Exceptions and Interrupts

Overview

- *What Are Exceptions and Interrupts?*
- *Exception Types on the Cortex-M0 Processor*
- *Exception Priority Definition*
- *Vector Table*
- *Exception Sequence Overview*
- Exception Entry Sequence
- Exception Exit Sequence

What Are Exceptions and Interrupts?

- Exceptions are events that cause changes in program flow control outside a normal code sequence.
- When it happens, the program that is currently executing is suspended, and a piece of code associated with the event (the exception handler) is executed.
- The events could either be external or internal. When an event is from an external source, it is commonly known as an interrupt or interrupt request (IRQ).

Exceptions and Interrupts

- The software code that is executed when an exception occurs is called exception handler.
- If the exception handler is associated with an interrupt event, then it can also be called an interrupt handler, or interrupt service routine (ISR).
- The exception handlers are part of the program codein the compiled program image.

Exceptions and Interrupts

- When the exception handler has finished processing the exception, it will return to the interrupted program and resume the original task.
- As a result, the exception handling sequence requires some way to store the status of the interrupted program and allow this information to be restored after the exception handler has completed its task.
- This can be done by a hardware mechanism or by a combination of hardware and software operations.

Exception Types on the Cortex-M0 Processor

Table 8.1: List of Exceptions in the Cortex-M0 Processor

Exception Number	Exception Type	Priority	Descriptions
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt
3	Hard fault	−1	Fault handling exception
4–10	Reserved	NA	—
11	SVC	Programmable	Supervisor call via SVC instruction
12–13	Reserved	NA	—
14	PendSV	Programmable	Pendable request for system service
15	SysTick	Programmable	System tick timer
16	Interrupt #0	Programmable	External interrupt #0
17	Interrupt #1	Programmable	External interrupt #1
...	...	...	...
47	Interrupt #31	Programmable	External interrupt #31

Exception Types on the Cortex-M0 Processor

Nonmaskable Interrupt (NMI)

- The NMI is similar to IRQ, but it cannot be disabled and has the highest priority apart from the reset.
- It is very useful for safety critical systems like industrial control or automotive.
- Depending on the design of the microcontroller, the NMI could be used for power failure handling, or it can be connected to a watchdog unit to restart a system if the system stopped responding.
- Because the NMI cannot be disabled by control registers, the responsiveness is guaranteed.

Exception Types on the Cortex-M0 Processor

Hard Fault

- Hard fault is an exception type dedicated to handling fault conditions during program execution.
- These fault conditions can be trying to execute an unknown opcode, a fault on a bus interface or memory system, or illegal operations like trying to switch to ARM state.

Exception Types on the Cortex-M0 Processor

SVCall (SuperVisor Call)

- SVC exception takes place when the SVC instruction is executed. SVC is usually used in systems with an operating system (OS), allowing applications to have access to system services.

PendSV (Pendable Service Call)

- PendSV is another exception for applications with OS. Unlike the SVC exception, which must start immediately after the SVC instruction has been executed, PendSV can be delayed.
- The OS commonly uses PendSV to schedule system operations to be carried out only when high-priority tasks are completed.

Exception Types on the Cortex-M0 Processor

SysTick

- The System Tick Timer inside the NVIC is another feature for OS application.
- Almost all operating systems need a timer to generate regular interrupt for system maintenance work like context switching.
- By integrating a simple timer in the Cortex-M0 processor, porting an OS from one device to another is much easier.
- The SysTick timer and its exception are optional in the Cortex-M0 microcontroller implementation.

Interrupts

- The Cortex-M0 microcontroller could support from 1 to 32 interrupts. The interrupt signals could be connected from on-chip peripherals or from an external source via the I/O port.
- In some cases (depending on the microcontroller design), the external interrupt number might not match the interrupt number on the Cortex-M0 processor.

Exception Priority Definition

- In the Cortex-M0 processor, each exception has a priority level. The priority level affects whether the exception will be carried out or if it will wait until later (stay in a pending state).
- The Cortex-M0 processor supports three fixed highest priority levels and four programmable levels. For exceptions with programmable priority levels, the priority level configuration registers are 8 bits wide, but only the two MSBs are implemented

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented		Not implemented, read as zero					

Figure 8.1:
A priority-level register with 2 bits implemented.

Exception Priority Definition

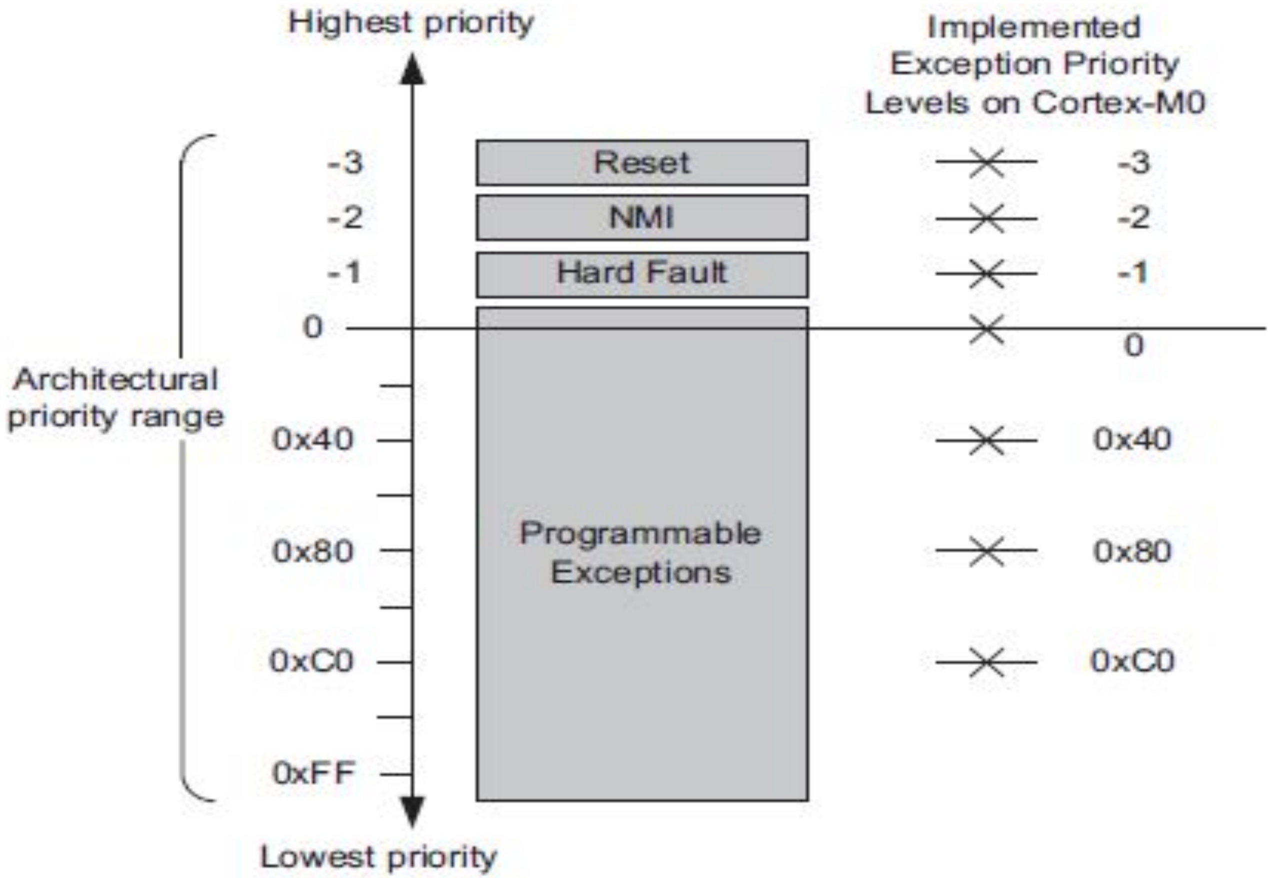


Figure 8.2:
Available priority levels in the Cortex-M0 processor.

Vector Table

- When the Cortex-M0 processor starts to process an interrupt request, it needs to locate the starting address of the exception handler.
- This information is stored in the beginning of the memory space, called the vector table
- The order of exception vector being stored is the same order of the exception number.
- Because each vector is one word (four bytes), the address of the exception vector is the exception number times four.
- Each exception vector is the starting address of the exception handler, with the LSB set to 1 to indicate that the exception handler is in Thumb code.

Vector Table

Memory Address		Exception Number
0x0000004C	Interrupt#3 vector	19
0x00000048	Interrupt#2 vector	18
0x00000044	Interrupt#1 vector	17
0x00000040	Interrupt#0 vector	16
0x0000003C	SysTick vector	15
0x00000038	PendSV vector	14
0x00000034	Not used	13
0x00000030	Not used	12
0x0000002C	SVC vector	11
0x00000028	Not used	10
0x00000024	Not used	9
0x00000020	Not used	8
0x0000001C	Not used	7
0x00000018	Not used	6
0x00000014	Not used	5
0x00000010	Not used	4
0x0000000C	Had Fault vector	3
0x00000008	NMI vector	2
0x00000004	Reset vector	1
0x00000000	MSP initial value	0

Note: LSB of each vector must be set to 1 to indicate Thumb state

Figure 8.3:
Vector table.

Exception Sequence Overview

- ***Acceptance of Exception Request***

The processor accepts an exception if the following condition are satisfied

- For interrupt and SysTick interrupt requests, the interrupt has to be enabled
- The processor is not running an exception handler of the same or a higher priority
- The exception is not blocked by the PRIMASK interrupt masking register

Exception Sequence Overview

- ***Stacking and Unstacking***
- ***Tail Chaining***
- ***Late Arrival***

Stacking and Unstacking

- To allow an interrupted program to be resumed correctly, **some parts of the current state of the processor must be saved before the program execution** switches to the exception handler that services the occurred exception.
- When an exception is accepted on the Cortex-M0 processor, some of the registers in the **register banks (R0 to R3, R12, R14), the return address (PC), and the Program Status Register (xPSR) are pushed to the current active stack memory automatically**.
- The **Link Register (LR/R14) is then updated to a special value** to be used during exception return **(EXC_RETURN**),and then the exception vector is automatically located and the exception handler starts to execute.

Stacking and Unstacking

- At the end of the exception handling process, the exception handler executes a return using the special value (EXC_RETURN, previously generated in LR) to trigger the exception return mechanism.
- The actions of automatically **saving and restoring of the register** contents are called "stacking" and "unstacking"

Stacking and Unstacking

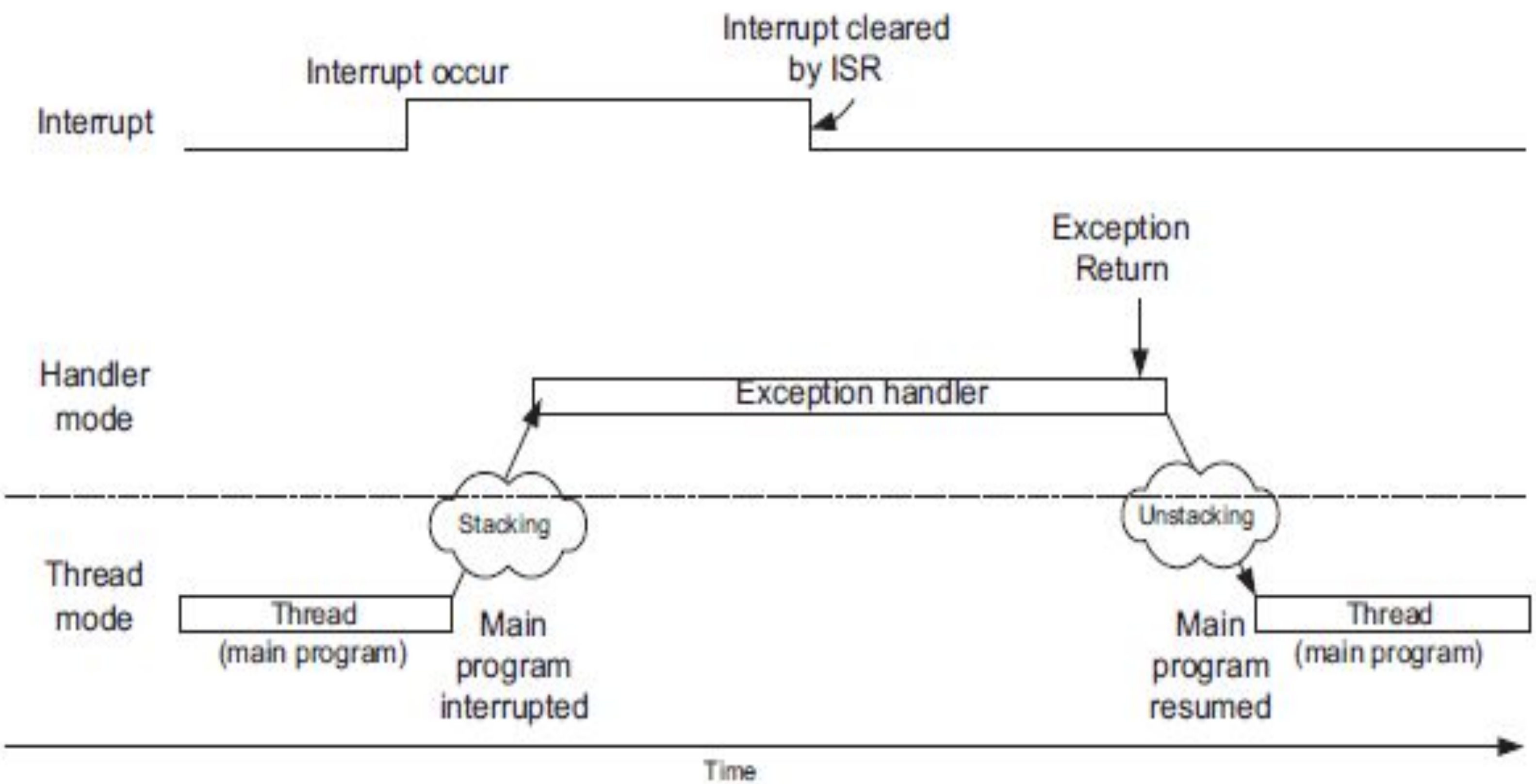


Figure 8.4:
Stacking and unstacking of registers at exception entry and exit.

Tail Chaining

- **If an exception is in a pending state when another exception handler has been completed**, instead of returning to the interrupted program and then entering the exception sequence again, a tail-chain scenario will occur.
- **When tail chain occurs, the processor will not have to restore all register values from the stack and push them back to the stack again**. The tail chaining of exceptions allows lower exception processing overhead and hence better energy efficiency

Tail Chaining

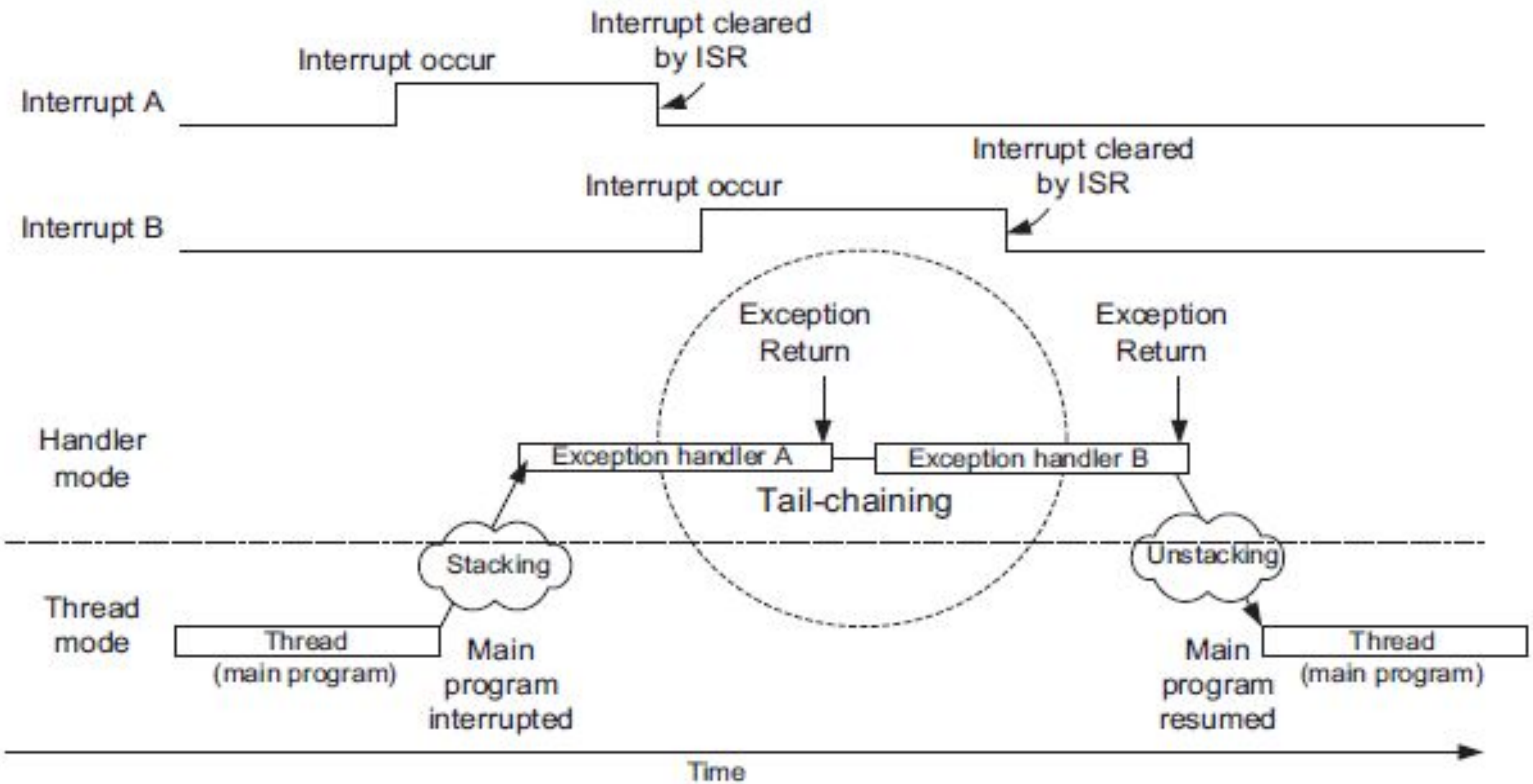


Figure 8.5:
Tail chaining of an interrupt service routine.

Late Arrival

- Late arrival is an optimization mechanism in the Cortex-M0 to speed up the processing of higher priority exceptions. **If a higher priority exception occurs during the stacking process of a lower priority exception, the processor switches to handle the higher-priority exception first** (Figure 8.6).
- At the end of the stacking process, the vector for the higher-priority exception is fetched instead of the lower-priority one. Without the late arrival optimization, a processor will have to preempt and enter the exception entry sequence again at the beginning of the lower-priority exception handler. This results in longer latency as well as larger stack memory usage.

Late Arrival

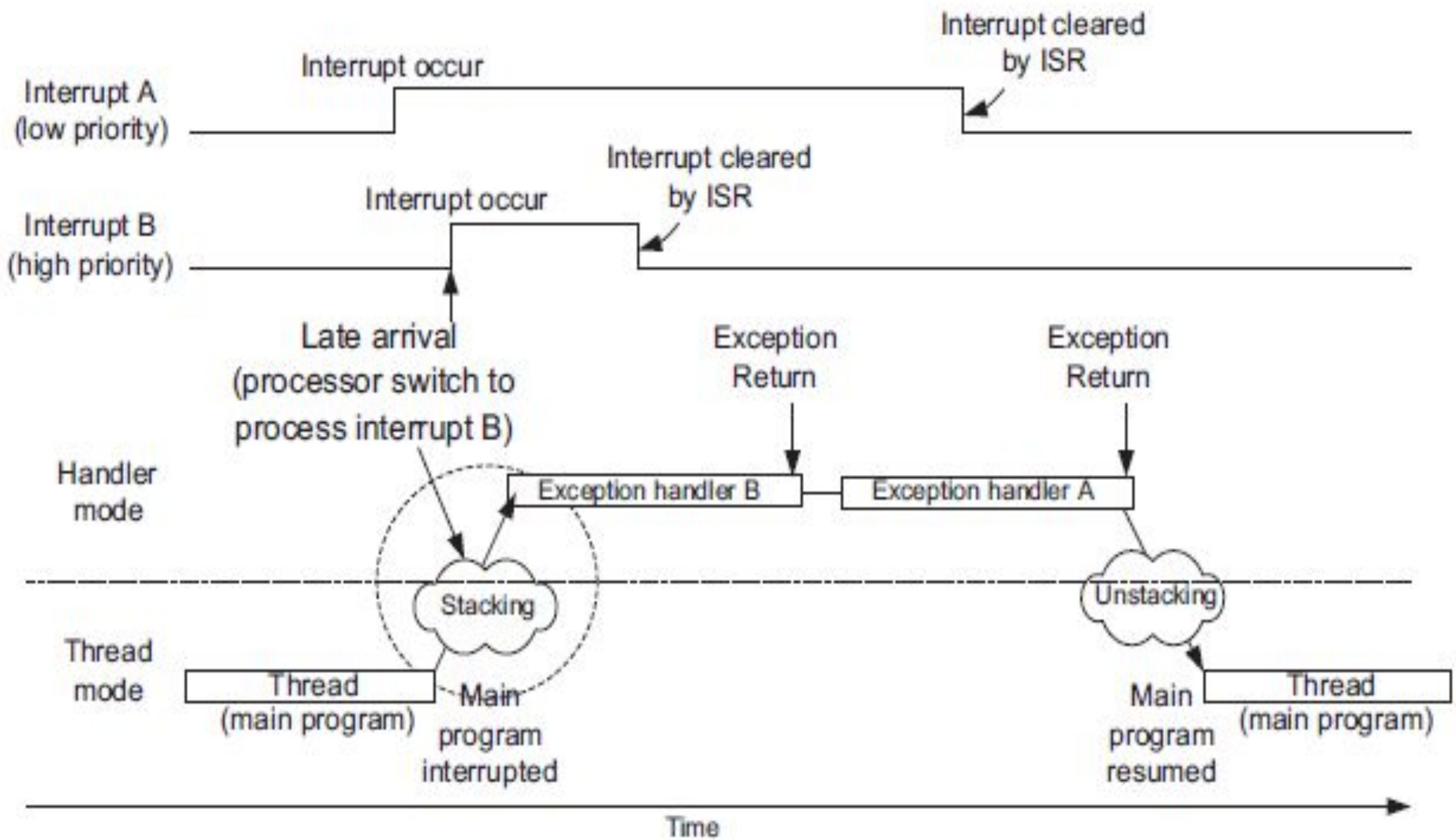


Figure 8.6:
Late arrival optimization.

EXC_RETURN

- The EXC_RETURN is a special architecturally defined value for triggering and helping the exception return mechanism.
- This value is generated automatically when an exception is accepted and is stored into the Link Register (LR, or R14) after stacking.
- The EXC_RETURN is a 32-bit value; the upper 28 bits are all set to 1, and bits 0 to 3 are used to provide information for the exception return mechanism

EXC_RETURN

- Bit 0 of EXC_RETURN on the Cortex-M0 processor is reserved and must be 1.
- Bit 2 of EXC_RETURN indicates whether the unstacking should restore registers from the main stack (using MSP) or process stack (using PSP).
- Bit 3 of EXC_RETURN indicates whether the processor is returning to Thread mode or Handler mode.

EXC_RETURN

Table 8.2: Bit Fields in EXC_RETURN Value

Bits	31:28	27:4	3	2	1	0
Descriptions	EXC_RETURN indicator	Reserved	Return mode	Return stack	Reserved	Reserved
Value	0xF	0xFFFFFF	1 (thread) or 0 (handler)	0 (main stack) or 1 (process stack)	0	1

Table 8.3: Valid EXC_RETURN Value

EXC_RETURN	Condition
0xFFFFFFF1	Return to Handler mode (nested exception case)
0xFFFFFFF9	Return to Thread mode and use the main stack for return
0xFFFFFFFD	Return to Thread mode and use the process stack for return

EXC_RETURN

- If the thread is using main stack (CONTROL register bit 1 is zero), the value of the LR will be set to 0xFFFFFFF9 when it enters an exception and 0xFFFFFFF1 when a nested exception is entered, as shown in Figure

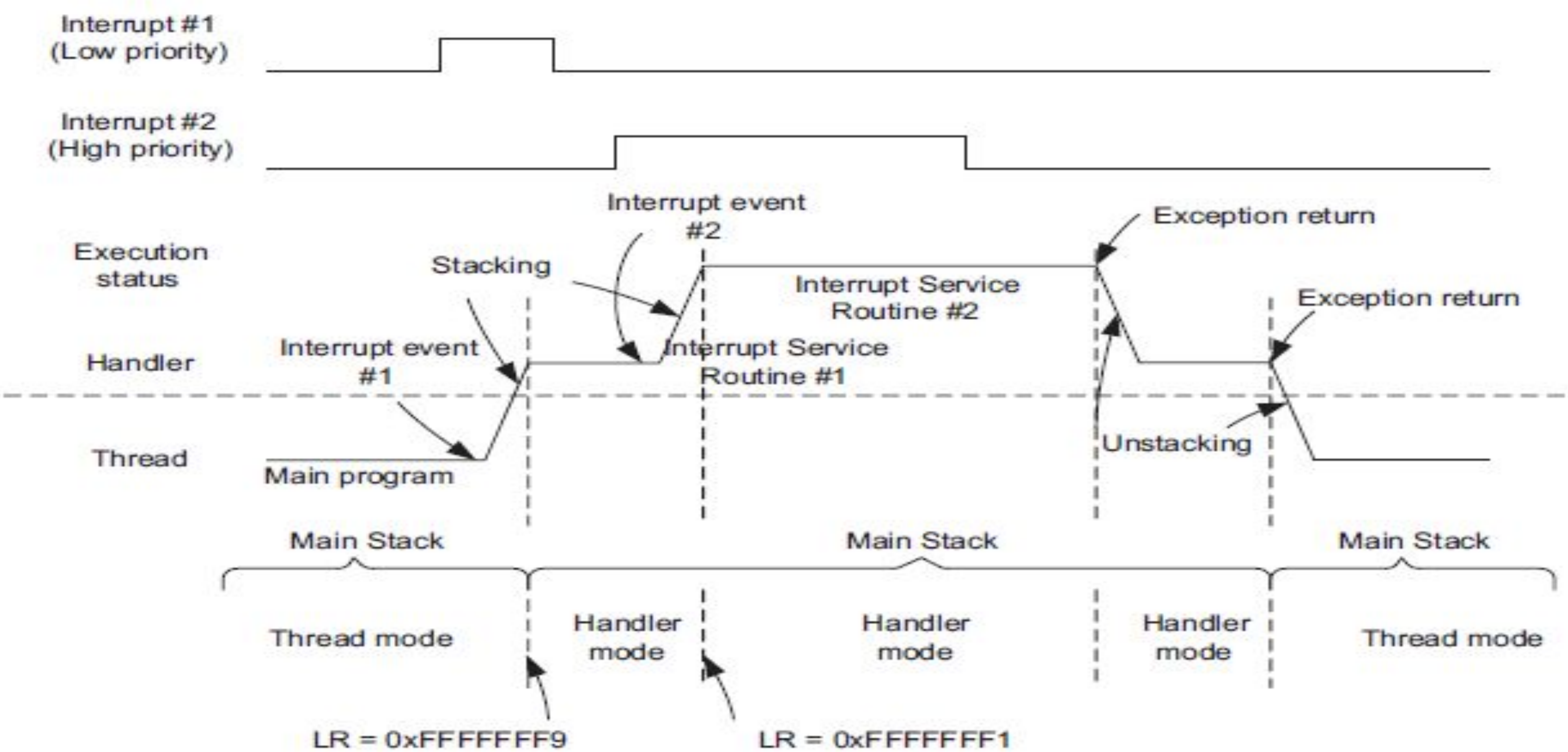


Figure 8.7:
LR set to EXC_RETURN values at exceptions (main stack is used in Thread mode).

EXC_RETURN

- If the thread is using process stack (CONTROL register bit 1 is set to 1), the value of LR will be 0xFFFFFFFD when entering the first exception and 0xFFFFFFF1 when entering a nested exception, as shown in Figure

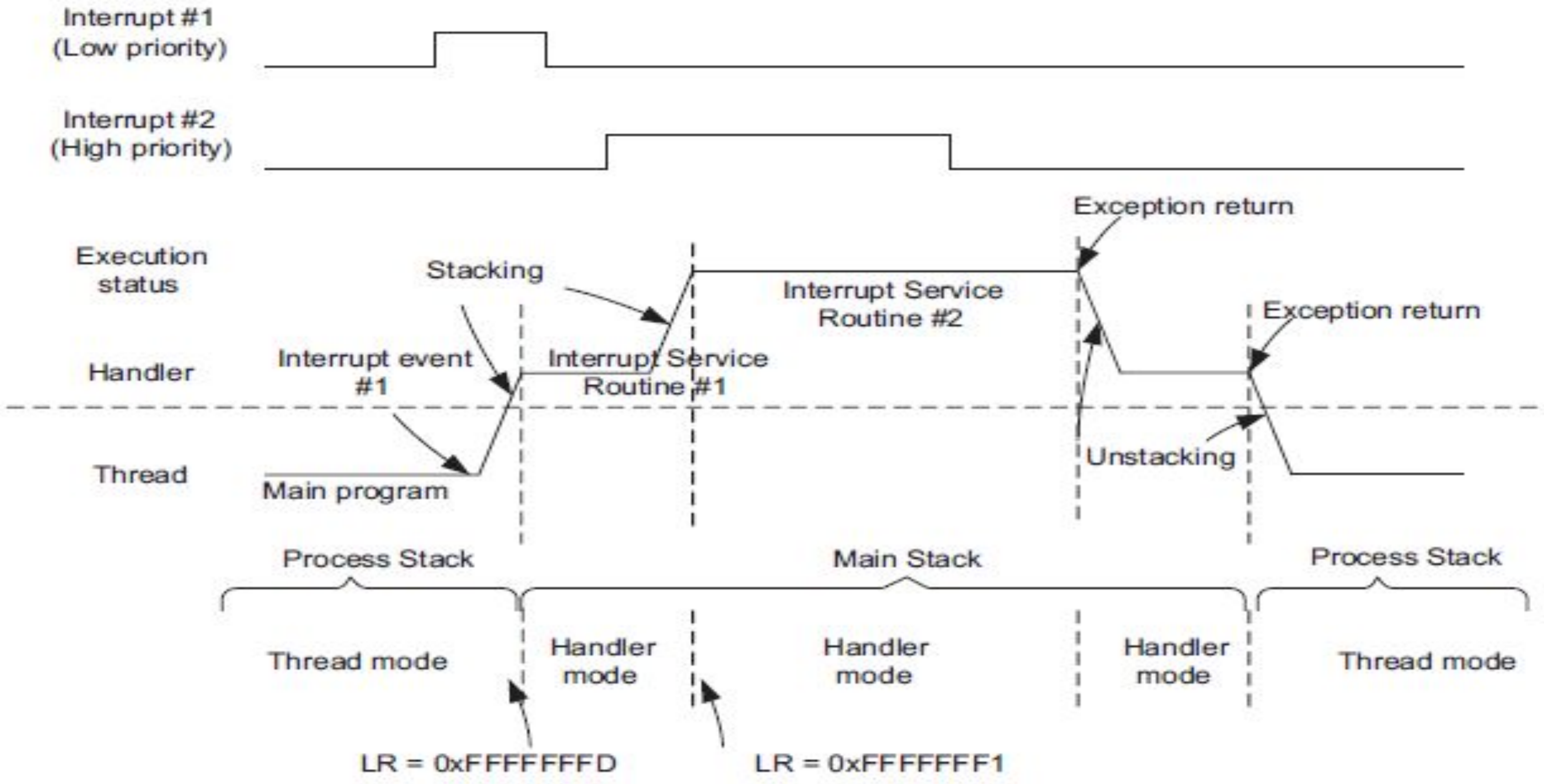


Figure 8.8:
LR set to EXC_RETURN values at exceptions (process stack is used in Thread mode).

Exception Entry Sequence

- The stacking of registers is carried out in the order shown in

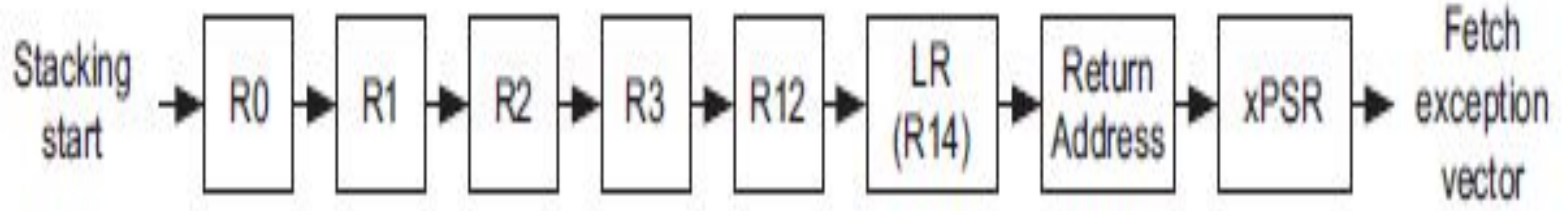
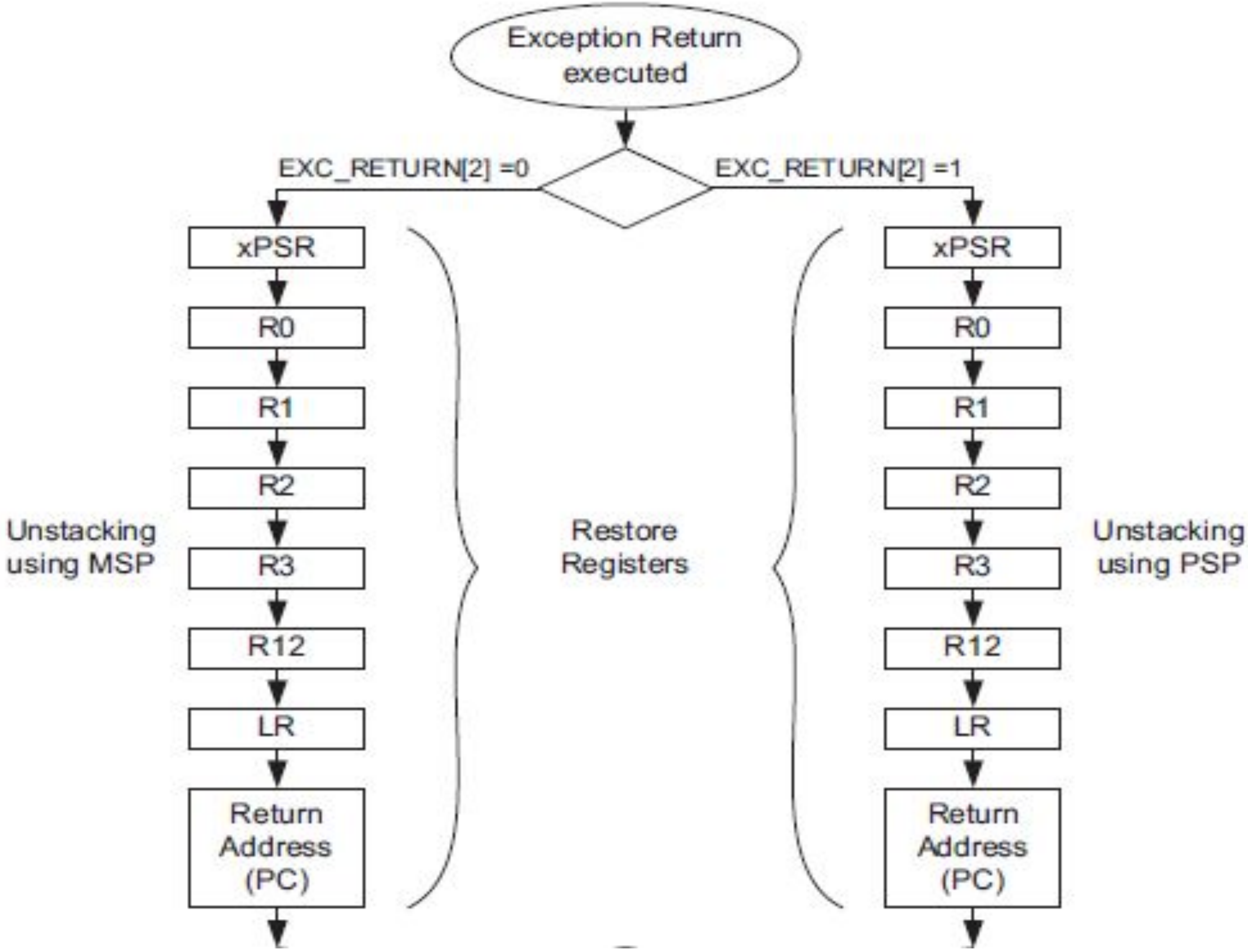


Figure 8.12:

Order of register stacking during the exception sequence in the Cortex-M0 processor.

Exception Exit Sequence



Exception Exit Sequence

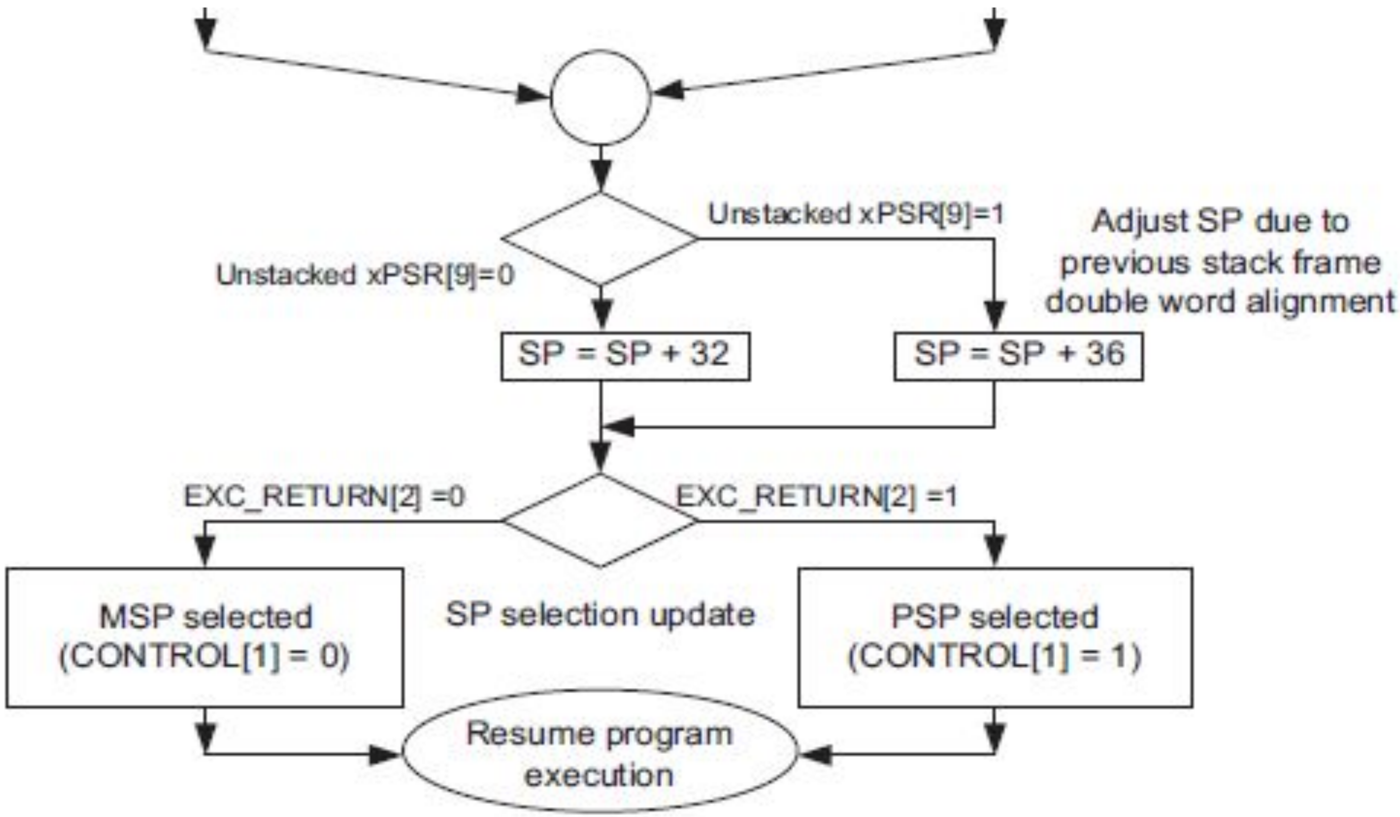


Figure 8.13:
Unstacking at the exception exit.

Questions?