *Floating-point Unit* (FPU)
- Cortex-M4F with a FPU.

Comparing the Cortex-M4F to the Cortex-M4 or Cortex-M3, the Cortex-M4F includes an additional floating-point register bank ranging from S0 – S31, and several other registers of which most are memory mapped:

Floating-point Status and Control Register **(FPSCR)**

This is a special register in the FPU and it is not memory mapped. It contains control bit fields for controlling floating-point operations and a number of status bit fields that indicate the status of the FPU. The VMSR (*Move to floating-point System Register from ARM Core Register*) and VMRS (*Move to ARM Core Register from floating-point System Register*) instructions are used to transfer data between the FPSCR and general purpose registers in the integer register bank.

Coprocessor Access Control Register **(CPACR)**

This register is located at address `0xE000ED88`. To enable the FPU, bits[23:20] of the CPACR, which are the CP10 and CP11 bit fields, must be set to `0xF`. By default, the floating-point unit is disabled at RESET.

Floating-point Context Control Register **(FPCCR)**

This register is located at address `0xE000EF34`. This register controls the context saving behavior. By default, it enables the lazy stacking behavior, see *Lazy stacking feature* on page 7.

Floating-point Context Address Register **(FPCAR)**

This register is located at address `0xE000EF38`. This register holds the address location of the reserved space allocated in an exception stack frame for floating-point registers which have not yet been saved to the stack after an exception entry.

Note

The FPCAR register points to a section of stack space within the current stack. The address value in the FPCAR is automatically generated by the hardware of the processor.

Floating-point Default Status Control Register **(FPDSCR)**

This register is located at `0xE000EF3C`. This register holds the default values of FPSCR.

In addition:

- The CONTROL Register, a special register, that is also available on the Cortex-M3, has an additional bit field to support the FPU feature. Bit[2] of the CONTROL Register is defined as *Floating-Point Context Active* (FPCA). This bit is set automatically when the FPU is used, and cleared when a new context is started, for example when starting of an

Interrupt Service Routine (ISR). The FPCA bit is available on the Cortex-M4F only. The CONTROL Register is not memory mapped. You can access this register by using MSR and MRS instructions.

- The EXC_RETURN code value which is used in the exception mechanism has an additional bit field to define floating-point stack status. When the value of EXC_RETURN[4] is 0, it indicates that the exception stack frame for an exception return contains floating-point registers. If this bit is 1, then it means the exception stack frame does not contain floating-point registers.

  Note

  EXC_RETURN (Exception Return) is a code value generated automatically by the Cortex-M processors when entering an exception handler. The value is stored in the *Link Register* (LR) and is used at exception return. Several bits of this code are used to store information about the status of the processor before the exception, for example, which stack pointer was being used.

- The exception stack frame has an additional format type shown in Figure 1 on page 5 which permits automatic saving of registers, S0-S15 and FPSCR, in the FPU. This is in addition to registers that are saved in the Cortex-M3 stack frame consisting of R0-R3, R12, LR, PC, spacer. The original stack frame format type for Cortex-M3 is also used when the stacking of floating-point registers is not required.

With the requirements outlined by the *Procedure Call Standard for the ARM Architecture* (AAPCS), a C function must preserve those registers in S16-S31 only when they are used during the execution of the function. The other registers in the FPU, that is, S0-S15 and FPSCR, are always saved automatically. These registers can also be modified by a C function.

To permit an interrupt handler to be written as a normal C function, the processor is required to automatically save the S0-S15 registers and FPSCR on the stack when the current contents of the floating-point registers might be required later.

- If the FPU has been used, indicated by 1 in the FPCA bit in the CONTROL Register, the processor has to automatically save S0-S15 and FPSCR on the stack, in addition to the R0-R3, R12, LR, PC and the spacer registers, which are already saved in an existing Cortex-M stack frame. See the stack frame, on the right side of Figure 1 on page 5 for more information.
- If the FPU has not been used, indicated by 0 in the FPCA bit in the CONTROL Register, only the R0-R3, R12, LR, PC and spacer registers have to be saved. See the stack frame, on the left side of Figure 1 on page 5 for more information.

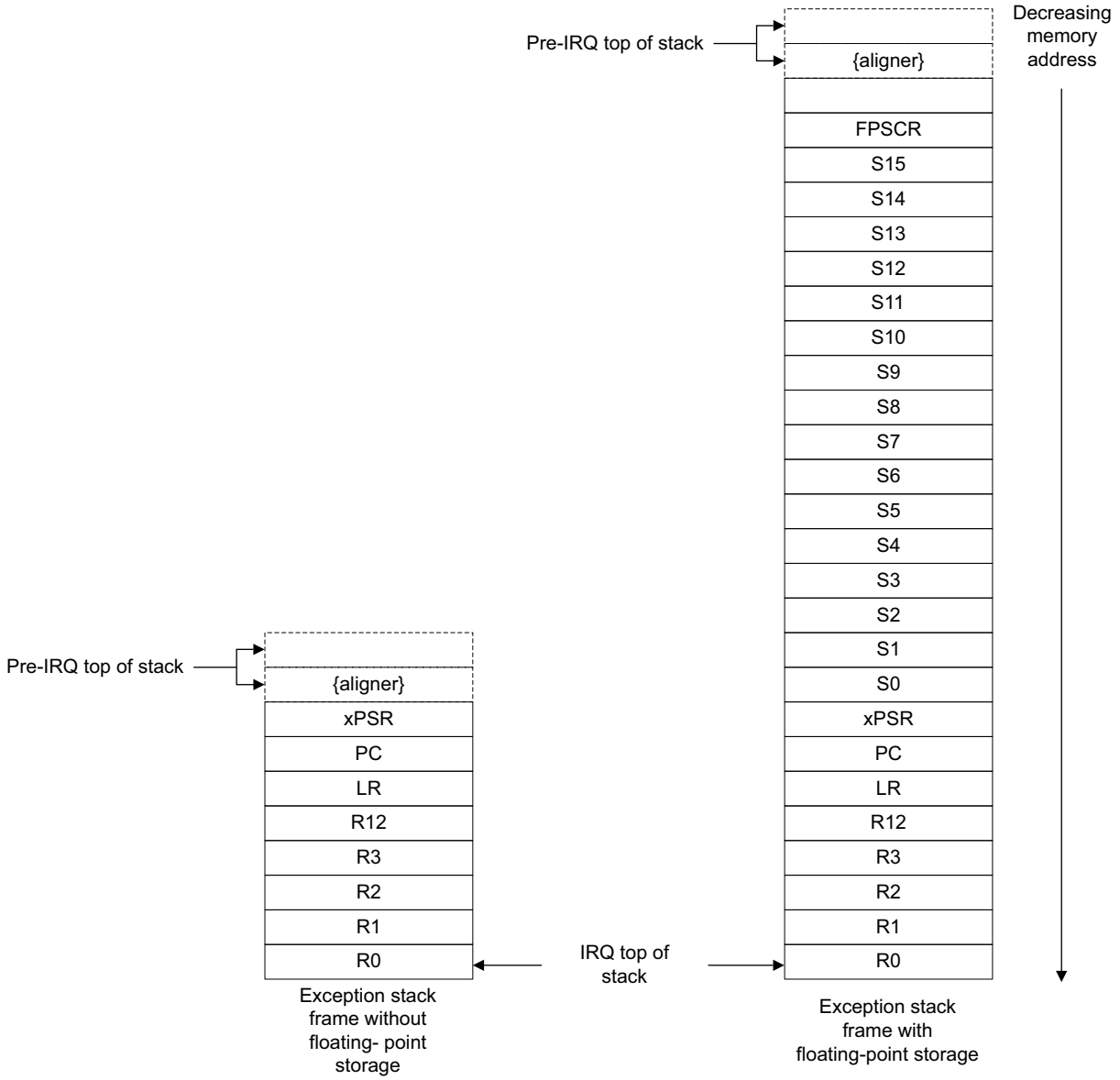

Figure 1 Exception stack frame without and with floating-point register storage

The stack frame type being used is determined by hardware automatically based on settings in the FPCCR and whether the FPU has already been used in the current context, as indicated by FPCA bit in the CONTROL special register. If the FPCA is set and the automatic state saving feature has been enabled, the exception stack frame with floating-point storage is then used. This is because the register values in the FPU might be required by the current context later, after the interrupt handling is completed.

The stacking of floating-point registers has the following effects:

- increases stack frame size
- potentially increases interrupt latency in interrupt processing
- increases context switching time in an embedded *Operating System* (OS).

Because the presence of the two possible stack frame formats, C functions with arguments passed on the stack have to take the value of EXC_RETURN into account when extracting the arguments from the stack.

For interrupt handling in applications without an OS, the automatic hardware state preservation is sufficient and is easy to use. You can write interrupt handlers as normal C functions, and the automatic stacking mechanism handles the required floating-point register stacking and unshackling if required.

For developers of an embedded OS, the situation is more complex. To permit a multi-tasking system to use the FPU in multiple tasks, you must update an OS or *Real-Time Operating System* (RTOS) to handle context saving of the extra registers, S0-S31, and FPSCR, contained in the FPU. During context switching, the OS must:

1. Determine whether an application task has used the FPU, using bit[4] of EXC_RETURN.
2. Save the floating-point context if required.
3. Restore the floating-point context for the next task if required.
4. Switch to the next task using an exception return, with the EXC_RETURN code value matching the stack frame type.

The automatic stacking mechanism handles only S0-S15 registers and the FPSCR. The OS has to handle the saving and restoration of S16 to S31 registers manually.

2 Lazy stacking feature

The Cortex-M4F adds a feature called lazy stacking. This feature avoids an increase of interrupt latency by skipping the stacking of floating-point registers, if not required, that is:

- if the interrupt handler does not use the FPU, or
- if the interrupted program does not use the FPU.

If the interrupt handler has to use the FPU and the interrupted context has also previously used by the FPU, then the stacking of floating-point registers takes place at the point in the program where the interrupt handler first uses the FPU.

The lazy stacking feature is programmable. By default this is turned ON. The controlling of lazy stacking is handled by the FPCCR. OS developers have to consider the effects of lazy stacking when developing the context switching code.

If lazy stacking is to be enabled, then bit[31] in the FPCCR, named ASPEN, indicating *always save enable*, and bit[30] named LSPEN, indicating *lazy save enable* must both be set to 1. This is the default value. You can disable lazy stacking by setting LSPEN to 0.

When an application has previously used the FPU, indicated by bit[2] of the CONTROL Register, FPCA is automatically set to 1. If an interrupt occurs, and if the lazy stacking feature is turned ON, the processor reserves extra space in the stack frame for the S0-S15 registers and FPSCR. However, the actual stacking of these registers does not take place, and bit[0] of FPCCR, *Lazy State Preservation Active* (LSPACT) is set to 1 to indicate this. Bit[4] of the EXC_RETURN value, generated at the exception entry is set to 0 to indicate that the exception stack frame contains stack space for floating-point registers, although the actual register contents is not present.

- If the interrupt handler does not use the FPU, then LSPACT stays HIGH until the end of the interrupt. When returning from the interrupt, the processor hardware detects that bit[4] of the EXC_RETURN is 0 and LSPACT is 1 indicating that the stack frame contains space for floating-point registers, but they were not pushed onto the stack so the unshackling of the floating-point registers is ignored.
- If the interrupt handler uses the FPU at some stage, the processor is stalled when the first floating-point instruction take place, while the floating-point registers, that is, S0-S15 registers and FPSCR, are pushed to the stack, and LSPACT is cleared. The program execution then continues. At the end of the interrupt handler, the processor hardware detects that EXC_RETURN[4] is 0 and LSPACT is 0, indicating that the stack frame contains pushed floating-point registers and unstacks them accordingly.

If lazy stacking is disabled, the processor always pushes the floating-point registers to the stack at exception entry and unstacks them at interrupt return. This increases interrupt latency.

You can use the following setup shown in Table 2 depending on the application:

Table 2 FPCCR setup procedure

LSPEN	ASPEN	Scenarios
0	0	No automatic state preservation. You can use this setting: 1. In applications without an embedded OS or multi-task scheduler, if none of the interrupt or exception handlers use the FPU. 2. In application code where only one exception handler uses the FPU. If multiple interrupt handlers use the FPU, they must not be permitted to be nested. This can be done by setting them to the same priority level.
0	1	Lazy stacking disabled, automatic state saving enabled. CONTROL.FPCA is automatically set to 1 when floating-point is used. At the exception entry, the floating-point registers S0-S15, and FPSCR are pushed to the stack if CONTROL. FPCA is 1.
1	1	Lazy stacking enabled, automatic state saving enabled. CONTROL.FPCA is automatically set to 1 when floating-point is used. If CONTROL.FPCA is 1 at the exception entry, the processor reserves space in the stack frame and sets LSPACT to 1. But the actual stacking does not happen unless the interrupt handler uses the FPU.
1	0	Invalid configuration.

3 Example lazy stacking scenarios

The follow diagrams show a few scenarios for Cortex-M4F exception entry and exception returns with lazy stacking enabled.

Figure 2 shows the case in which there is no floating-point operation in the interrupted main program since reset, and no floating-point operation in the ISR.

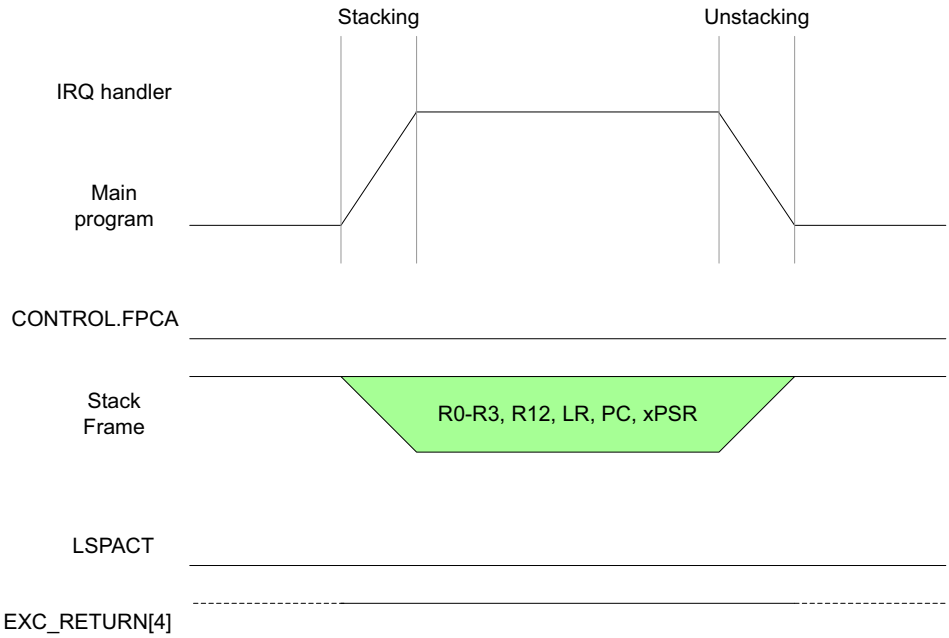

Figure 2 Exception handling with no floating-point operation in the interrupted program

If there is no previous floating-point operation in the interrupted main program, then FPCA stays LOW. In this situation, the exception stack frame is the same as the Cortex-M3 processor or a Cortex-M4 processor, without the FPU. EXC_RETURN[4] is 1 in Figure 2 to indicate that the stack frame does not have floating-point contents.

In Figure 3, there has been a floating-point operation in the interrupted main program since reset, and there is no floating-point operation in the ISR.

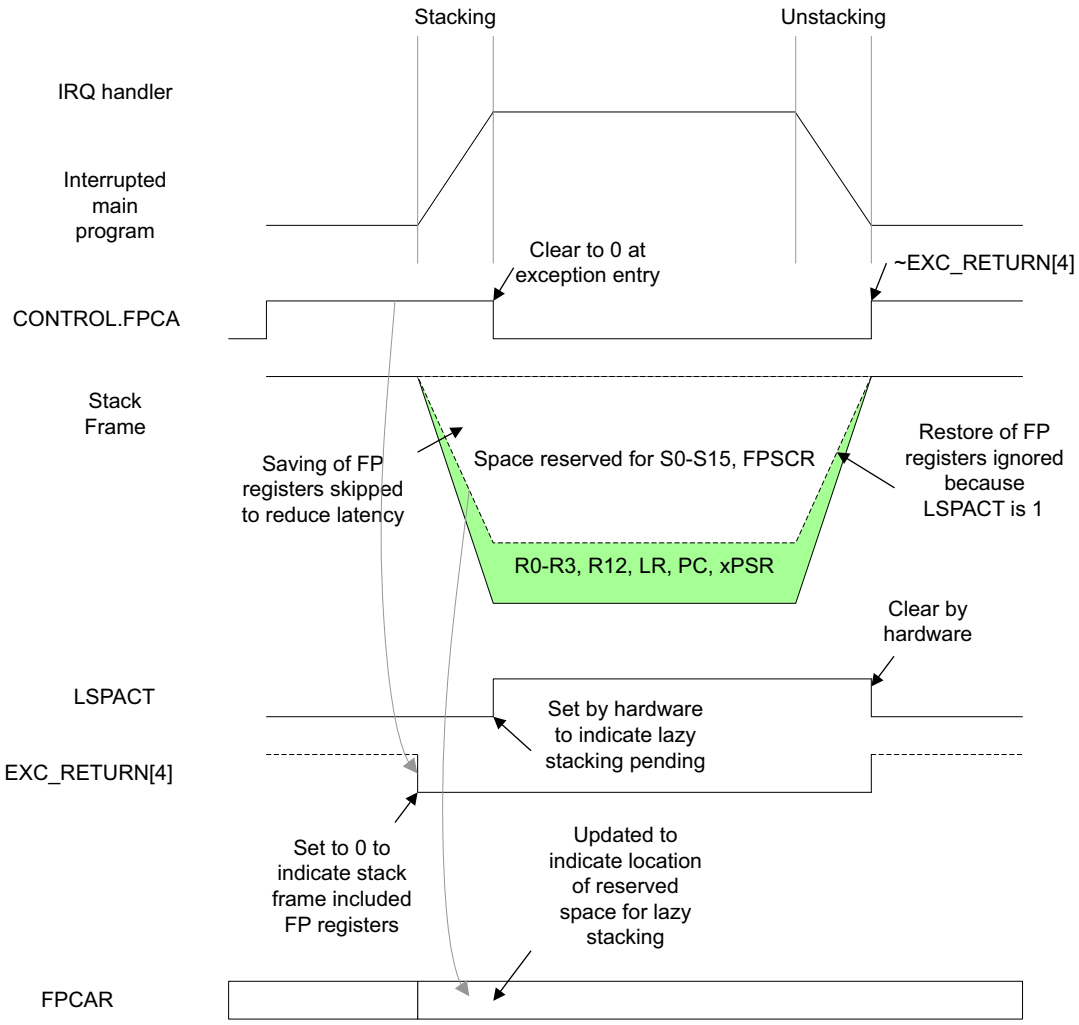

Figure 3 Exception handling with floating-point operation in the interrupted program

In this case, space for the floating-point registers is reserved in the stack frame. These registers are not pushed to the stack because lazy stacking is enabled by default.

Figure 4 shows a floating-point operation in the interrupted main program, and a floating-point operation in the ISR.

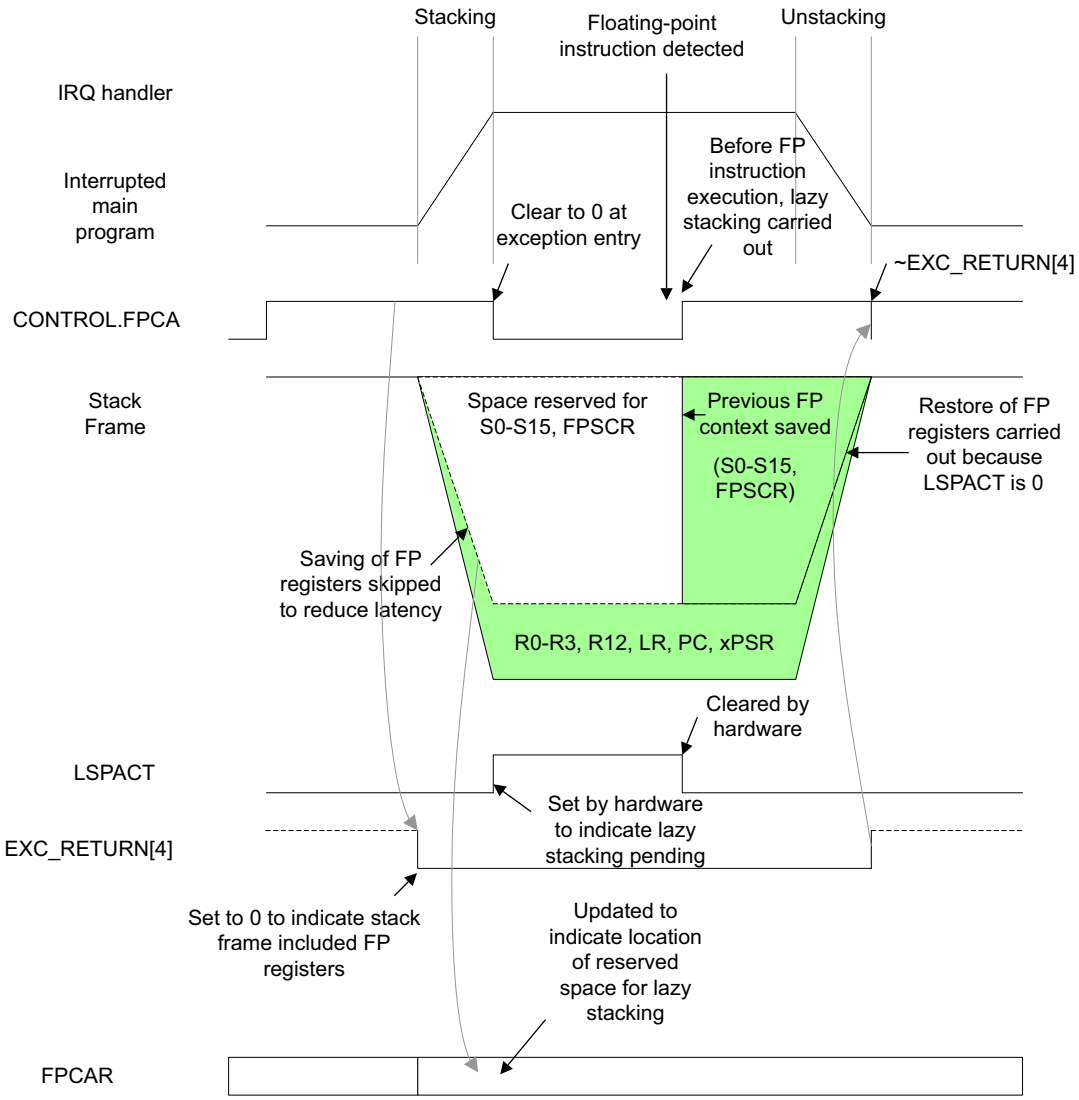

Figure 4 Exception handling with floating-point operations in the interrupted program and ISR

If a floating-point operation is carried out during the execution of ISR, the floating-point registers are saved to the reserved space using lazy stacking, indicated by the FPCAR. To do this, the processor stalls to permit lazy stacking to take place, and continues to execute the floating-point operation after the lazy stacking is complete, see Figure 4.

Note

The reserved space does not include registers S16 to S31. These registers are not saved by the hardware. An AAPCS compliant C compiler must preserve them across subroutine calls.

Figure 5 on page 12 shows nested interrupts, with floating-point operations in the interrupted main program and in a higher priority interrupt handler.

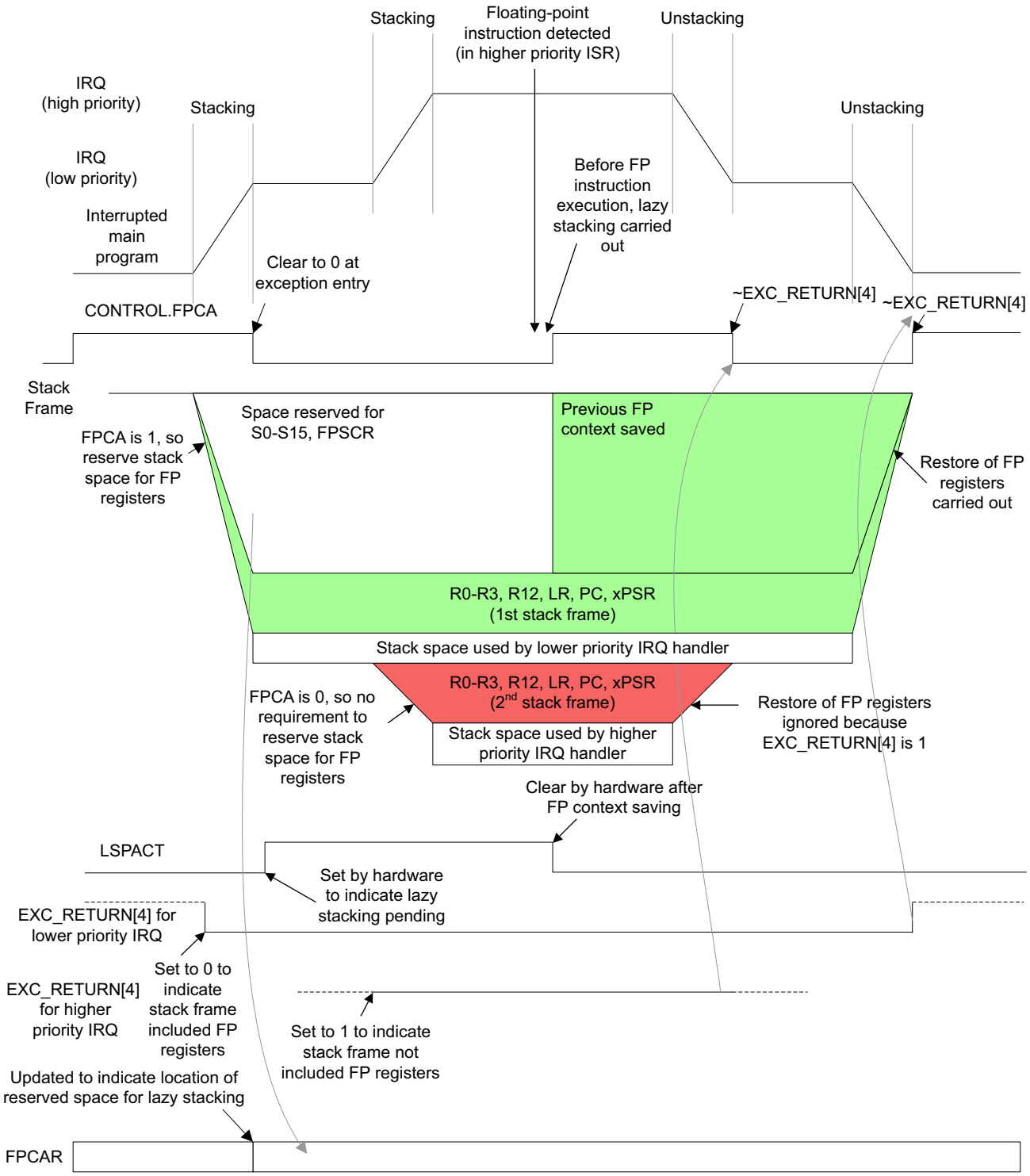

Figure 5 Nested exception handling with floating-point operations in interrupted program and higher priority ISR

If there is no floating-point operation in the first lower priority interrupt handler, then there is no requirement to reserve stack space for the floating-point registers when entering the higher priority interrupt. As a result, the FPCAR is still pointing to the reserved space in the first exception stack frame, see Figure 5.

Figure 6 shows nested interrupts, with floating-point operations in the interrupted main program and both lower and higher priority interrupts:

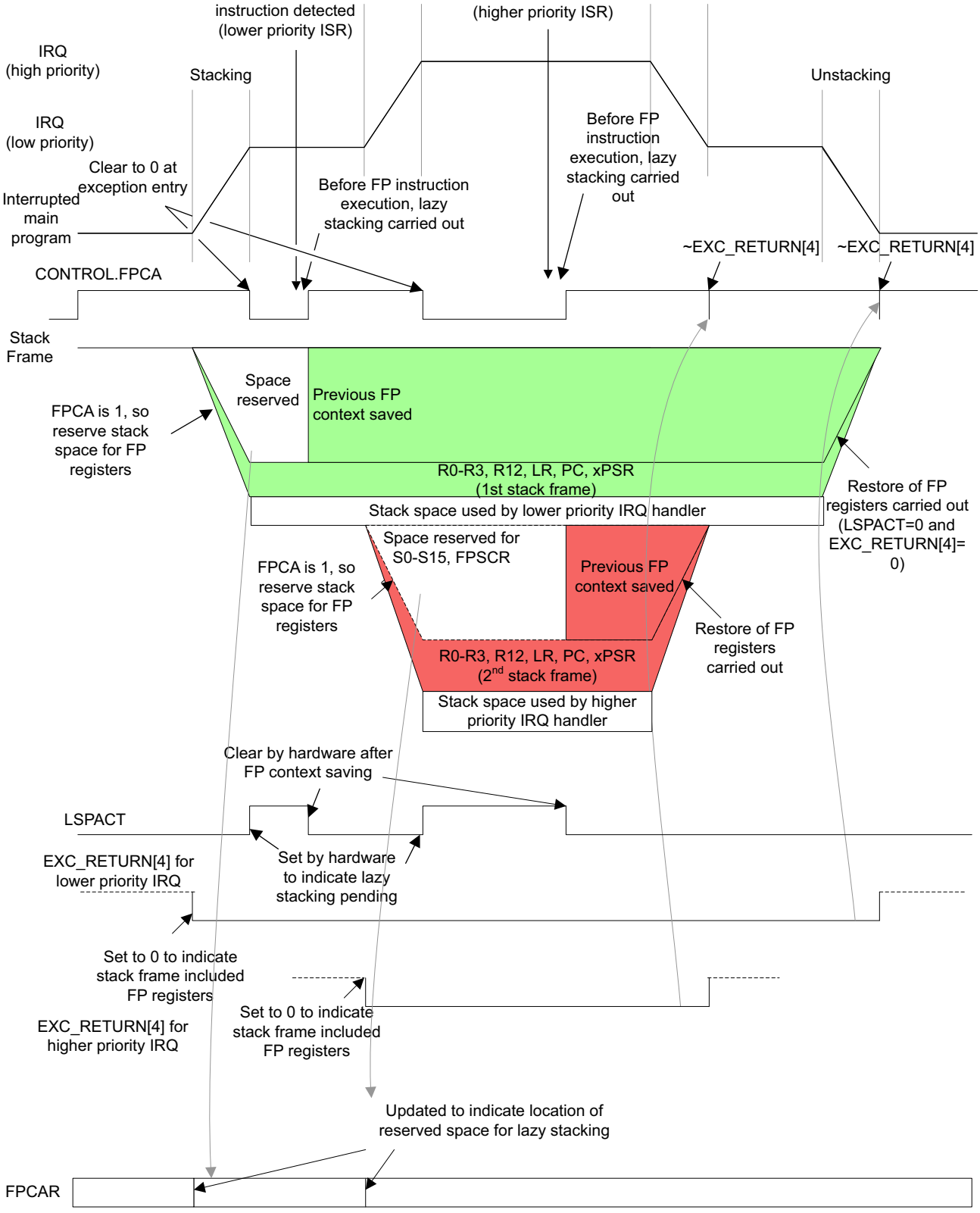

Figure 6 Nested exception handling with floating operations in interrupted programmed ISRs

If both the lower and higher priority interrupt handlers and the interrupted main program use floating-point instructions, then the stack frame for both interrupts require space reserved for the floating-point registers, see Figure 6 on page 13. The FPCAR is updated in both exception entrances.

4 Handling of context switching in a RTOS

When designing a RTOS supporting the Cortex-M4F, it is important that the context switching code is aware of the lazy stacking mechanism.

Based on the floating-point usage of the application tasks, the possible arrangement for floating-point context switching can be divided into three different cases:

- no floating-point operation in any application tasks
- only one application task uses the FPU
- multiple application tasks use the FPU.

4.1 Case 1: No floating-point operation in application tasks

If the application tasks have not used the FPU, as indicated by EXC_RETURN[4]=1, then there is no requirement to handle floating-point context saving, as shown in Figure 7.

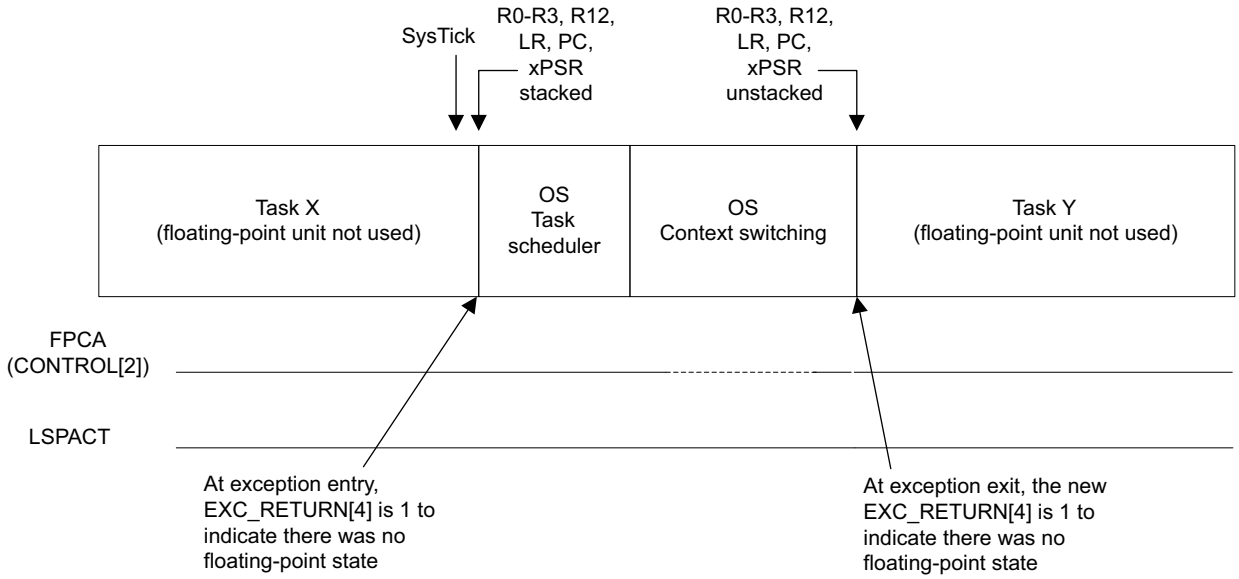

Figure 7 Context switching with no floating-point instructions in tasks

Typically the OS task scheduler is executed regularly, for example, triggered by the SysTick timer. It is common for the context switching operation to be performed in the PendSV exception handler. If context switching is required, the OS might request context switching by setting the pending status bit of the PendSV exception. The PendSV handler runs after the SysTick handler and before the new task context.

Note

- PendSV is a standard exception type in Cortex-M processors. It is an interrupt-driven request for system-level service. In an OS environment, you can use PendSV for context switching when no other exception is active.
- There are alternate methods of handling context switching, depending on the implementation of the particular OS.

When dealing with interrupt handlers:

- If multiple interrupt handlers use the FPU and these interrupts can be nested, then context saving for these interrupt handlers are handled by the lazy stacking mechanism.

- If one interrupt handler uses the FPU, or if multiple interrupt handlers use the FPU but the priority levels of these interrupts are setup so that they are not permitted to be nested, then there is no requirement to preserve floating-point states after the interrupt handlers have completed their tasks. In such cases, you can disable the automatic saving of the floating-point registers.
- If the interrupt handlers do not require the FPU, then you can disable it, and the design works like a Cortex-M4 processor without a FPU.

  Note

  Disabling the FPU might help to reduce power consumption.

ARM recommends that the context switching code within the OS checks for EXC_RETURN bit[4] during context switching to ensure that floating-point operations are not used accidentally. For example, a library object exported into the project might have been compiled with floating-point instructions.

4.2 Case 2: Only one application task uses the FPU

When dealing with tasks for the FPU:

- If it is known that only one application task is going to use the FPU and no interrupt handler is going to use it, then it is possible to omit the floating-point context saving procedure.
- If any interrupt handler uses the FPU in addition to any one of the application tasks, then the automatic saving feature must be enabled by setting ASPEN=1. You can enable the lazy stacking feature by setting LSPEN=1, to permit a shorter interrupt latency if possible. Because both LSPEN and ASPEN are set to 1 by default, the context saving for the interrupt handlers within nested interrupt scenarios, are handled by the lazy stacking mechanism automatically.
- If none of the interrupt handlers are using the FPU, you can set both LSPEN and ASPEN to 0. In this way the stack space could be reduced by not having to reserve space for the floating-point context.

If an OS is being ported from the Cortex-M3 processor, you must update the context switching code to handle bit[4] of the EXC_RETURN value correctly. Also, you must check EXC_RETURN bit[4] in each context switch to ensure that no other task has unexpectedly used the FPU.

4.3 Case 3: Multiple application tasks use the FPU

A simple way to handle context switching is to always store and restore all floating-point registers as shown in Figure 8 on page 17.

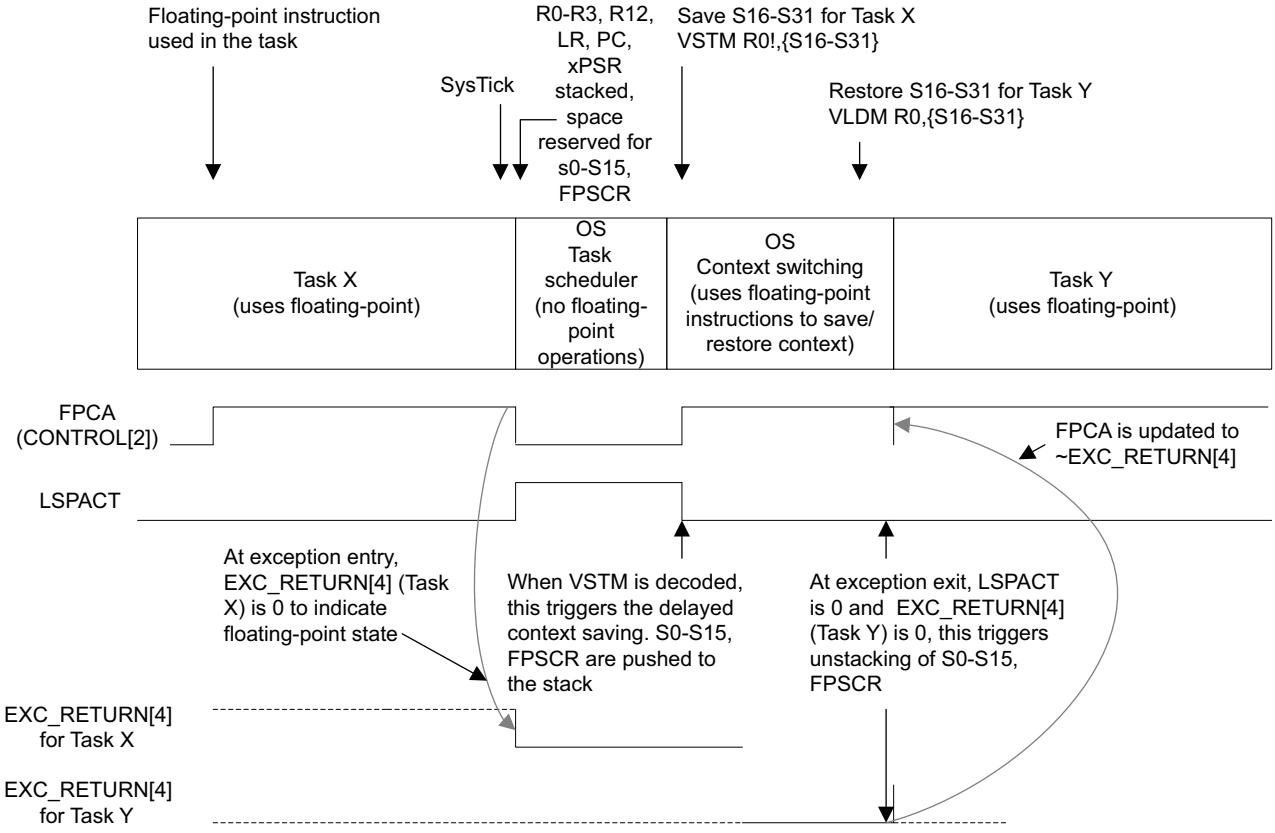

Figure 8 Context switching with floating-point instructions in multiple tasks

During context switching, which usually takes place within the PendSV exception, the processor executes the VSTM instruction (also known as the Floating-point Store Multiple command), which stores multiple floating-point registers to memory to store the S16 to S31 registers to the context data in the process stack or *Task Control Block* (TCB), depending on OS implementation, as shown in Figure 8.

Before the VSTM instruction is executed, the processor detects the use of the FPU in a new context while the S0-S15 registers and FPSCR from the previous context have not yet been saved to the stack, (indicated by the LSPACT bit in FPCCR). This triggers the pending lazy stacking, causing the floating-point registers S0-S15 and the FPSCR to be pushed to the memory location specified by FPCAR.

The context switching code can then restore the context of the next task for registers S16 to S31 of the FPU. The restore of registers S0-S15, and the FPSCR can be handled in the exception return when the PendSV exception ends. By setting both EXC_RETURN[4] and LSPACT to 0, the restore of registers S0-S15 and FPSCR are automatic.

In this arrangement, each context switch for tasks requiring the FPU requires at least 2 x 34 = 68 cycles, using registers S0-S31 and FPSCR, to handle the store and restore of floating-point registers, plus addition cycles in context switching software to check the status of bit[4] in EXC_RETURN. The checking of the EXC_RETURN[4] bit must be performed within the context switching software.

4.4 Other considerations

When creating a new application task, EXC_RETURN[4] must be set to 1, to use a return stack without the floating-point registers.

Note

- When the CONTROL.FPCA bit, that is, bit[2] of CONTROL register is set as a result of a floating-point operation, it stays HIGH for that application task. When this application task is suspended, this information is stored on EXC_RETURN[4], that is:
  - bit[4] of EXC_RETURN is LOW if CONTROL.FPCA is 1.
- If a task decides that it no longer requires the FPU, it can clear the CONTROL.FPCA bit. The saved EXC_RETURN[4] must not be changed by the task scheduler when the process is suspended because this bit indicates the size of the stack frame when the task was last pre-empted.

When dealing with systems where the FPU is used by multiple tasks, the context switching code can be designed in either of the following ways:

1. The context of the FPU is always saved and restored.
2. The context of the FPU is saved and restored only if the associated EXC_RETURN[4] value is 0. This means the value of CONTROL.FPCA was 1 when the task was running, before entering the context switching code.

Method 1 is simpler, but takes longer for each context switch and requires more memory space.

ARM recommends that:

- C compilers do not generate floating-point instructions when there is no floating-point operation
- functions in run time libraries do not contain floating-point instructions when they contain no floating-point operations.

Not following these recommendations results in increased times for interrupt latency, stack memory size and context switching. See *Tool support considerations* on page 26 for more information.

5 Alternative context switching scheme

This section consists of:

- *Concept of the lazy stacking context switching strategy*
- *Comparison of the two approaches* on page 24
- *Additional considerations* on page 24.

5.1 Concept of the lazy stacking context switching strategy

You can develop an alternative context switching mechanism based on lazy stacking. This is similar to lazy FPU stacking as implemented in a Linux kernel for classic ARM processors.

This method attempts to utilize the lazy context saving mechanism in the Cortex-M4F, so that it carries out the saving and restoration of FP registers only when necessary. For example, when there is only one application process using the FPU, then there is no requirement to carry out any context saving of the floating-point registers, as shown in Figure 9.

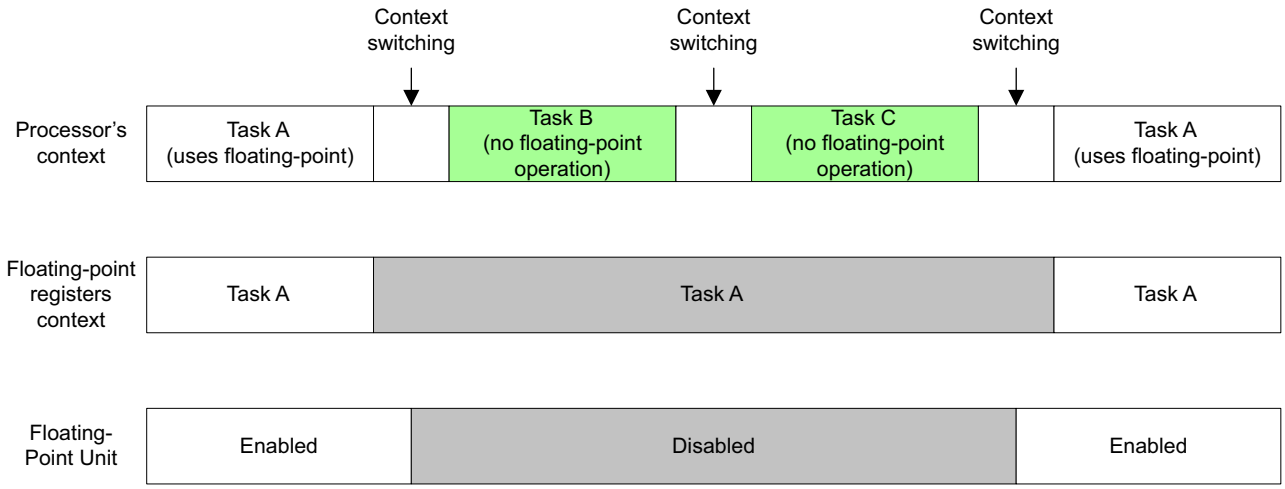

Figure 9 Lazy context switching with floating-point instructions in only one task

The FPU is disabled when switching to another application task during context switching. When the OS switches the context back to the last process that used the FPU, the unit is re-enabled by the context switching code. Although the FPU is disabled, the context information inside it is retained. The disabling or enabling of the FPU can be handled using the *Co-Processor Access Control Register* (CPACR) at address `0xE000ED88`.

If another application task uses the FPU, and the FPU is disabled, a fault exception is triggered:

- if the usage fault exception is enabled and priority permits it, the usage fault handler is executed, otherwise
- the HardFault exception handler is executed.

When this occurs, the fault exception handler can then enable the FPU, and trigger the context switching process by setting the PendSV pending status, see Figure 10 on page 20.

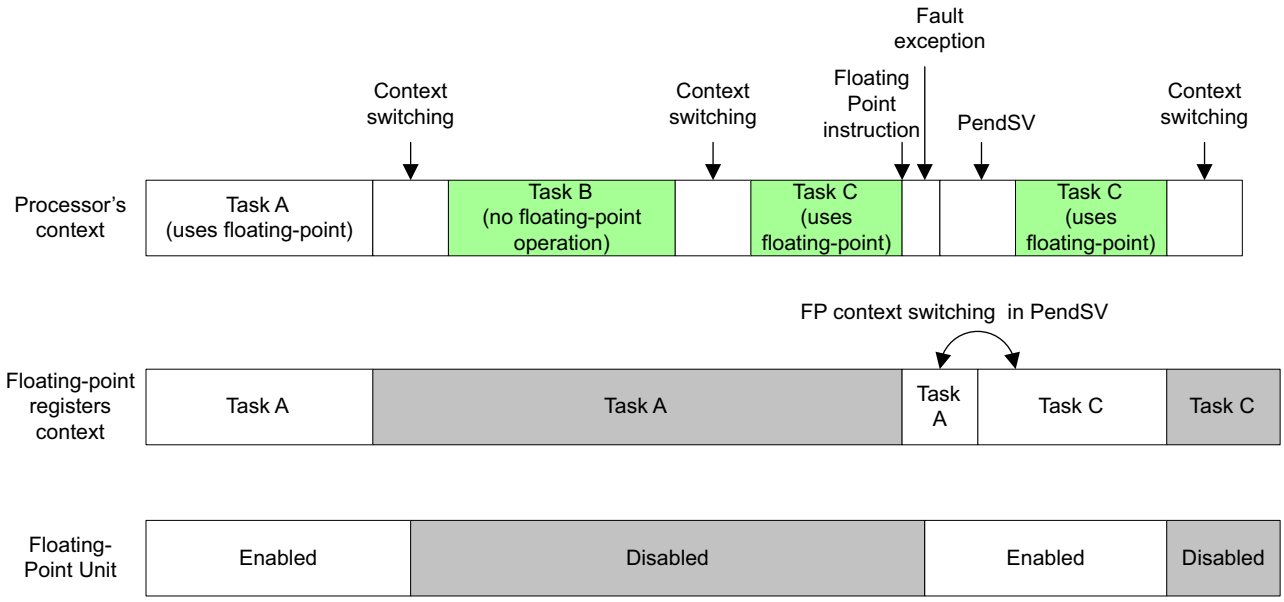

Figure 10 Context switching with floating-point instructions in multiple tasks

During the execution of PendSV, when executing the first floating-point instruction, it also triggers the lazy stacking operation to save registers S0-S15 and FPSCR into the reserved stack space when Task A was pre-empted.

To look at the context switching, that is, the process in PendSV more thoroughly, see Figure 11 on page 21.

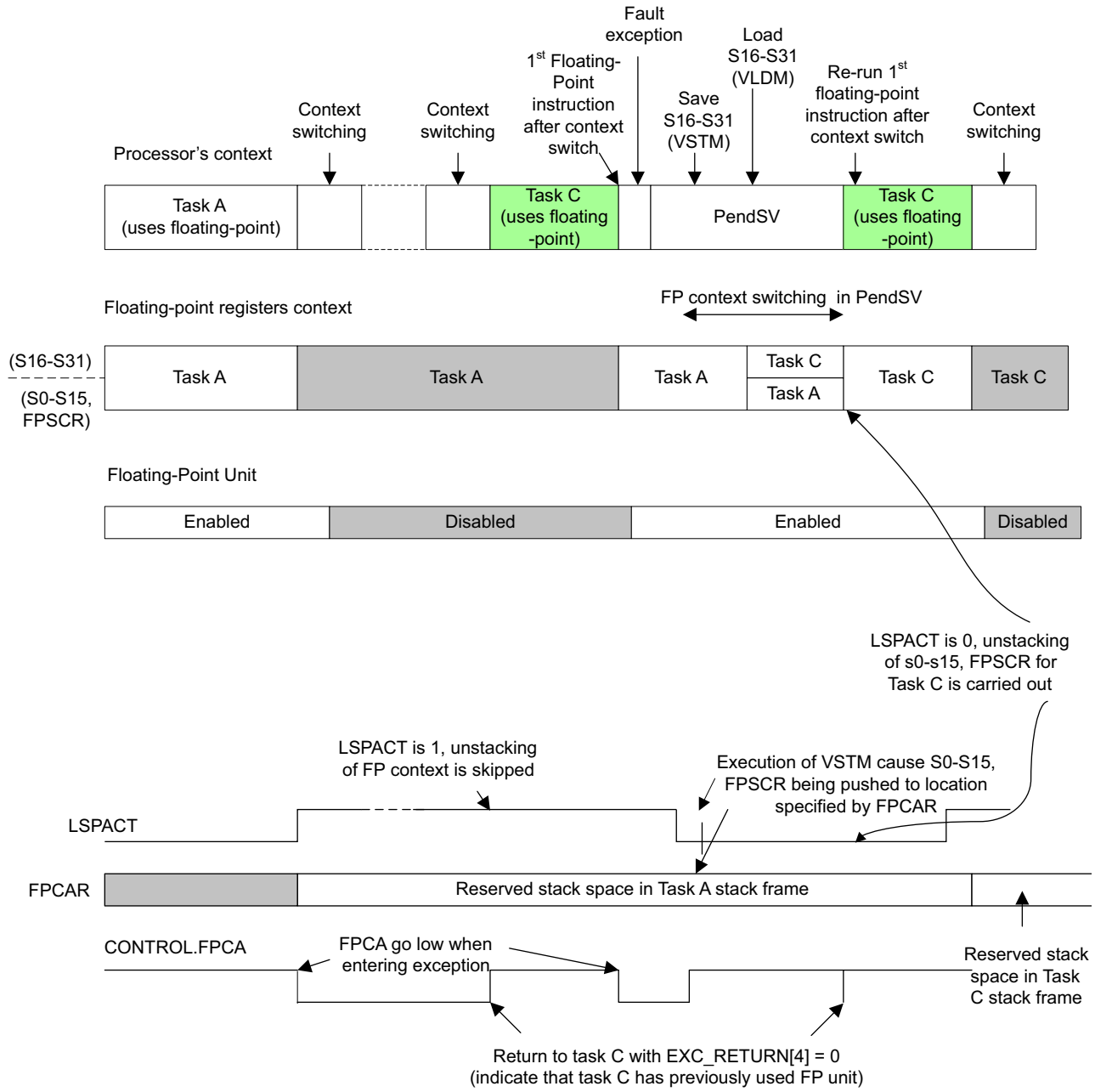

Figure 11 Context switching

Because fault exceptions can be triggered by other fault conditions, the fault exception handler HardFault or Usage Fault, must first check various Fault Status Registers and the status of the CPACR register:

- If the FPU was turned off, that is, CP10 and CP11 field in CPACR were zeros, and if *No Coprocessor Usage Fault* (NOCP) bit[3] in the Usage Fault Status Register is 1, then the OS must:
  - clear the NOCP bit
  - enable the FPU (CP10 and CP11)
  - set PendSV pending status
  - update software variables used by the OS for context switching control.

- If the FPU was already turned on, and if the NOCP bit is 1, then the fault is related to denial of access to a different coprocessor.
- If the NOCP bit is 0, then the fault is not caused by denial of access to a coprocessor.

If any of the interrupt service routines have to use the FPU, the PendSV operation is slightly different. Consequently, the fault handler for handling the NOCP fault condition must check the stacked IPSR, to determine whether the FPU is being called from an application task or an interrupt service routine. After this information is obtained, the fault handler can then set up software variables used by the PendSV accordingly. In this way the PendSV exception can handle the floating-point register saving differently when the floating-point is used by an interrupt service routine.

After the hard fault handler is executed, the interrupt service routine that used the FPU, and triggered the NOCP fault, is resumed and starts the execution of the first floating-point instruction. This triggers the lazy stacking so registers S0-S15 and FPSCR are pushed to the stack space pointed to by FPCAR. The rest of the floating-point registers, S16-S31, if being modified by the interrupt service routine, must be saved by the ISR itself, which is an AAPCS requirement.

Figure 12 and Figure 13 on page 23 show the use of floating-point instructions in ISR with the lazy stacking context switching scheme.

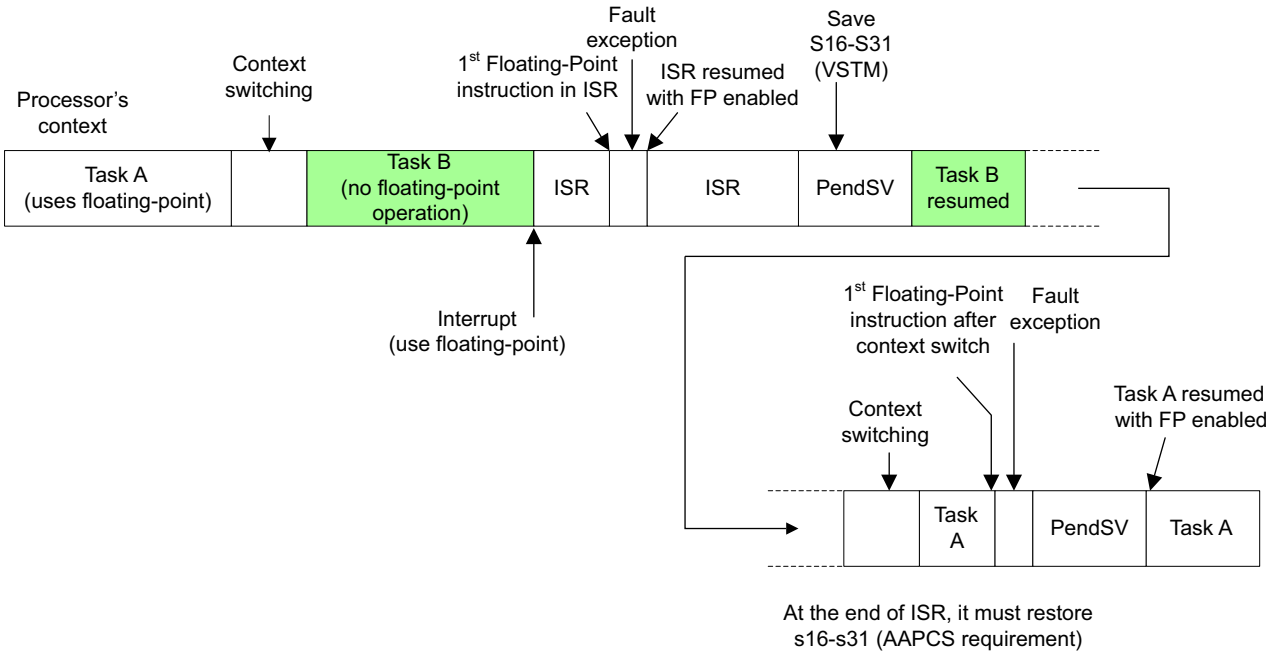

Figure 12 Use of floating-point instructions in ISR with lazy stacking context switching scheme (a)

Figure 13 on page 23 show the use of floating-point instructions in an ISR with a lazy stacking context switching scheme.

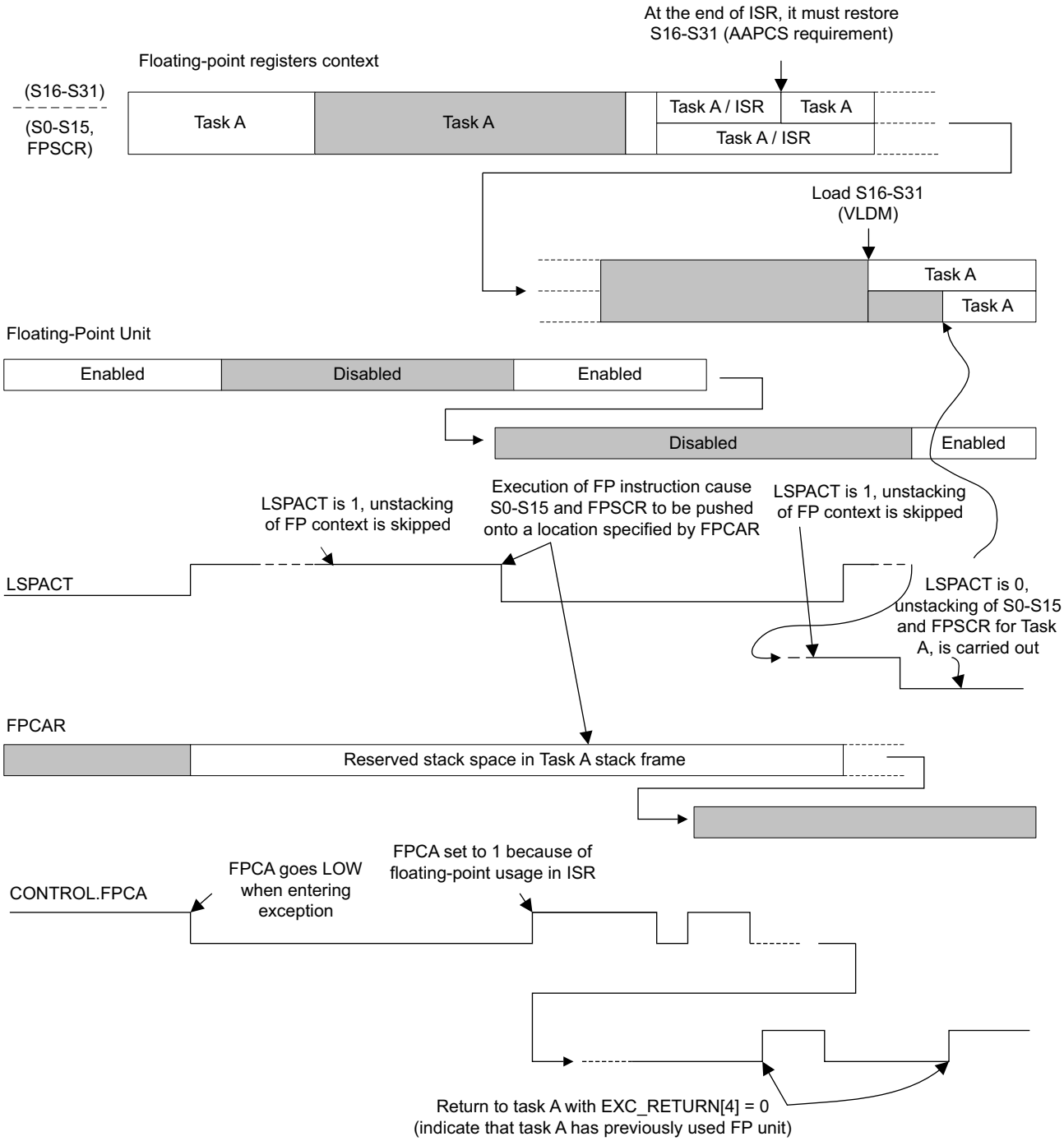

Figure 13 Use of floating-point instructions in ISR with lazy stacking context switching scheme (b)

After the ISR is completed, the PendSV exception is executed. The PendSV can then:

- disable the FPU
- save the values of S16-S31 to the *Task Control Block* (TCB) and update software variables in OS to indicate that the floating-point register contents for Task A are saved
- the next time Task A is resumed, reload all floating-point registers from the stack and TCB.

If an ISR that uses the FPU is triggered when the unit is enabled, there is no fault exception triggered when the interrupt service routine uses the FPU. In such a case, the saving of the floating-point registers is the same as normal floating-point context saving in software without an OS.

If an ISR that uses the FPU is triggered during context switching, that is, during a PendSV exception, and the interrupt triggering occurs after the FPU is disabled, then the fault handler sets the PendSV pending state and the PendSV exception is re-entered after the current PendSV completes its process. In this case the PendSV exception executes twice in a row. The second execution of the PendSV handler disables the FPU again.

To use this lazy stacking context switching scheme, the ASPEN and LSPEN bits in FPCCR must both set to 1, which is the default setting.

5.2 Comparison of the two approaches

Table 3 shows that both approaches have their own benefits.

Table 3 Comparison of both approaches

Context saving strategy	
Always save floating-point content	Pros: • Easier to implement • Predictable timing. Cons: • Longer context switching time.
Lazy stacking	Pros: • Floating-point context switching is carried out only when required, reducing the average context switching time. Cons: • If many tasks use the FPU, the overhead of using fault exception, and PendSV to handle floating-point context switching might result in larger overhead. • Can be difficult to debug certain tasks in halt mode, because registers in the FPU might not hold the context of the current task. • Latency for interrupt services that require the FPU, can increase as a consequence of disabling the FPU.

In many cases, it can be difficult to estimate the actual performance gain, or potential loss, of using the lazy stacking scheme unless real world benchmarking is carried out.

5.3 Additional considerations

The lazy stacking approach for context switching also brings additional considerations:

- *Non-Maskable Interrupts* (NMI) and HardFault handler must not use any floating-point operations.
- If the usage fault is used to handle the enabling of the FPU, any interrupt service that requires the FPU must have a lower priority than the usage fault handler.
- It requires that compiler generated code and runtime libraries do not generate any floating- point instructions if the program code does not contain floating-point operations. See *Tool support considerations* on page 26.

- If the application that you are working on also requires the use of fault handlers, the OS fault handler must be executed first, and then if the fault is not caused by FPU trapping, you must branch to your fault handlers.
- Application tasks must not enable the FPU themselves. So you must set CP10 and CP11 in CPACR. Otherwise, the context of another task is not saved, and consequently lost.

Note

— CPACR is a 32-bit register. Both CP10 and CP11 have a 2-bit length. CP10 covers bits[21:20] and CP11 covers bits[23:22]. The remainder of the register is reserved including bits[19:0] and bits[31:24].

— To set CP10, CP11 and with *Cortex Microcontroller Software Interface Standard* (CMSIS) compliant driver, you can use:

```
SCB->CPACR = (0x3 << (10*2)) | (0x3 << (11*2))
```

6 Tool support considerations

This section describes:

- *Compilers and runtime library implications*
- *Current tool status* on page 27.

6.1 Compilers and runtime library implications

OS content switching designed with the lazy stacking strategy requires that tool-chains, that is, compilers and runtime libraries, behave in certain ways. Otherwise, such an OS might not able to operate or could have very poor performance. Most of these requirements are listed in Requirements 1 and 2. OS context switching with the *always save floating-point context strategy*, can also benefit from these requirements.

Requirement 1

The compiler must not generate floating-point instructions for code that does not use floating-point itself.

Note

Use of floating-point includes calling a function which has a floating-point argument.

If requirement 1 is not met, then floating-point instructions could be generated in a large number of tasks and processes. This can increase the overhead in context switching in addition to affecting latency in interrupt handling. The situation could be worst where the OS code is compiled together with the applications. In such cases the OS code might end up containing floating-point instructions.

This issue is made more complicated by the fact that the *Embedded Application Binary Interface* (EABI) standard permits the use of the FPU for non-floating-point operations. For example, if the code is compiled with compile options which specify that a FPU is available, and the function being compiled requires many registers for data processing and ends up fully utilizing the registers in the general register bank, then the C compiler might use some of the registers in the FPU as temporary data storage.

Note

Such an arrangement is permitted in EABI, because it makes sense for application processors where memory accesses can take a very long time if there are cache misses. But in embedded applications where the Cortex-M4F is targeted, the interrupt rate can be substantially higher and the floating-point context switching overhead could reduce the performance of the system.

It is possible to compile the software files separately with different compiler options, and link the object files together afterwards to ensure that only application processes that use floating-point features are compiled with the floating-point option.

Note

This method might not be suitable for users of the gcc tool chain. Most C compiler vendors providing a gcc based tool chain recommend their customers compile and link the application in one step because with gcc, separating the compilation and linking steps can be error prone.

Requirement 2

Runtime library functions must not use the FPU unless they are floating-point functions. The only special cases are:

1. `print()` or `printf()` function family
2. `setjmp()` and `longjmp()` functions.

This is similar to requirement 1, the use of floating-point instructions can increase the overhead in context switching and interrupt services.

Ideally, the tool chain could also:

1. Provide a compilation switch (and/or pragma) that permits you to compile sources so that it would report an error if floating-point use was found, as a result of direct use of floating-point data in application code. This would help you to detect accidental use of floating-point operations, for example, through a macro, or in-lined assembly code, or in an argument auto-casted. In addition:
   a. The use of floating-point through a runtime function is assumed to be detected only by arguments or a return value.
   b. It is understood that the compiler cannot detect a call to a function which contains floating point operations but uses no floating-point arguments or a return value.
2. Provide a compilation switch that specifies not to use the upper 16 single precision FPU registers. For cases where the floating-point is not heavily used, for example, where there is minimum register pressure, and not using the upper 16 registers would reduce context switching time.
3. Ensure the debugger does not enable the vector catch for NOCP when the lazy stacking context saving strategy is used. Otherwise the processor is halted automatically when the NOCP fault occurs. The debugger must also permit accesses to the floating-point registers.

6.2 Current tool status

This section describes:

ARM/Keil development tools

Table 4 shows the number of development tool chains already support the Cortex-MF4.

Table 4 ARM C compiler related switches

Compile switch	version	Description
`--no_allow_fpreg_for_nonfpdata`	armcc v4.1 patch 4, or later versions of armcc v4 (default is disabled) armcc v5.0 update 1,or later versions of armcc v5 (default is enabled)	Disable the use of floating-point registers and floating-point instructions for non-floating-point data. **Note** The opposite of this option is: `--allow_fpreg_for_nonfpdata`
`--fpu=none`	RealView compiler 2.0 or later.	Selects no floating-point option, so no floating-point code can be used. **Note** An error is generated if your code contains float types.

Runtime library

This consists of:

- Currently `memcpy()` and `printf()` functions for Cortex-M do not use floating-point instructions.
- A few functions in mathlib might contain floating-point instructions for non-floating-point data.

In Keil MDK-ARM 4.21 (armcc v4.1.0.713), or later versions that are based on armcc version 4.1, the option `--no_allow_fpreg_for_nonfpdata` must be added to the Misc Controls field of the C/C++ options to disable the use of floating-point instructions in non-floating point code.

GNUC C compiler (gcc)

If a program is compiled with the FPU option, gcc might make use of the floating-point registers if register pressure is high, and running low on available registers for data processing. In some cases, the memory copy might also utilize floating-point registers to hold data.

It is possible to avoid the use of floating-point instructions in non-floating-point code by using `-mfloat-abi=soft`.

By default, libraries are built with `-mfloat-abi=soft`. So they must not contain floating-point instructions. However, because there are various gcc vendors with different build options, you might have to check with your gcc tool chain supplier to find out the status of the libraries.

IAR Embedded Workbench for ARM

Table 5 shows that the IAR C compiler does not use any floating-point registers in the following two cases:

Table 5 IAR C compiler commands

Command	Description
`--fpu=none`	Floating-point is handled by library functions
`--fpu=xxx`	Your code does not use floating-point

7 Additional Reading

This section lists publications by ARM and by third parties.

See Infocenter, http://infocenter.arm.com, for access to ARM documentation.

7.1 ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:

- *ARMv7-M Architecture Reference Manual* (ARM DDI 0403)
- *ARM Compiler toolchain, Compiler Reference* (ARM DUI 0491)
- *Cortex-M4 Devices Generic User Guide* (ARM DUI 0553).

7.2 ARM Technical Support Knowledge Article

This article contains information that is specific to this product:

- *How do I get the best performance when compiling floating-point code for Cortex*-M4F?
  See Infocenter, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka15451.html, for more information.