Supervisor Calls (SVC)

P. Bernardi

Exception and Interrupt

- The terms *exception* and *interrupt* are often confused
- *Exception* usually refers to an internal CPU event such as
  - floating point overflow
  - MMU fault (e.g., page fault)
  - trap (SWI)
- *Interrupt* usually refers to an external I/O event such as
  - I/O device request
  - reset
- In the ARM architecture the ASM instruction to raise a software interrupt is
  - **SWI** (followed by an ID)
  - i.e., SWI 0x08

Exception Handling

- Reset
- NMI
- Faults
  - Hard Fault
  - Memory Management
  - Bus Fault
  - Usage Fault
- SVCall
- Debug Monitor
- PendSV
- SysTick Interrupt
- External Interrupt

Interrupt Vector Table (IVT) in ARM v7-M

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	1	0x00000004
NMI	2	0x00000008
Hard fault	3	0x0000000C
Memory management fault	4	0x00000010
Bus fault	5	0x00000014
Usage fault	6	0x00000018
SVcall	11	0x0000002C
Debug monitor	12	0x00000030
PendSV	14	0x00000038
SysTick	15	0x0000003C
Interrupts	≥16	≥0x00000040

Each line contains an address to be copied in the PC in case a specific exception occurs.

The access mechanism to the table is hardware-based and «transparent» to the programmer

Anyway, it is a programmer duty to setup the IVT at boot time.

Interrupt Vector Table (IVT) in ARM v7-M

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	**1**	**0x00000004**
NMI	2	0x00000008
Hard fault	3	0x0000000C
Memory management fault	4	0x00000010
Bus fault	5	0x00000014
Usage fault	6	0x00000018
SVcall	11	0x0000002C
Debug monitor	12	0x00000030
PendSV	14	0x00000038
SysTick	15	0x0000003C
Interrupts	≥16	≥0x00000040

Reset is invoked on power up or a warm reset. The exception model treats reset as a special form of exception.

When reset is asserted, the operation of the processor stops, potentially at any point in an instruction.

When reset is deasserted, execution restarts from the address provided by the reset entry in the vector table.

Execution restarts as privileged execution in Thread mode.

Interrupt Vector Table (IVT) in ARM v7-M

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	1	0x00000004
NMI	**2**	**0x00000008**
Hard fault	3	0x0000000C
Memory management fault	4	0x00000010
Bus fault	5	0x00000014
Usage fault	6	0x00000018
SVcall	11	0x0000002C
Debug monitor	12	0x00000030
PendSV	14	0x00000038
SysTick	15	0x0000003C
Interrupts	≥16	≥0x00000040

A *NonMaskable Interrupt* (NMI) can be signalled by a peripheral or triggered by software. This is the highest priority exception other than reset.

It is permanently enabled and has a fixed priority of -2.

NMIs cannot be:

- masked or prevented from activation by any other exception
- preempted by any exception other than Reset.

Interrupt Vector Table (IVT) in ARM v7-M

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	1	0x00000004
NMI	2	0x00000008
Hard fault	**3**	**0x0000000C**
Memory management fault	**4**	**0x00000010**
Bus fault	**5**	**0x00000014**
Usage fault	**6**	**0x00000018**
SVcall	11	0x0000002C
Debug monitor	12	0x00000030
PendSV	14	0x00000038
SysTick	15	0x0000003C
Interrupts	≥16	≥0x00000040

If the application ends up in one of these handlers, then something has gone wrong.
Hard faults are the most common fault type, as other fault types that are not enabled individually will be escalated to become a hard fault.

Interrupt Vector Table (IVT) in ARM v7-M

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	1	0x00000004
NMI	2	0x00000008
Hard fault	3	0x0000000C
Memory management fault	4	0x00000010
Bus fault	5	0x00000014
Usage fault	6	0x00000018
SVcall	**11**	**0x0000002C**
Debug monitor	12	0x00000030
PendSV	14	0x00000038
SysTick	15	0x0000003C
Interrupts	≥16	≥0x00000040

A supervisor call (SVC) is an exception that is triggered by the SVC instruction.
In an OS environment, applications can use SVC instructions to access OS kernel functions and device drivers.

Interrupt Vector Table (IVT) in ARM v7-M

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	1	0x00000004
NMI	2	0x00000008
Hard fault	3	0x0000000C
Memory management fault	4	0x00000010
Bus fault	5	0x00000014
Usage fault	6	0x00000018
SVcall	11	0x0000002C
Debug monitor	12	0x00000030
PendSV	**14**	**0x00000038**
SysTick	15	0x0000003C
Interrupts	≥16	≥0x00000040

PendSV is an interrupt-driven request for system-level service.
In an OS environment, use PendSV for context switching when no other exception is active.

Interrupt Vector Table (IVT) in ARM v7-M

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	1	0x00000004
NMI	2	0x00000008
Hard fault	3	0x0000000C
Memory management fault	4	0x00000010
Bus fault	5	0x00000014
Usage fault	6	0x00000018
SVcall	11	0x0000002C
Debug monitor	12	0x00000030
PendSV	14	0x00000038
SysTick	**15**	**0x0000003C**
Interrupts	≥16	≥0x00000040

A SysTick exception is an exception the system timer generates when it reaches zero.
In an OS environment, the processor can use this exception as system tick.

Interrupt Vector Table (IVT) in ARM v7-M

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	1	0x00000004
NMI	2	0x00000008
Hard fault	3	0x0000000C
Memory management fault	4	0x00000010
Bus fault	5	0x00000014
Usage fault	6	0x00000018
SVcall	11	0x0000002C
Debug monitor	12	0x00000030
PendSV	14	0x00000038
SysTick	15	0x0000003C
Interrupts	**≥16**	**≥0x00000040**

A interrupt, or IRQ, is an exception signalled by a peripheral, or generated by a software request. All interrupts are asynchronous to instruction execution. In the system, peripherals use interrupts to communicate with the processor.

Detailed information including priority

Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address or offset[b]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	-
4	-12	MemManage	Configurable[c]	0x00000010	Synchronous
5	-11	BusFault	Configurable[c]	0x00000014	Synchronous when precise, asynchronous when imprecise
6	-10	UsageFault	Configurable[c]	0x00000018	Synchronous
7-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable[c]	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable[c]	0x00000038	Asynchronous
15	-1	SysTick	Configurable[c]	0x0000003C	Asynchronous
16	0	Interrupt (IRQ)	Configurable[d]	0x00000040[e]	Asynchronous

System Handler Priority Registers

- The SHPR1-SHPR3 registers set the priority level, 0 to 255, of the exception handlers that have configurable priority.
- Interrupt sources priority level is setup by the Interrupt Controller

- Example: System Handler Priority Register 1 (SHPR1)

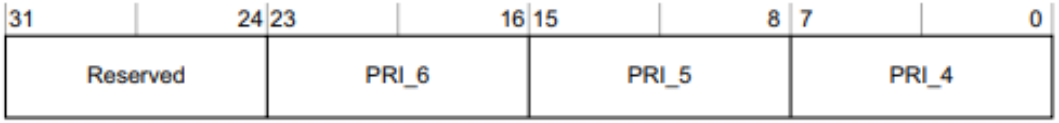


Table 4-21 SHPR1 register bit assignments

Bits	Name	Function
[31:24]	PRI_7	Reserved
[23:16]	PRI_6	Priority of system handler 6, UsageFault
[15:8]	PRI_5	Priority of system handler 5, BusFault
[7:0]	PRI_4	Priority of system handler 4, MemManage

Exception states

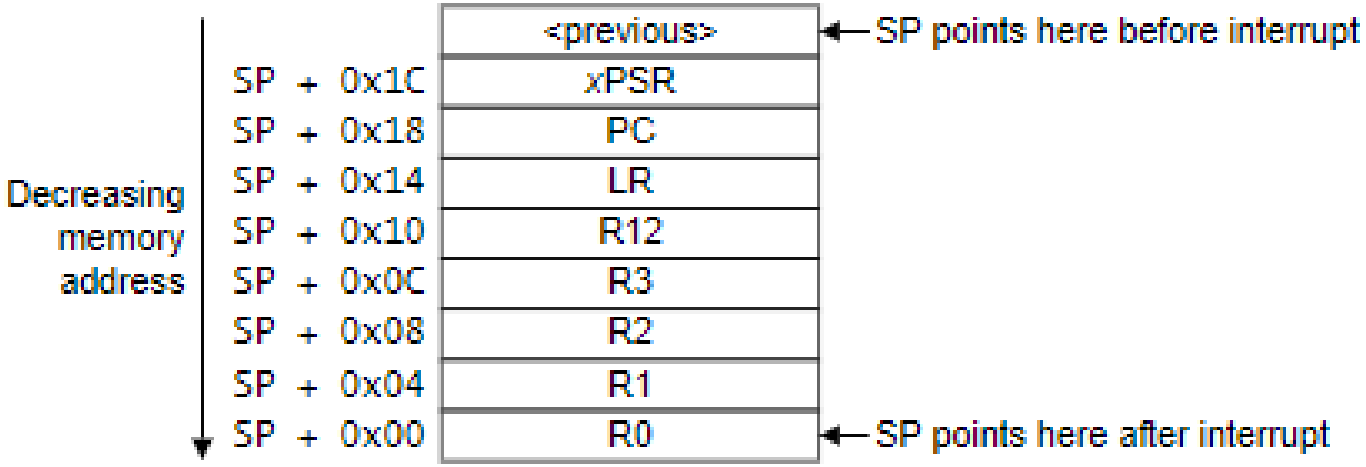
Each exception is in one of the following states:

- Inactive
  - The exception is not active and not pending.
- Active
  - An exception that is being serviced by the processor but has not completed.
- Pending
  - The exception is waiting to be serviced by the processor.

- Note
  - An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.
  - An exception handler can interrupt the execution of another exception handler: in this case both exceptions are in the active state.

Stack Frame

- When the processor takes an exception, the processor pushes information onto the current stack.
- This operation is referred to as *stacking* and the structure of eight data words is referred to as the *stack frame*. The stack frame contains the following information:



Exception Handler

- The default handlers are declared as weak symbols to allow the application writer to install their own handler simply by implementing a function with the correct name.
- If an interrupt occurs for which the application writer has not provided their own handler then the default handler will execute.
- Default interrupt handlers are typically implemented as an infinite loop.
  - If an application ends up in such a default handler it is first necessary to determine which interrupt is actually executing.

```
SVC_Handler     PROC
                EXPORT  SVC_Handler                 [WEAK]
                B       .
                ENDP
```

SoftWare Interrupts (SWI)

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	1	0x00000004
NMI	2	0x00000008
Hard fault	3	0x0000000C
Memory management fault	4	0x00000010
Bus fault	5	0x00000014
Usage fault	6	0x00000018
SVcall	**11**	**0x0000002C**
Debug monitor	12	0x00000030
PendSV	14	0x00000038
SysTick	15	0x0000003C
Interrupts	≥16	≥0x00000040

Supervisor Calls (SVC) is related to the 11th element of the IVT

Supervisor Calls (SVC) sintax

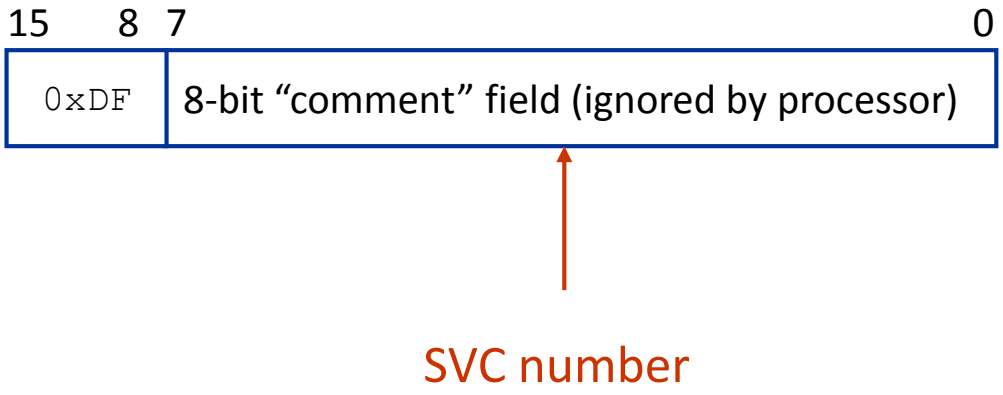
- Supervisor calls are normally used to request privileged operations or access to system resources from an operating system.
- As with previous ARM cores there is an instruction, SVC (formerly SWI) that generates a supervisor call.

```
{label} SVC immediate
```

- The SVC instruction has a number embedded within it, often referred to as the SVC number.
  - This is used to indicate what the caller is requesting
  - Is an expression evaluating to an integer in the range 0-255 (8-bit value).

Supervisor Calls (SVC) sintax (II)

- The SVC instruction has a number embedded within it, often referred to as the SVC number.
- This is sometimes used to indicate what the caller is requesting.



SVC number extraction

- On previous ARM cores you had to extract the SVC number from the instruction using the return address in the link register, and the other SVC arguments were already available in R0 through R3.
- On the Cortex-M3,
  - the core saves the argument registers to the stack on the initial exception entry
  - Any return value must also be passed back to the caller by modifying the stacked register values.
  - In order to do this, a short piece of assembly code must be implemented as the start of the SVC handler.
    - To identify which stack the registers were saved to
    - To extract the SVC number from the instruction.

SVC number extraction

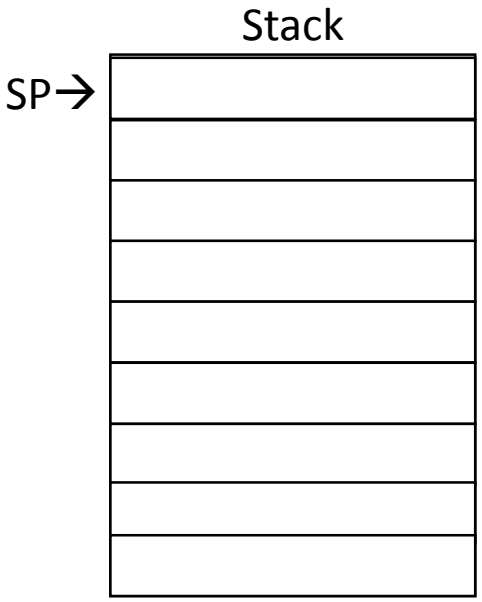
- On previous ARM core
  from the instruction u
  register, and the other
  in R0 through R3.
- On the Cortex-M3,
  - the core saves the argument registers to the stack on the initial exception entry
  - Any return value must also be passed back to the caller by modifying the stacked register values.
  - In order to do this, a short piece of assembly code must be implemented as the start of the SVC handler.
    - To identify which stack the registers were saved to
    - To extract the SVC number from the instruction.

A late-arriving exception, taken before the first instruction of the SVC handler executes, might corrupt the copy of the arguments still held in R0 to R3. This means that the stack copy of the arguments must be used by the SVC handler.

SVC number extraction

- On previous ARM cores you had to extract the SVC number from the instruction using the return address in the link register, and the other SVC arguments were already available in R0 through R3.
- On the Cortex-M3,
  - the core saves the argument registers to the stack on the initial exception entry
  - any return value must also be passed back to the caller by modifying the stacked register values.
  - a short piece of assembly code must be implemented as the SVC handler:
    - To identify which stack the registers were saved to
    - To extract the SVC number from the instruction.

What Happens on an SVC? (I)



1. SVC instruction execution : `PC <- [0x2C]`

SVC *#imm*

0x00000004	(1) Reset
0x00000008	(2) NMI
0x0000000C	(3) Hard fault
0x00000010	(4) Memory fault
0x00000014	(5) Bus fault
0x00000018	(6) Usage fault
0x0000002C	(11) **SVcall**
0x00000030	(12) Debug monitor
0x00000038	(14) PendSV
0x0000003C	(15) SysTick
≥0x00000040	(≥ 16) Interrupts

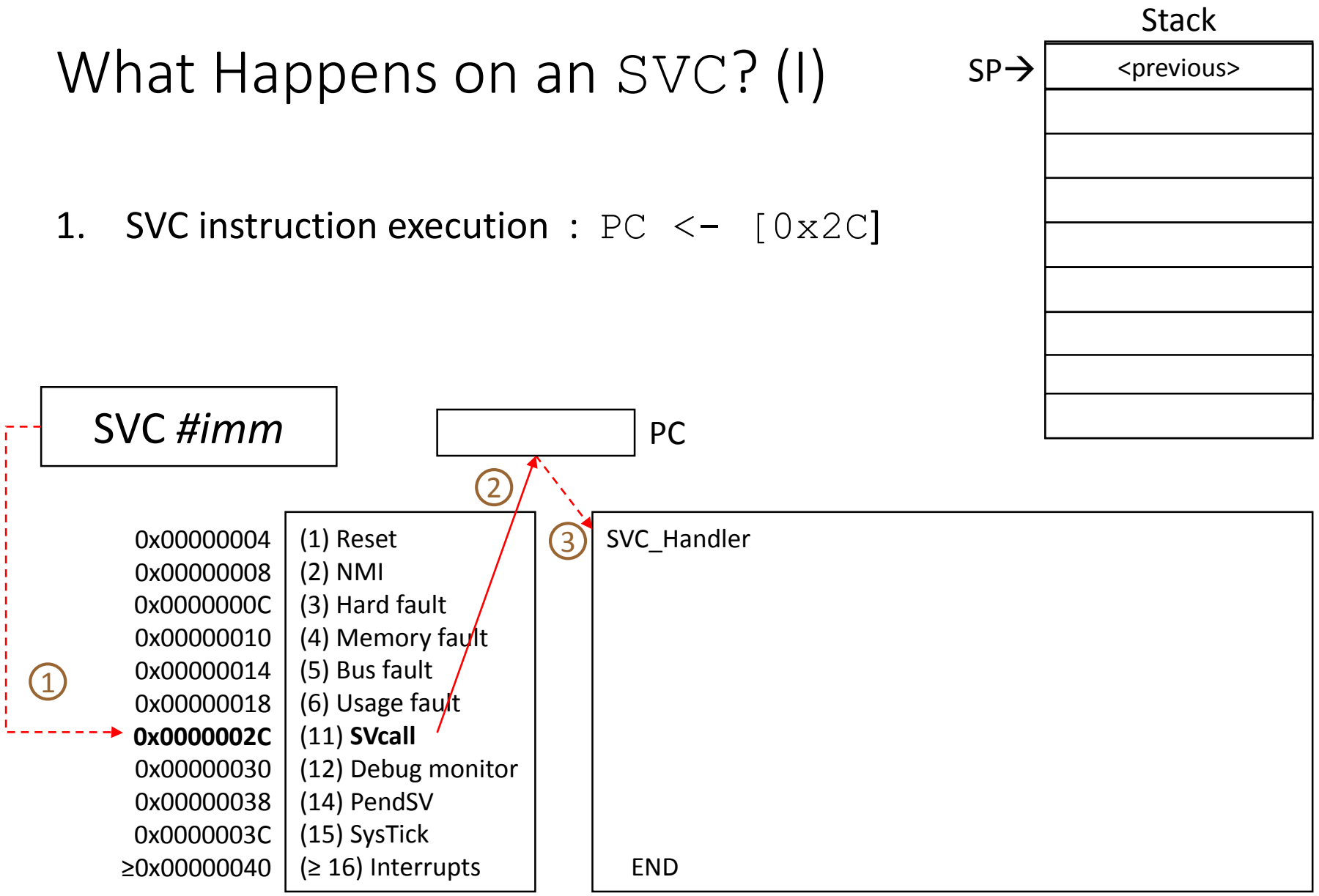
```
SVC_Handler




   END
```

What Happens on an SVC? (I)

1. SVC instruction execution : `PC <- [0x2C]`

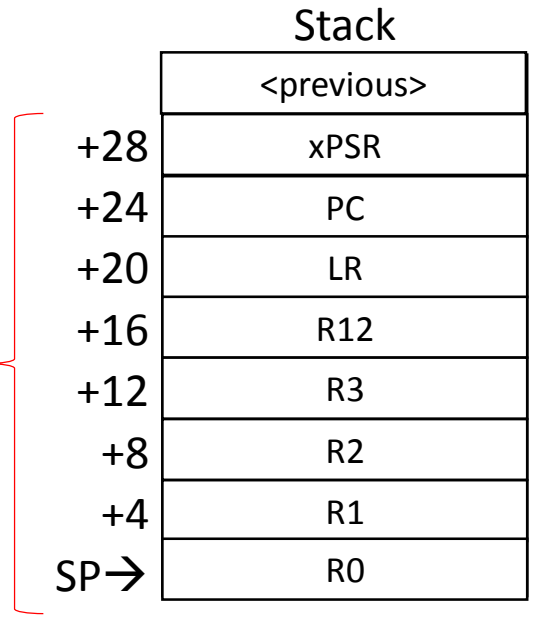


What Happens on an SVC? (II)

1. SVC instruction execution : `PC <- [0x2C]`
2. Stack Frame is saved
3. LR is updated

SVC *#imm*

EXC_RETURN LR



0x00000004	(1) Reset
0x00000008	(2) NMI
0x0000000C	(3) Hard fault
0x00000010	(4) Memory fault
0x00000014	(5) Bus fault
0x00000018	(6) Usage fault
0x0000002C	(11) **SVcall**
0x00000030	(12) Debug monitor
0x00000038	(14) PendSV
0x0000003C	(15) SysTick
≥0x00000040	(≥ 16) Interrupts

```
SVC_Handler




END
```

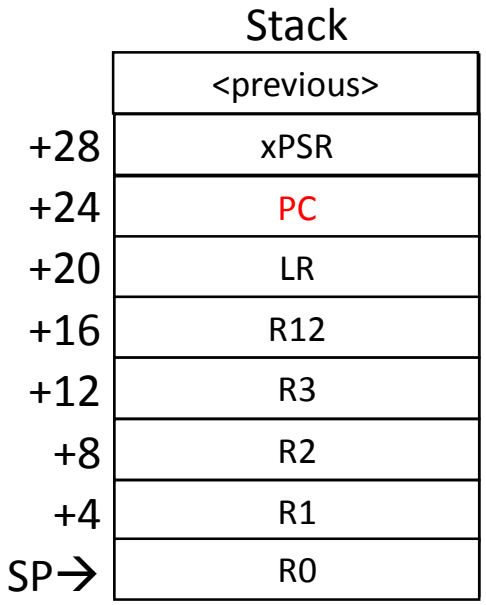
Exception return (I)

- The processor saves an EXC_RETURN value to the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler. Bits[31:4] of an EXC_RETURN value are 0xFFFFFFF.

EXC_RETURN	Description
0xFFFFFFF1	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode. Exception Return get state from the main stack. Execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode. Exception return gets state from the process stack. Execution uses PSP after return.
All other values	Reserved.

What Happens on an SVC? (III)

3. The stack frame includes the return address, that is the address of the next instruction in the interrupted program



SVC *#imm*

0x00000004	(1) Reset
0x00000008	(2) NMI
0x0000000C	(3) Hard fault
0x00000010	(4) Memory fault
0x00000014	(5) Bus fault
0x00000018	(6) Usage fault
0x0000002C	(11) **SVcall**
0x00000030	(12) Debug monitor
0x00000038	(14) PendSV
0x0000003C	(15) SysTick
≥0x00000040	(≥ 16) Interrupts

```
SVC_Handler
        LDR R0, [SP, #24]




END
```

What Happens on an SVC? (III)

4. Use the return address to retrieve the instruction code

Stack	
	<previous>
+28	xPSR
+24	PC
+20	LR
+16	R12
+12	R3
+8	R2
+4	R1
SP→	R0

SVC *#imm*

[R0-4] [R0] SVC *#imm* *next*

0x00000004	(1) Reset
0x00000008	(2) NMI
0x0000000C	(3) Hard fault
0x00000010	(4) Memory fault
0x00000014	(5) Bus fault
0x00000018	(6) Usage fault
0x0000002C	(11) **SVcall**
0x00000030	(12) Debug monitor
0x00000038	(14) PendSV
0x0000003C	(15) SysTick
≥0x00000040	(≥ 16) Interrupts

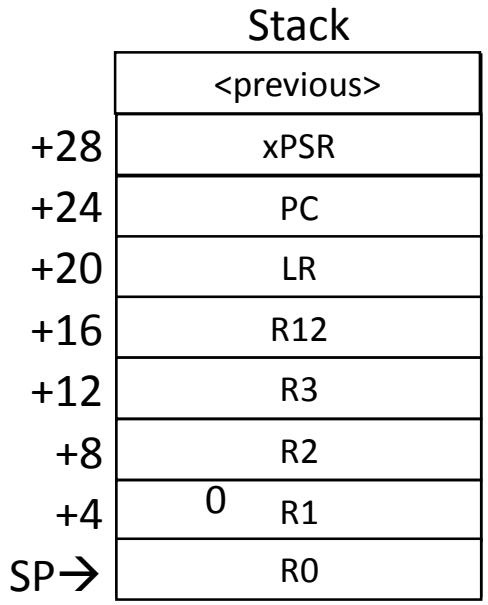
```
SVC_Handler
        LDR R0, [SP, #24]
        LDR R1, [R0,#-4]




END
```

What Happens on an SVC? (III)

5. Elaborate the SVC code using logical instructions



SVC *#imm*

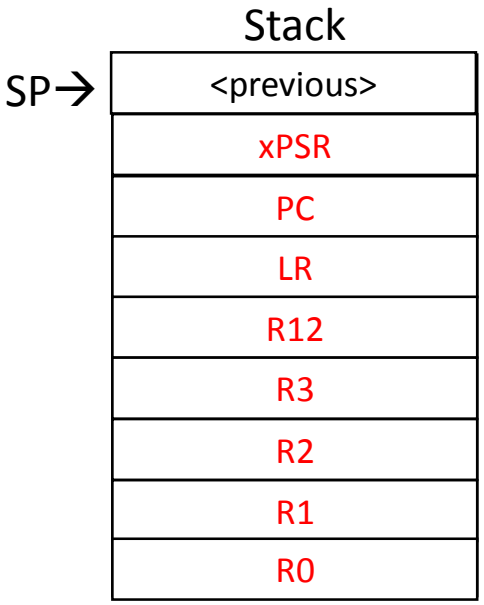
0x00000004	(1) Reset
0x00000008	(2) NMI
0x0000000C	(3) Hard fault
0x00000010	(4) Memory fault
0x00000014	(5) Bus fault
0x00000018	(6) Usage fault
0x0000002C	(11) **SVcall**
0x00000030	(12) Debug monitor
0x00000038	(14) PendSV
0x0000003C	(15) SysTick
≥0x00000040	(≥ 16) Interrupts

```
SVC_Handler
        LDR R0, [SP, #24]
        LDR R1, [R0,#-4]
        BIC R1, 0xFF000000
        LSR R1, #16



END
```

What Happens on an SVC? (III)



6. Return exploiting the EXC_RETURN value in LR
7. Registers value is restored

SVC *#imm*

EXC_RETURN LR

0x00000004	(1) Reset
0x00000008	(2) NMI
0x0000000C	(3) Hard fault
0x00000010	(4) Memory fault
0x00000014	(5) Bus fault
0x00000018	(6) Usage fault
0x0000002C	(11) **SVcall**
0x00000030	(12) Debug monitor
0x00000038	(14) PendSV
0x0000003C	(15) SysTick
≥0x00000040	(≥ 16) Interrupts

```
SVC_Handler
        LDR R0, [SP, #24]
        LDR R1, [R0,#-4]
        BIC R1, 0xFF000000
        LSR R1, #16

        BX LR
END
```

Exception return (II)

- When the processor loads a value matching an EXC_RETURN pattern to the PC it detects that the operation is a not a normal branch operation and, instead, that the exception is complete.
- Therefore, it starts the exception return sequence.
- Bits[3:0] of the EXC_RETURN value indicate the required return stack and processor mode.

EXC_RETURN	Description
0xFFFFFFF1	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode. Exception Return get state from the main stack. Execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode. Exception return gets state from the process stack. Execution uses PSP after return.
All other values	Reserved.

Enter a processor specific configuration

The following instruction enables updating special purpose registers when at priviledged level

```
MSR{cond} spec_reg, Rn
```

where:

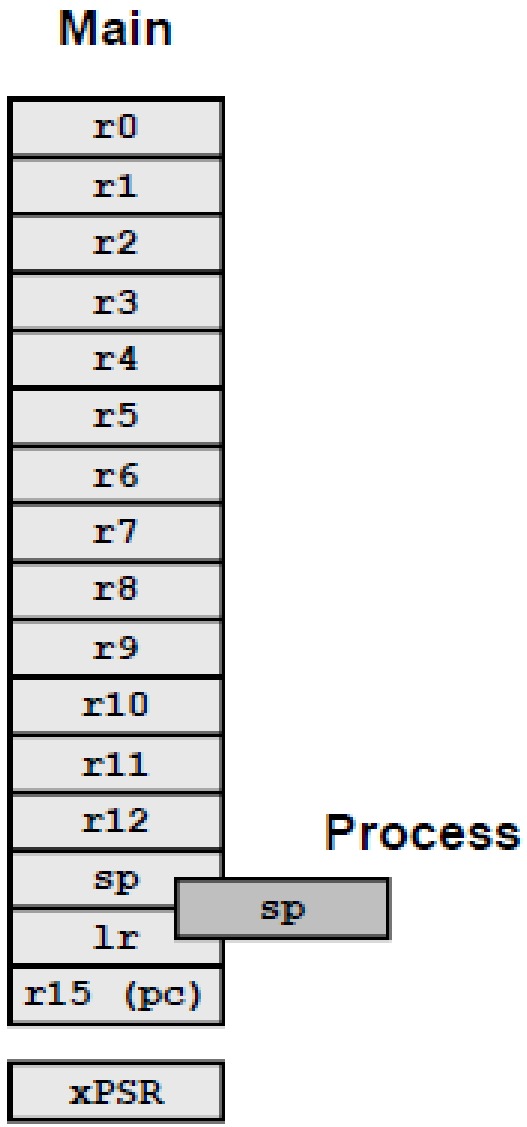
- *cond* is an optional condition code, see Conditional execution.
- Rn specifies the source register.
- spec_reg can be any of: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, or CONTROL

Processor operating modes and levels

- Two operating modes:
  - thread mode: on reset or after an exception
  - handler mode: when an exception occurs
- Two access levels:
  - user level: limited access to resources
  - privileged level: access to all resources
- Handler mode is always privileged.

CONTROL Register

- This register uses the following bits:
- **CONTROL**[2] [only Cortex-M4 and Cortex-M7]
  - =0 FPU not active
  - =1 FPU active
- **CONTROL**[1]
  - =0 In handler mode - MSP is selected. No alternate stack possible for handler mode.
  - =0 In thread mode - Default stack pointer MSP is used.
  - =1 In thread mode - Alternate stack pointer PSP is used.
- **CONTROL**[0] [not Cortex-M0]
  - =0 In thread mode and privileged state.
  - =1 In thread mode and user state.



Controlling processor modes and priviledges

- At RESET time, after required initializations, it is possible to setup a «user mode» processor status

```
MOV  R0, #3
MSR  CONTROL, R0
```

- This is bringing the system to
  - Unpriviledged,
  - Thread mode
  - Use of the Process Stack Pointer (PSP)
- Entering a Handling procedure is moving the system to
  - Priviledged
  - Handler mode
  - Use of the Master Stack Pointer (MSP)

An example

STACK segment

```
Stack_Size      EQU     0x00000200
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
                SPACE   Stack_Size/2
Stack_Process   SPACE   Stack_Size/2
__initial_sp
```

CALLER

```
MOV    R0, #3
MSR    CONTROL, R0
LDR    SP, =Stack_Process

SVC    0x10
```

HANDLER

```
STMFD SP!, {R0-R12, LR}
MRS R1, PSP
LDR R0, [R1, #24]
LDR R0, [R0,#-4]
BIC R0, #0xFF000000
LSR R0, #16
LDMFD SP!, {R0-R12, LR}
BX LR
```

An example

MSP assigned address which is retrieved from position 0 of the IVT

```
STACK segment
Stack_Size      EQU     0x00000200
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
                SPACE   Stack_Size/2
Stack_Process   SPACE   Stack_Size/2
__initial_sp
```

CALLER

```
MOV    R0, #3
MSR    CONTROL, R0
LDR    SP, =Stack_Process

SVC    0x10
```

HANDLER

```
STMFD SP!, {R0-R12, LR}
MRS R1, PSP
LDR R0, [R1, #24]
LDR R0, [R0,#-4]
BIC R0, #0xFF000000
LSR R0, #16
LDMFD SP!, {R0-R12, LR}
BX LR
```

An example

MSP assigned address which is retrieved from position 0 of the IVT

```
STACK segment
Stack_Size      EQU     0x00000200
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
                SPACE   Stack_Size/2
Stack_Process   SPACE   Stack_Size/2
__initial_sp
```

System is now Unpriviledged, Thread and using PSP

CALLER

```
MOV    R0, #3
MSR    CONTROL, R0
LDR    SP, =Stack_Process

SVC    0x10
```

HANDLER

```
STMFD SP!, {R0-R12, LR}
MRS R1, PSP
LDR R0, [R1, #24]
LDR R0, [R0,#-4]
BIC R0, #0xFF000000
LSR R0, #16
LDMFD SP!, {R0-R12, LR}
BX LR
```

An example

```
STACK segment
Stack_Size      EQU     0x00000200
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
                SPACE   Stack_Size/2
Stack_Process   SPACE   Stack_Size/2
__initial_sp
```

MSP assigned address which is retrieved from position 0 of the IVT

PSP address is assigned in the RESET handler

System is now Unpriviledged, Thread and using PSP

```
CALLER

MOV    R0, #3
MSR    CONTROL, R0
LDR    SP, =Stack_Process

SVC    0x10
```

```
HANDLER

STMFD SP!, {R0-R12, LR}
MRS R1, PSP
LDR R0, [R1, #24]
LDR R0, [R0,#-4]
BIC R0, #0xFF000000
LSR R0, #16
LDMFD SP!, {R0-R12, LR}
BX LR
```

An example

```
STACK segment
Stack_Size      EQU     0x00000200
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
                SPACE   Stack_Size/2
Stack_Process   SPACE   Stack_Size/2
__initial_sp
```

MSP assigned address which is retrieved from position 0 of the IVT

PSP address is assigned in the RESET handler

System is Unpriviledged, Thread and using PSP

```
CALLER

MOV    R0, #3
MSR    CONTROL, R0
LDR    SP, =Stack_Process

SVC    0x10
```

SVC causes the change of mode, level and Stack Pointer

```
HANDLER

STMFD SP!, {R0-R12, LR}
MRS R1, PSP
LDR R0, [R1, #24]
LDR R0, [R0,#-4]
BIC R0, #0xFF000000
LSR R0, #16
LDMFD SP!, {R0-R12, LR}
BX LR
```

An example

```
STACK segment
Stack_Size      EQU     0x00000200
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
                SPACE   Stack_Size/2
Stack_Process   SPACE   Stack_Size/2
__initial_sp
```

MSP assigned address which is retrieved from position 0 of the IVT

PSP address is assigned in the RESET handler

```
CALLER

MOV     R0, #3
MSR     CONTROL, R0
LDR     SP, =Stack_Process

SVC     0x10
```

System is Unpriviledged, Thread and using PSP

SVC causes the change of mode, level and Stack Pointer

```
HANDLER

STMFD SP!, {R0-R12, LR}
MRS R1, PSP
LDR R0, [R1, #24]
LDR R0, [R0,#-4]
BIC R0, #0xFF000000
LSR R0, #16
LDMFD SP!, {R0-R12, LR}
BX LR
```

Priviledged, Handler and using MSP when entered

An example

```
STACK segment
Stack_Size      EQU     0x00000200
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
                SPACE   Stack_Size/2
Stack_Process   SPACE   Stack_Size/2
__initial_sp
```

MSP assigned address which is retrieved from position 0 of the IVT

PSP address is assigned in the RESET handler

```
CALLER

MOV    R0, #3
MSR    CONTROL, R0
LDR    SP, =Stack_Process

SVC    0x10
```

System is Unpriviledged, Thread and using PSP

SVC causes the change of mode, level and Stack Pointer

```
HANDLER

STMFD SP!, {R0-R12, LR}
MRS R1, PSP
LDR R0, [R1, #24]
LDR R0, [R0,#-4]
BIC R0, #0xFF000000
LSR R0, #16
LDMFD SP!, {R0-R12, LR}
BX LR
```

Priviledged, Handler and using MSP when entered

LR is showing an extraordinary value that trigger the handler exit