Interrupts (1)

Lecture 9

Yeongpil Cho

Hanynag University

Topics

- Interrupt Handling Basics
- Priority Management

Interrupt Handling Basics

Interrupts

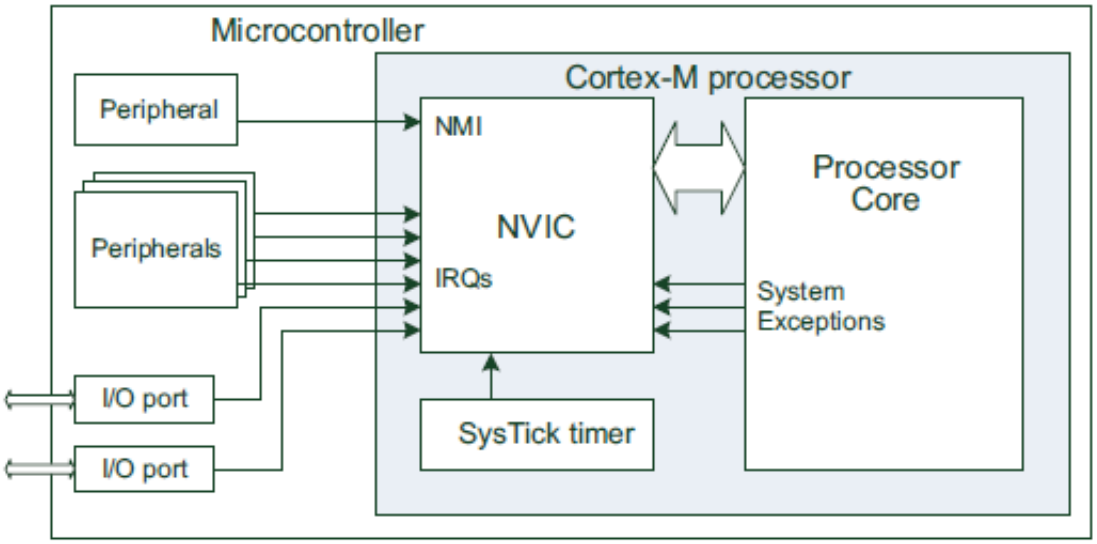
- Motivations
  - Inform processors of some external events timely
    - Polling *vs* Interrupt
  - Implement multi-tasking with priority support

Polling vs Interrupt

- Polling:
  - You pick up the phone every three seconds to check whether you are getting a call.
- Interrupt:
  - Do whatever you should do and pick up the phone when it rings.

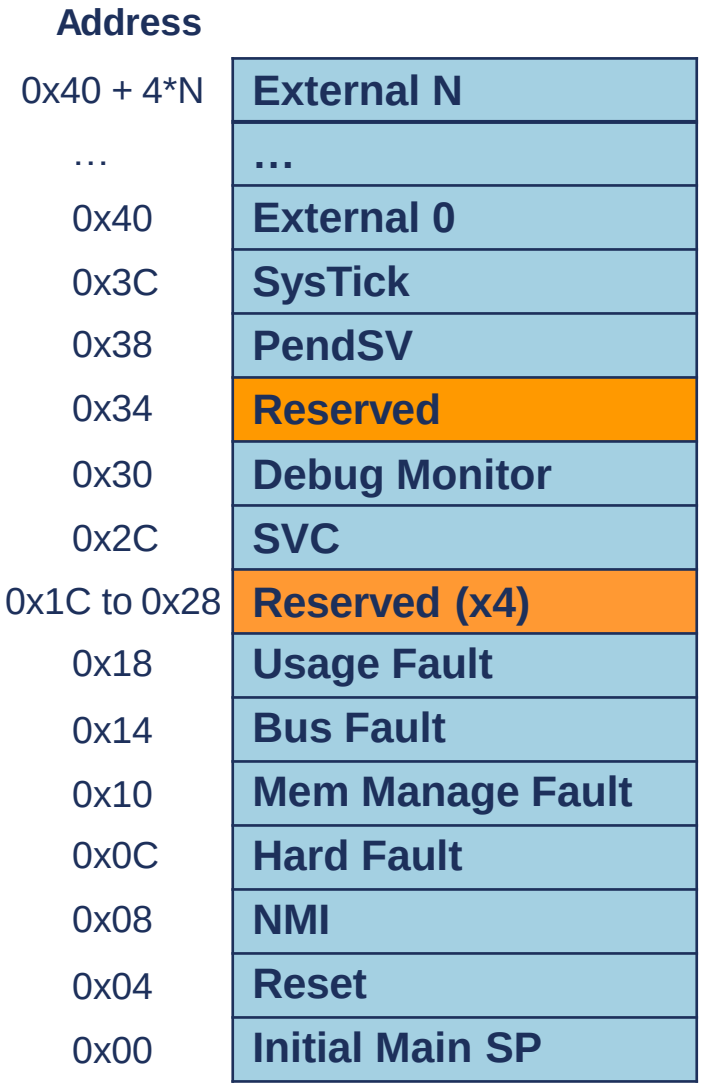
ARMv7-M Interrupt Handling

- One Non-Maskable Interrupt (NMI) supported
- Up to 512 (496 interrupts and 16 exceptions) prioritizable interrupts/exceptions supported
  - Interrupts can be masked
  - Implementation option selects number of interrupts supported
- Nested Vectored Interrupt Controller (NVIC) is tightly co upled with processor core



Interrupt Service Routine Vector Table

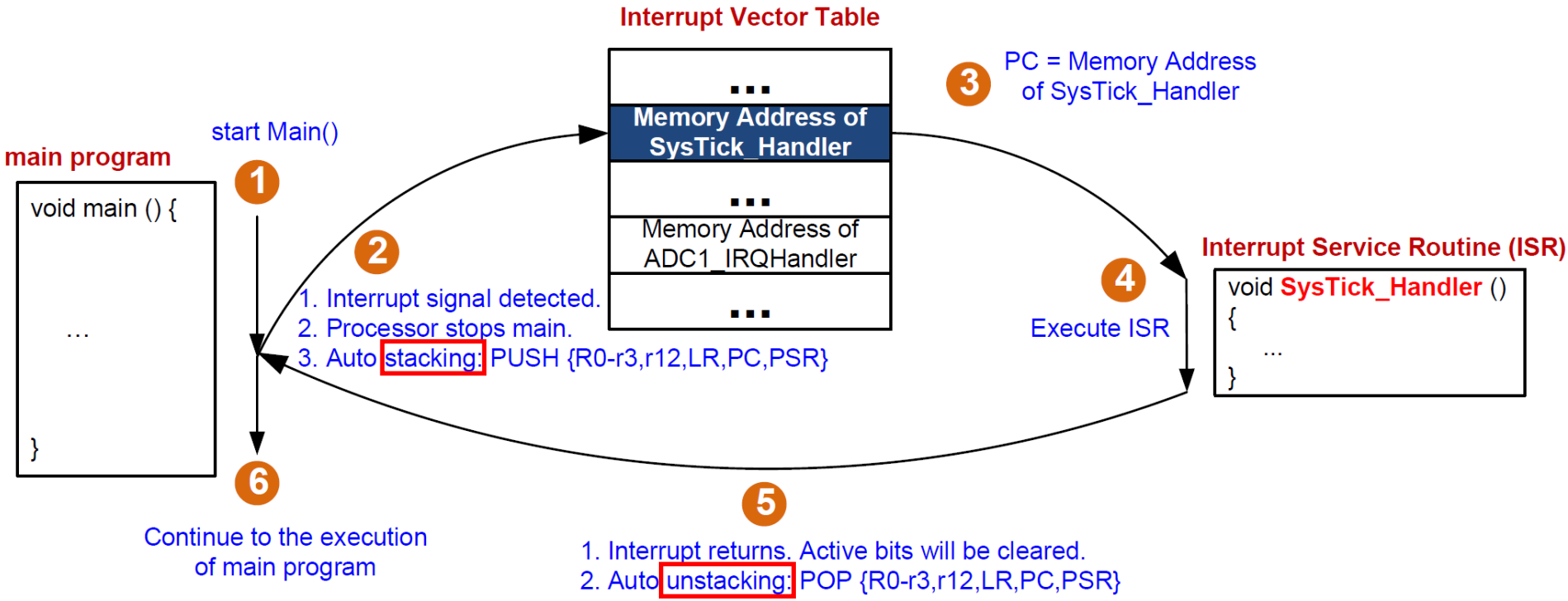
- First entry contains initial Main SP
- All other entries are addresses for
  - exception/interrupt handlers
  - Must always have LSBit = 1 (for Thumb)
- Table can be relocated
  - Use Vector Table Offset Register
  - Still require minimal table entries at 0x0 for booting the core
- Table can be generated using C code



Interrupt Service Routine Vector Table of Cortex-M processors

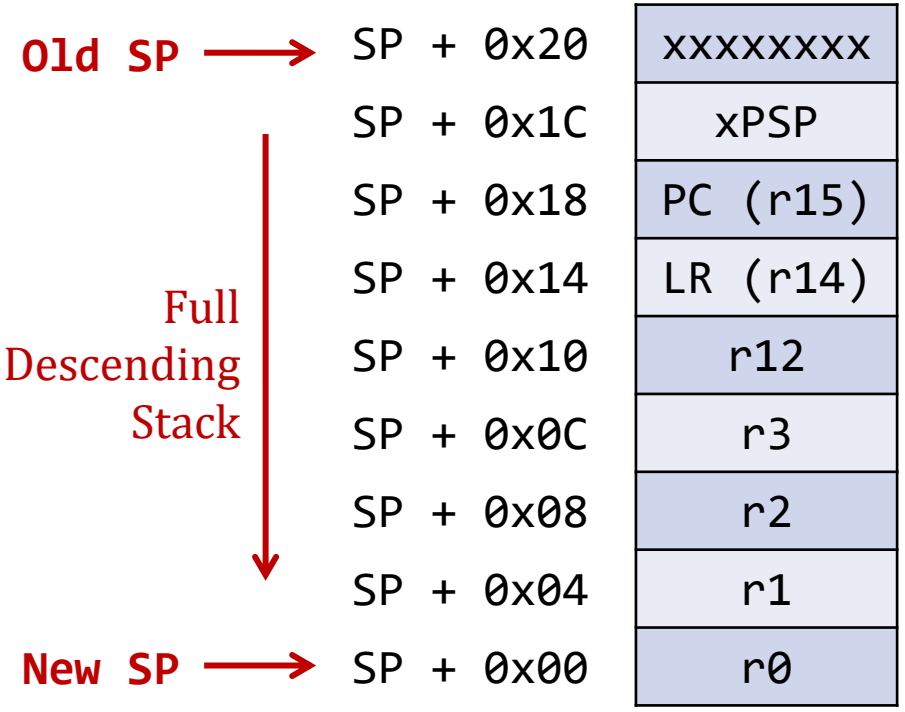
	Exception	Name	Priority	Descriptions
Fault Mode & Start-up Handlers	1	Reset	-3 (Highest)	Reset
	2	NMI	-2	Non-Maskable Interrupt
	3	Hard Fault	-1	Default fault if other hander not implemented
	4	MemManage Fault	Programmable	MPU violation or access to illegal locations
	5	Bus Fault	Programmable	Fault if AHB interface receives error
	6	Usage Fault	Programmable	Exceptions due to program errors
System Handlers	11	SVCall	Programmable	System SerVice call
	12	Debug Monitor	Programmable	Break points, watch points, external debug
	14	PendSV	Programmable	Pendable SerVice request for System Device
	15	Systick	Programmable	System Tick Timer
Custom Handlers	16	Interrupt #0	Programmable	External Interrupt #0
	...	...	...	...
	...	...	...	...
	...	...	...	...
	255	Interrupt #239	Programmable	External Interrupt #239

Interrupt Handling Process



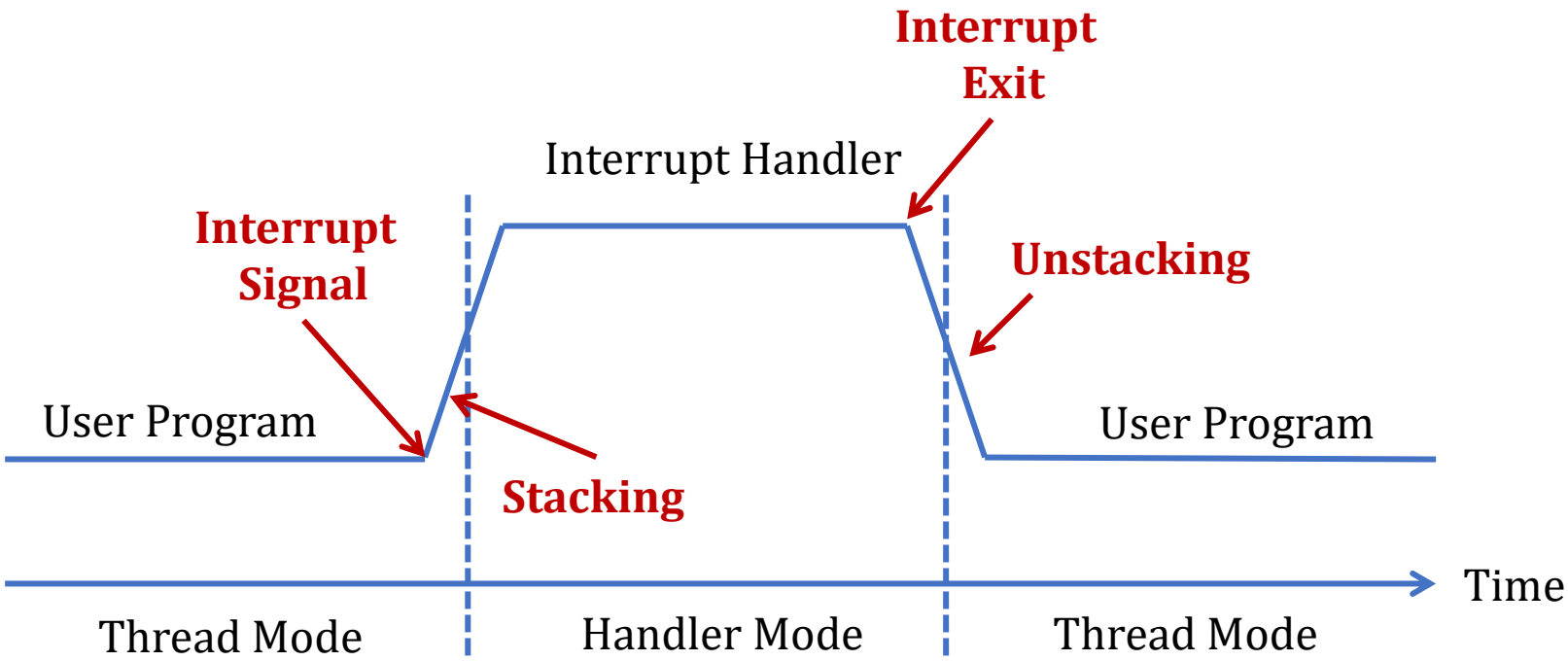
Stacking & Unstacking

- Current mode (either Thread mode or Handler mode)'s stack is used for stacking/unstacking.

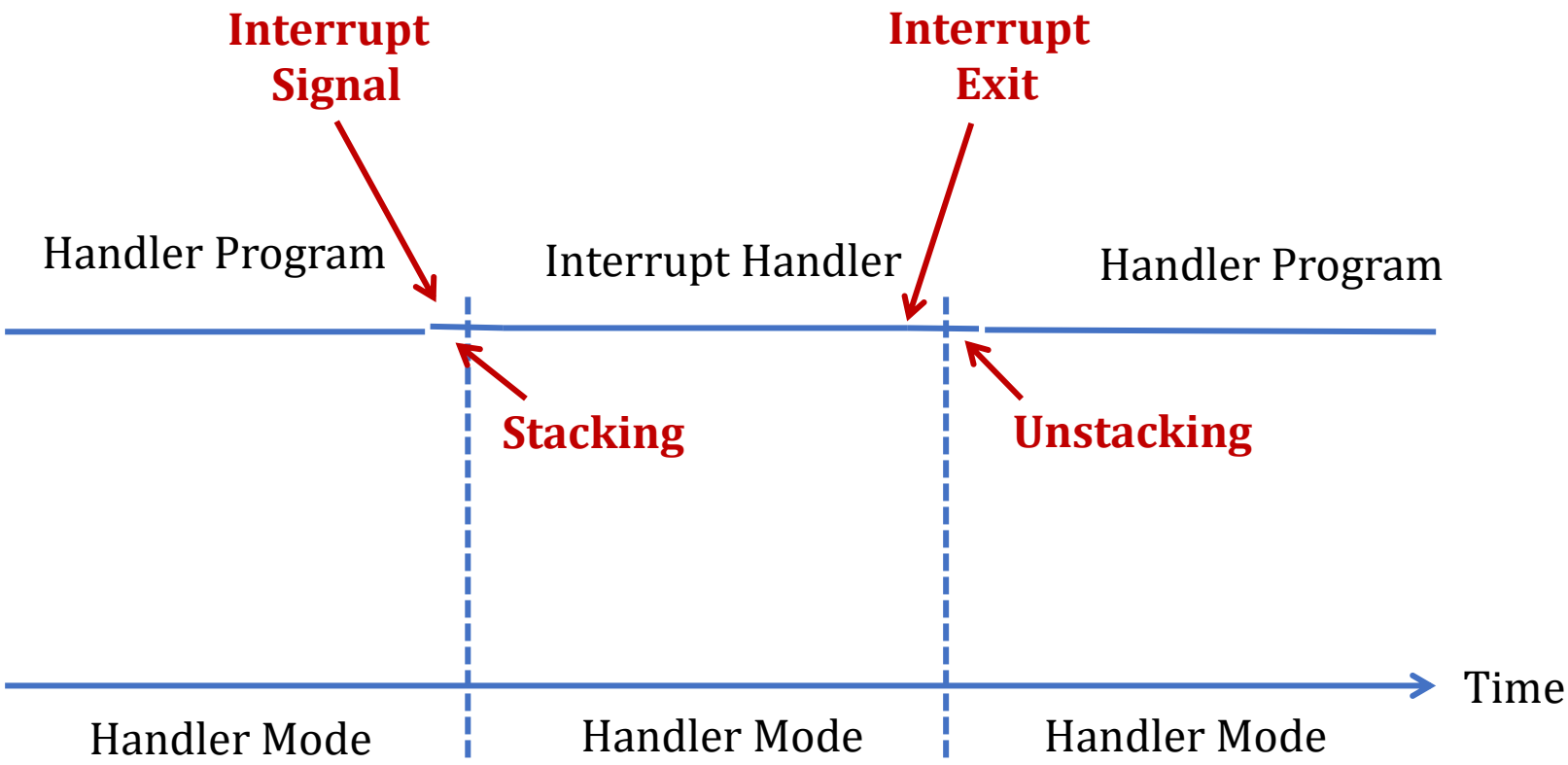


- **Stacking:** The processor automatically pushes these eight registers into the stack before an interrupt handler starts
- **Unstacking:** The processor automatically pops these eight register out of the stack when an interrupt hander exits.

Stacking & Unstacking



Stacking & Unstacking

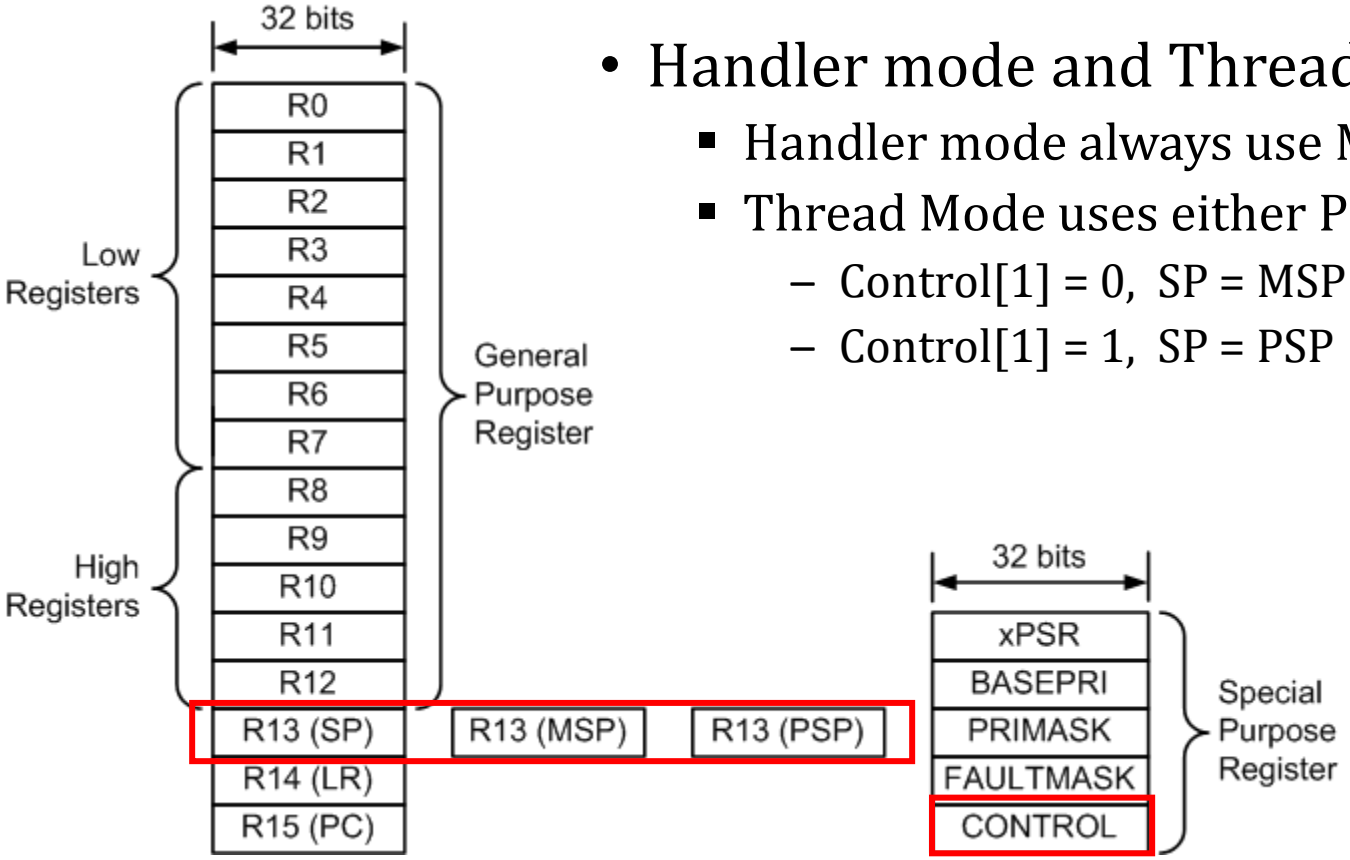


Exception Exits

- When the **EXC_RETURN** is loaded into the PC at the end of the exception handler execution, the processor performs an exception return sequence
- EXC_RETURN is generated and set to **LR** by processors when an exception arises.
- There are three ways to trigger the interrupt return sequence:

Return Instruction	Description
BX <*reg*>	If the EXC_RETURN value is still in LR, we can use the *BX LR* instruction to perform the interrupt return.
POP {PC}, or POP {...., PC}	Very often the value of LR is pushed to the stack after entering the exception handler. We can use the POP instruction, either a single POP or multiple POPs, to put the EXC_RETURN value to the program counter. This will cause the processor to perform the interrupt return.
LDR, or LDM	It is possible to produce an interrupt return using the LDR instruction with PC as the destination register.

Registers



- Handler mode and Thread mode
  - Handler mode always use MSP
  - Thread Mode uses either PSP or MSP
    - Control[1] = 0, SP = MSP (default)
    - Control[1] = 1, SP = PSP

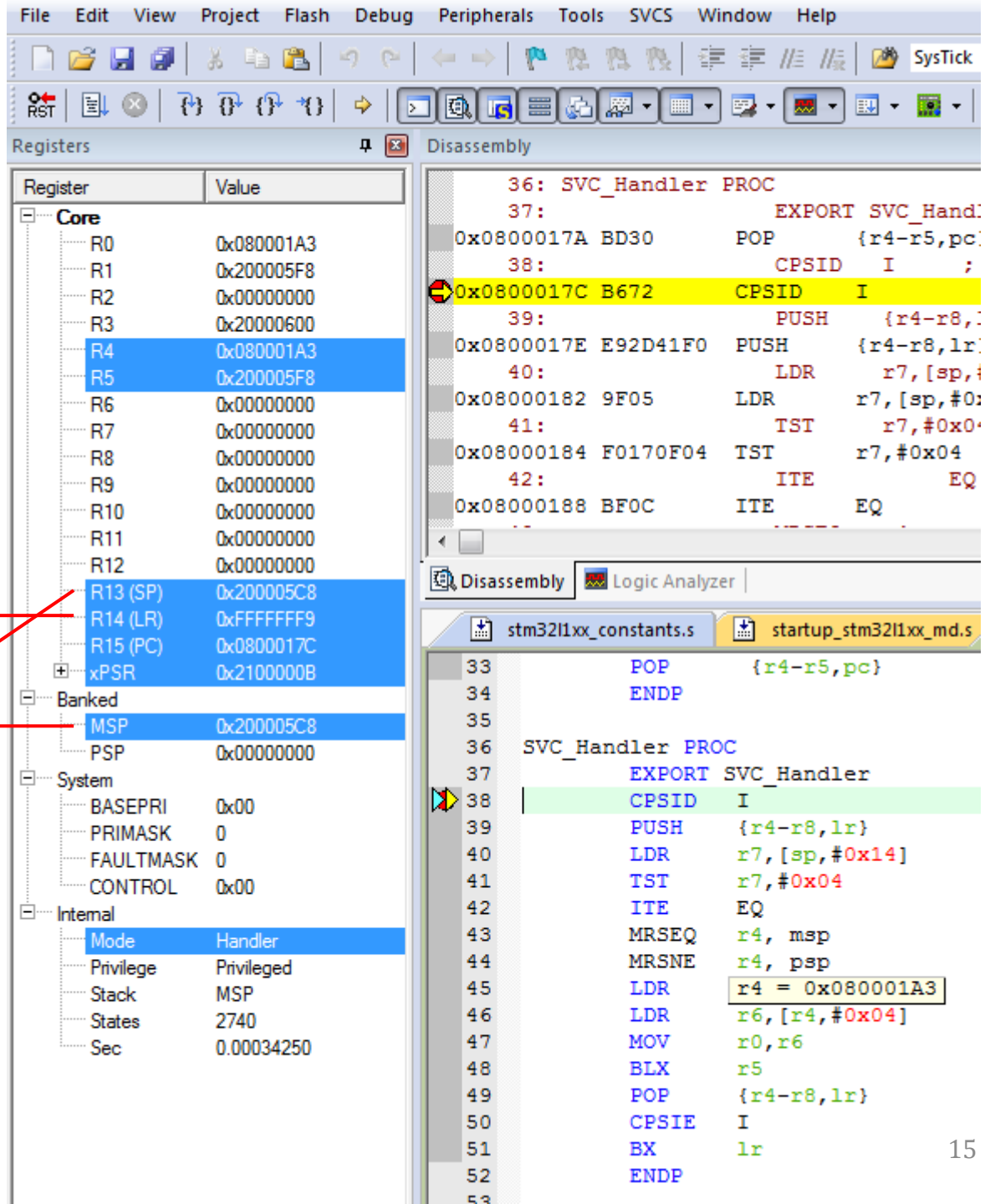
MSP: Main Stack Pointer
PSP: Process Stack Pointer

Register values in an interrupt service routine

LR = 0xFFFFFFF9

SP = MSP

ISR always in hander mode.



Which stack to use when exiting an interrupt?

- EXC_RETURN indicates processor mode and stack type to be activated when exiting an interrupt

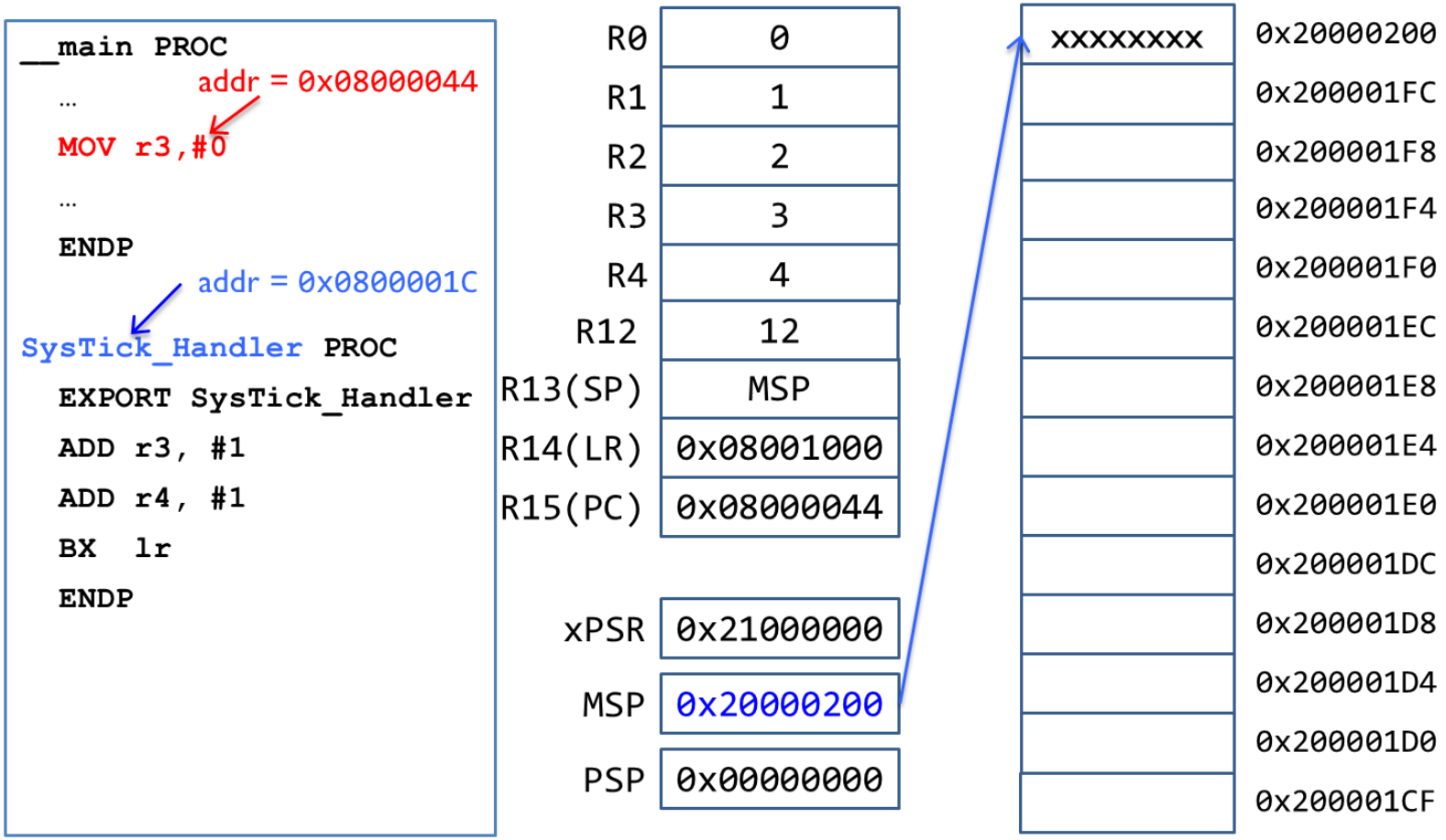
No FP extension:

EXC_RETURN	Return Mode	Return Stack
0xFFFFFFF1	Handler	SP = MSP
0xFFFFFFF9	Thread	SP = MSP
0xFFFFFFFD	Thread	SP = PSP

With FP extension:

EXC_RETURN	Return Mode	Return Stack
0xFFFFFFE1	Handler	SP = MSP
0xFFFFFFE9	Thread	SP = MSP
0xFFFFFFED	Thread	SP = PSP

Interrupt: Stacking & Unstacking



Interrupt: Stacking & Unstacking

Suppose SysTick interrupt occurs when PC = 0x08000044

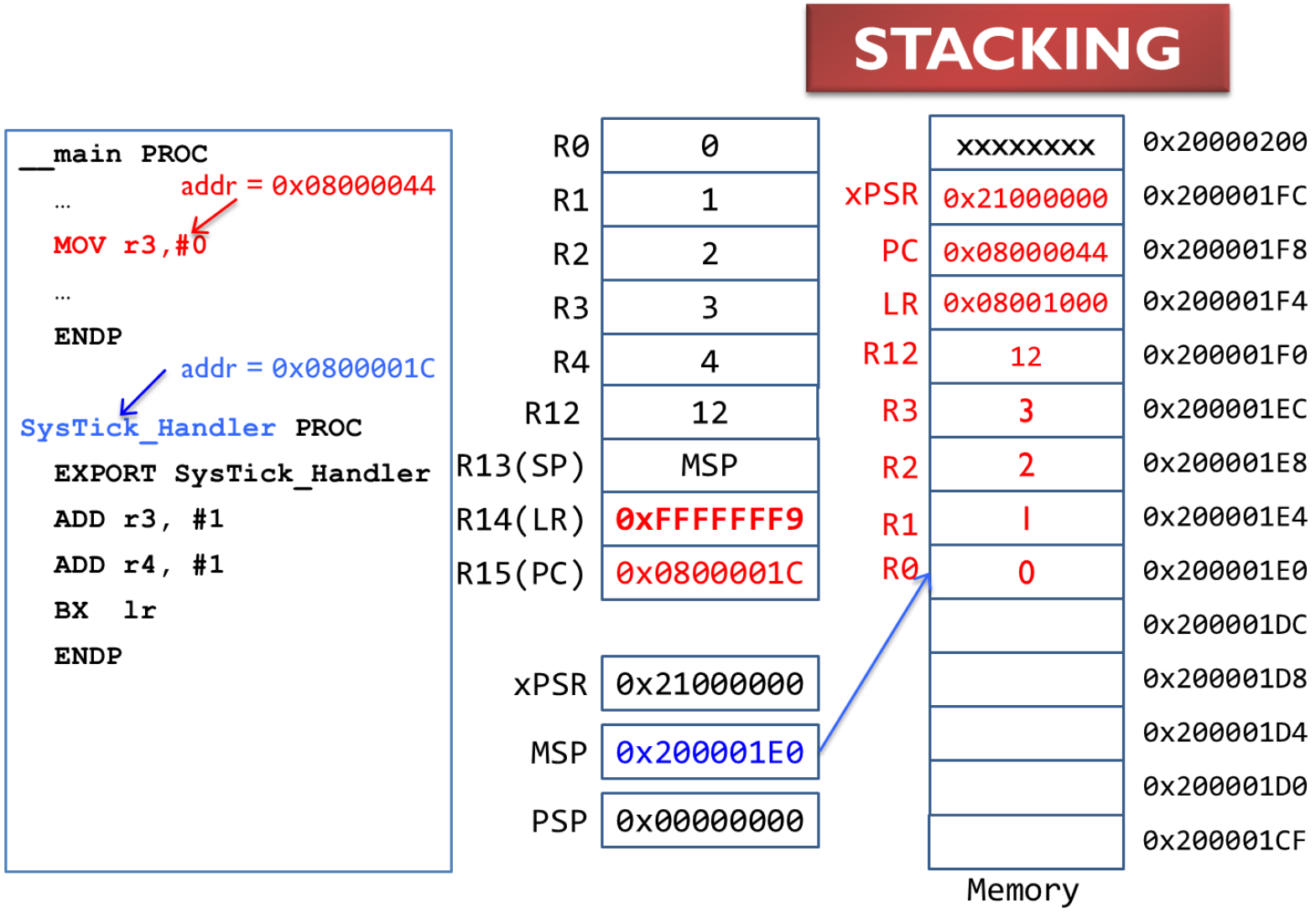
```
__main PROC
                addr = 0x08000044
   …
   MOV r3,#0
   …
   ENDP
            addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	MSP
R14(LR)	0x08001000
R15(PC)	0x08000044
xPSR	0x21000000
MSP	0x20000200
PSP	0x00000000

xxxxxxxx	0x20000200
	0x200001FC
	0x200001F8
	0x200001F4
	0x200001F0
	0x200001EC
	0x200001E8
	0x200001E4
	0x200001E0
	0x200001DC
	0x200001D8
	0x200001D4
	0x200001D0
	0x200001CF

Memory

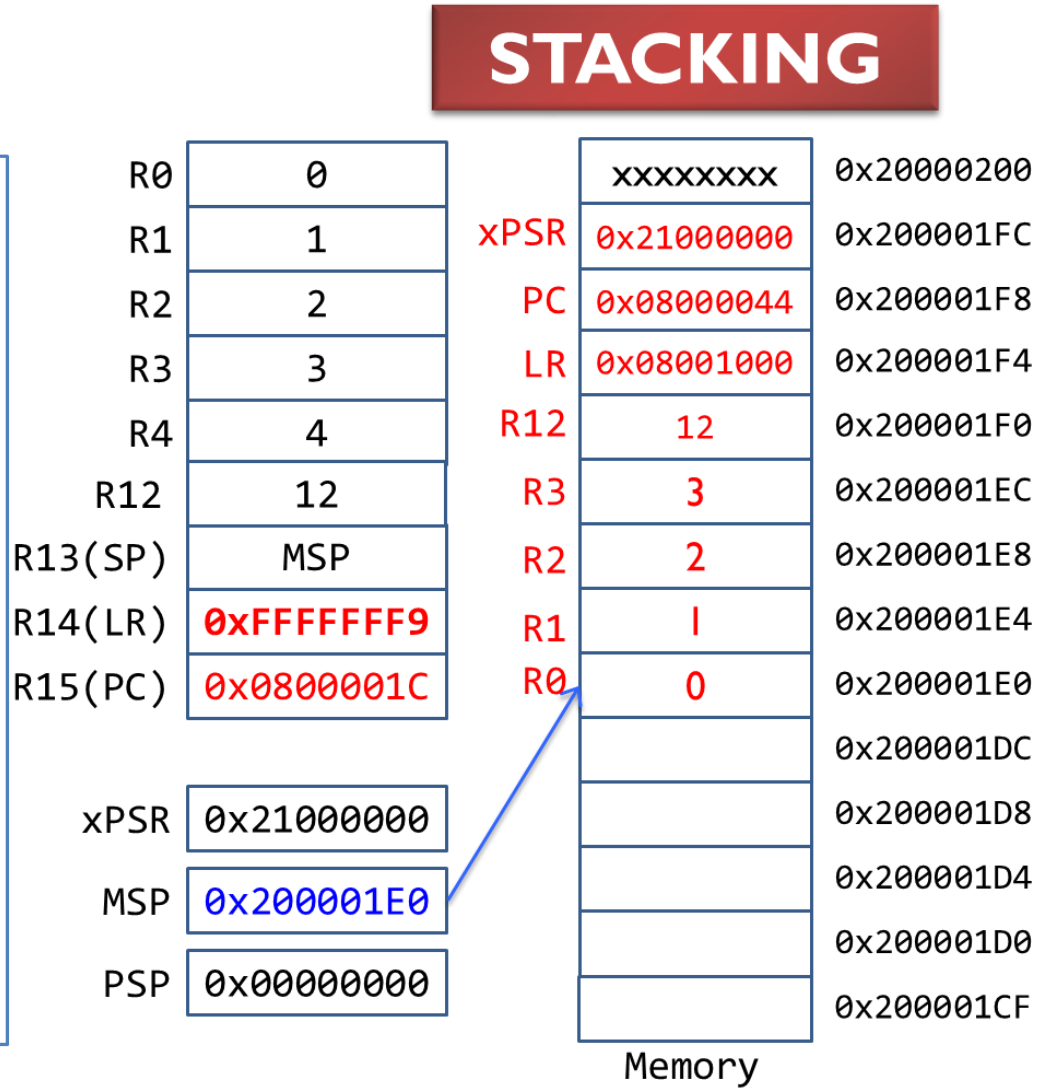
Interrupt: Stacking & Unstacking



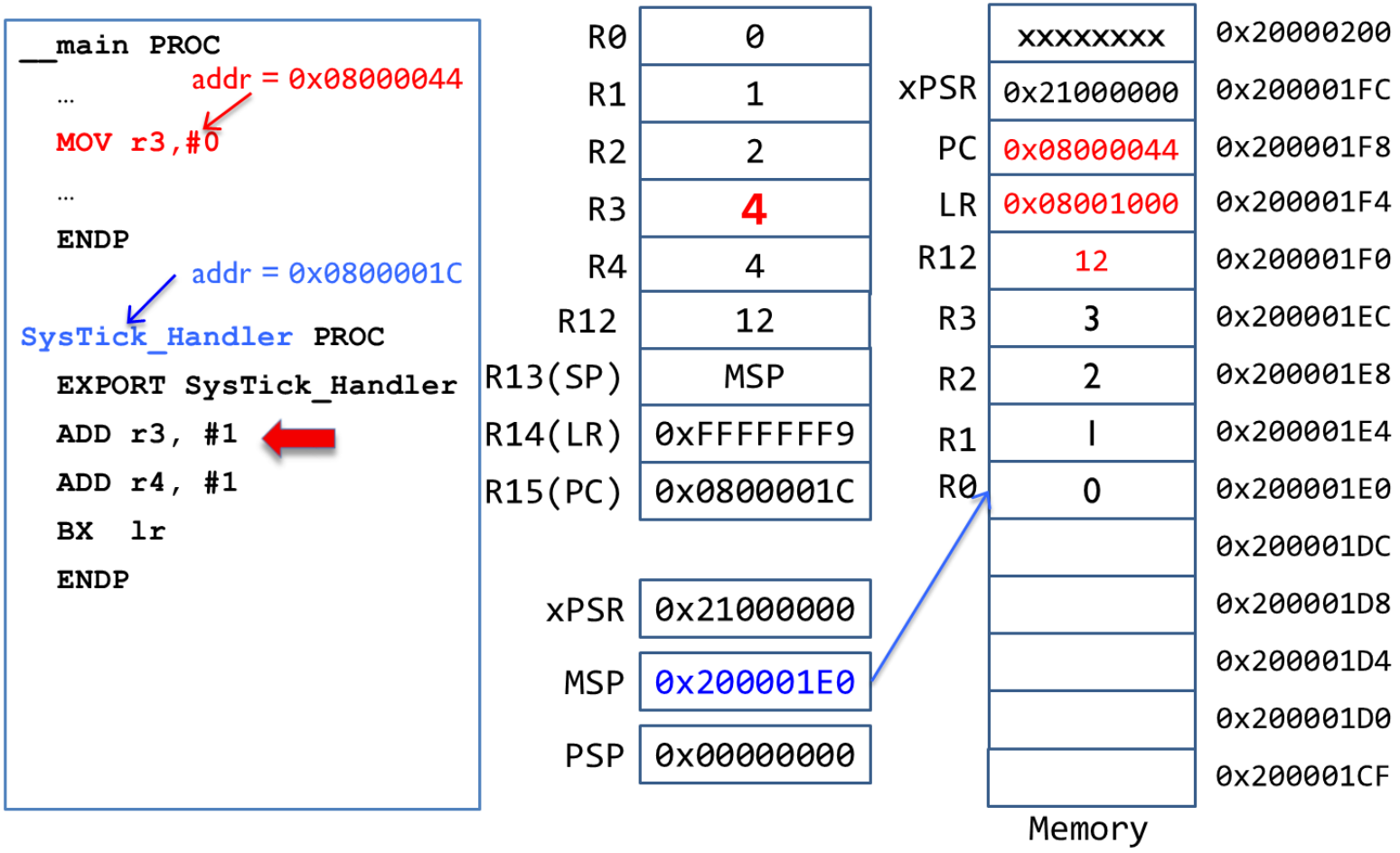
Interrupt: Stacking & Unstacking

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```

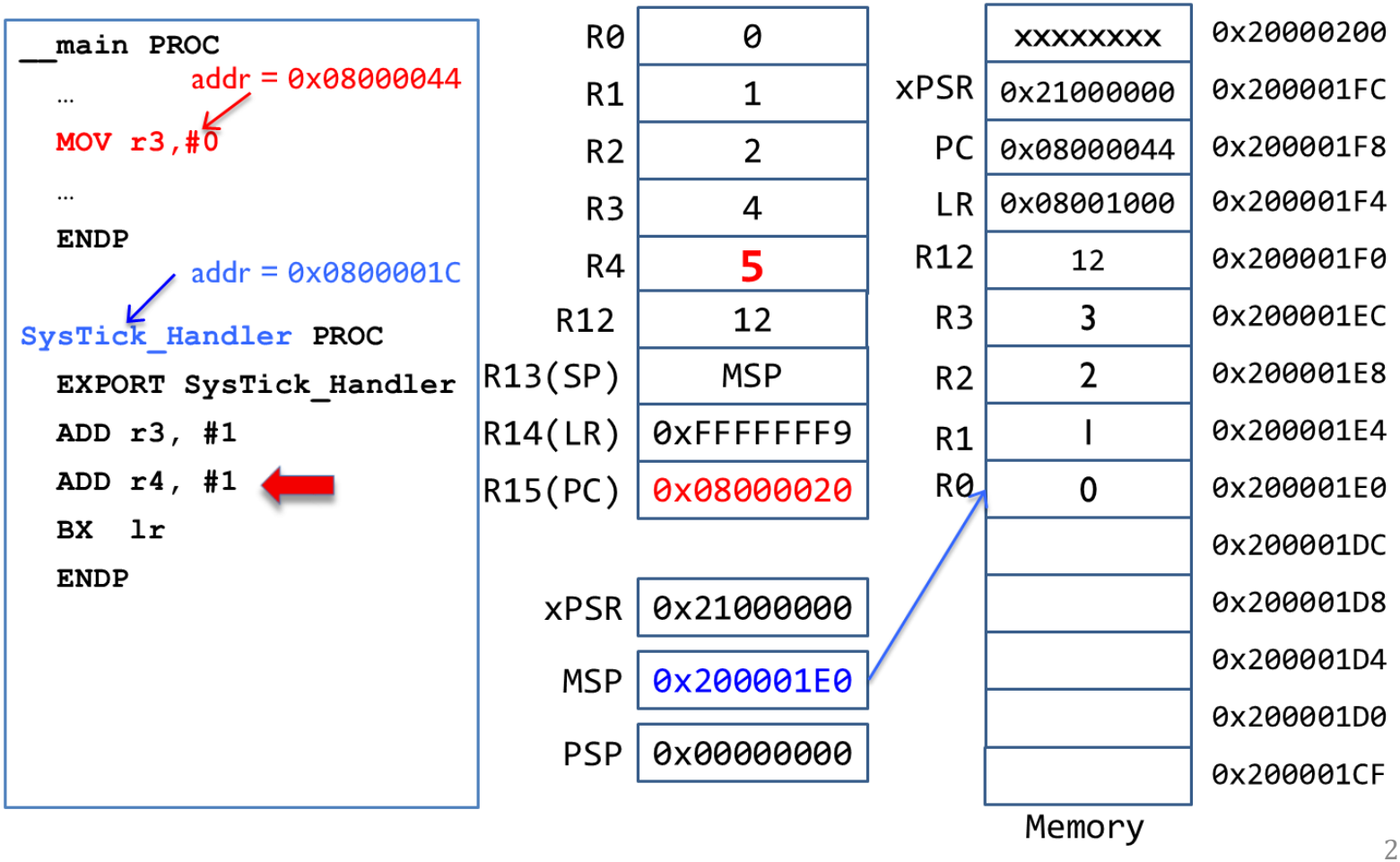
LR = 0xFFFFFFF9 to indicate MSP is used.



Interrupt: Stacking & Unstacking



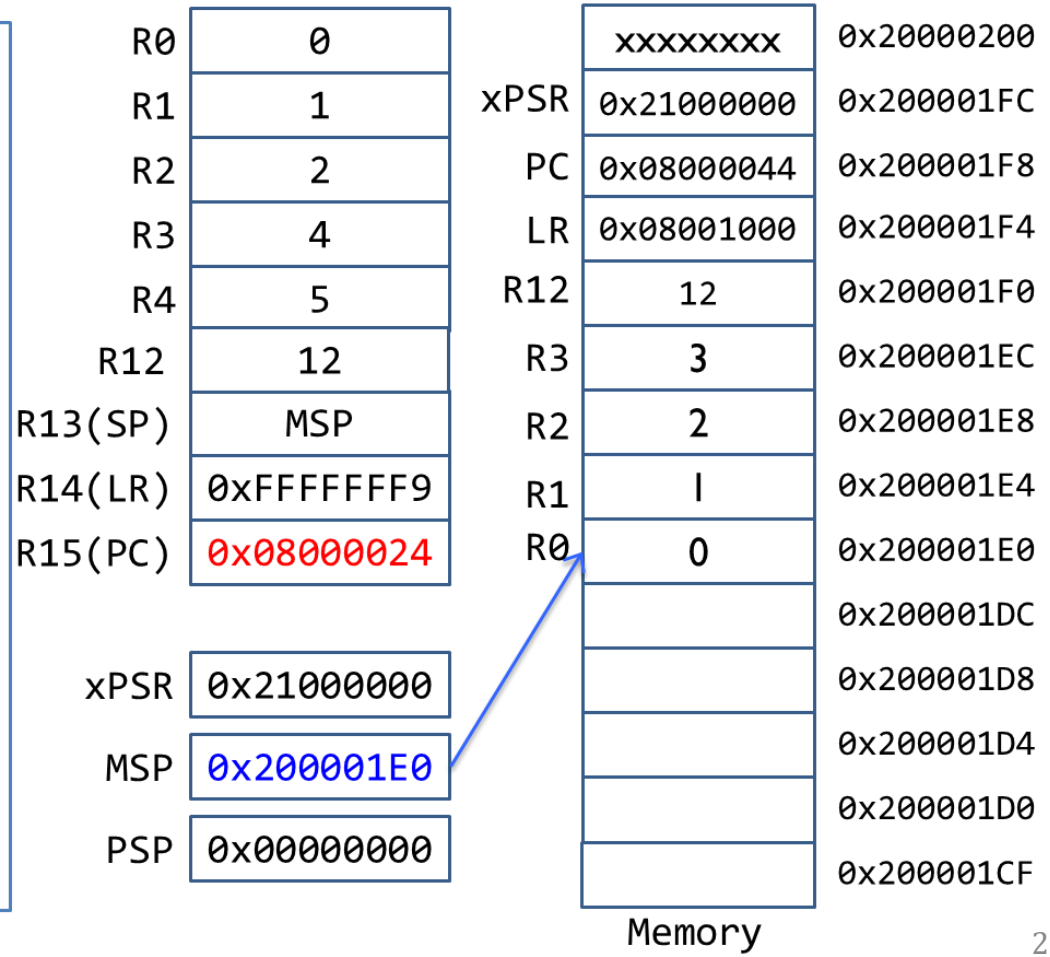
Interrupt: Stacking & Unstacking



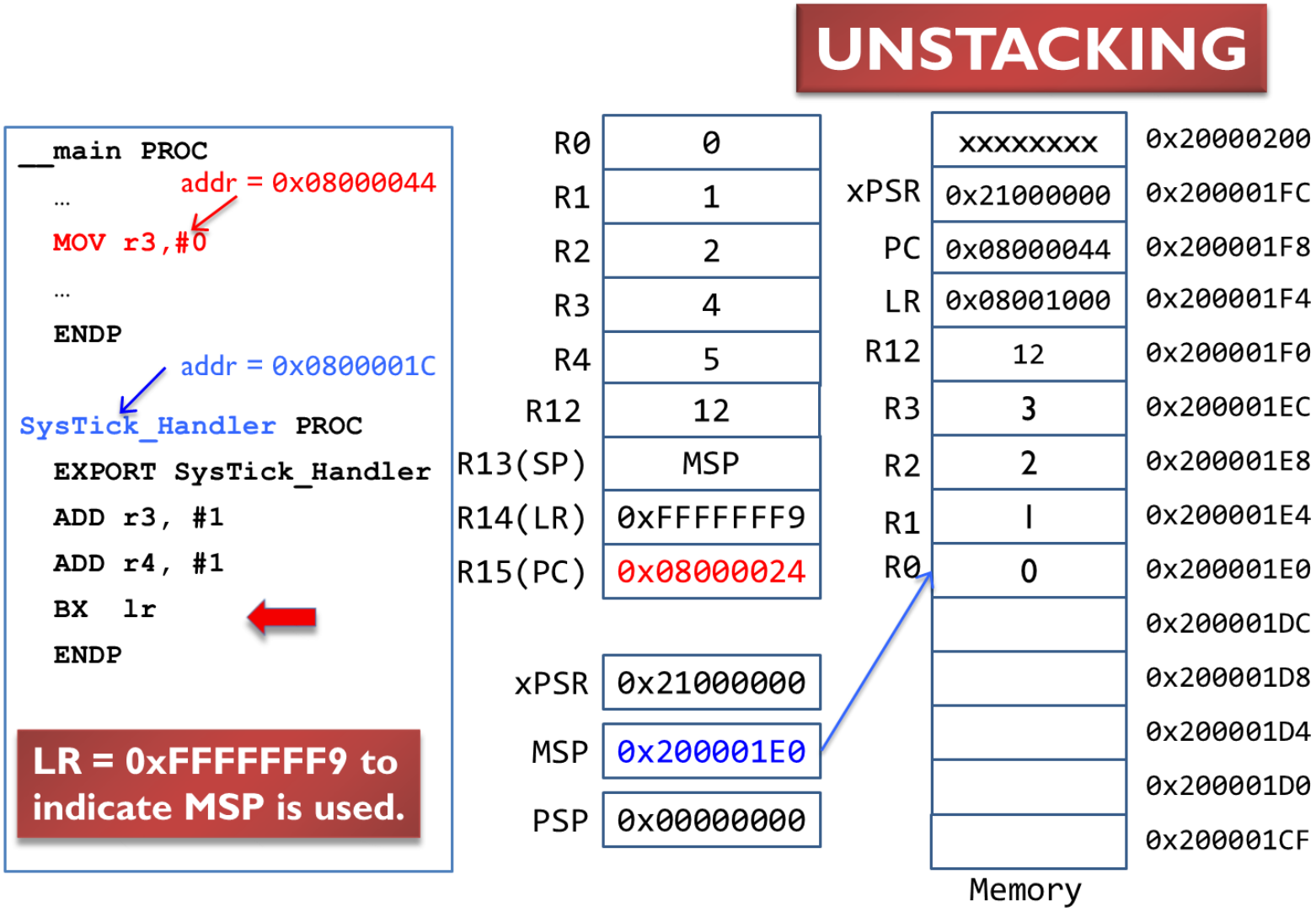
Interrupt: Stacking & Unstacking

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```

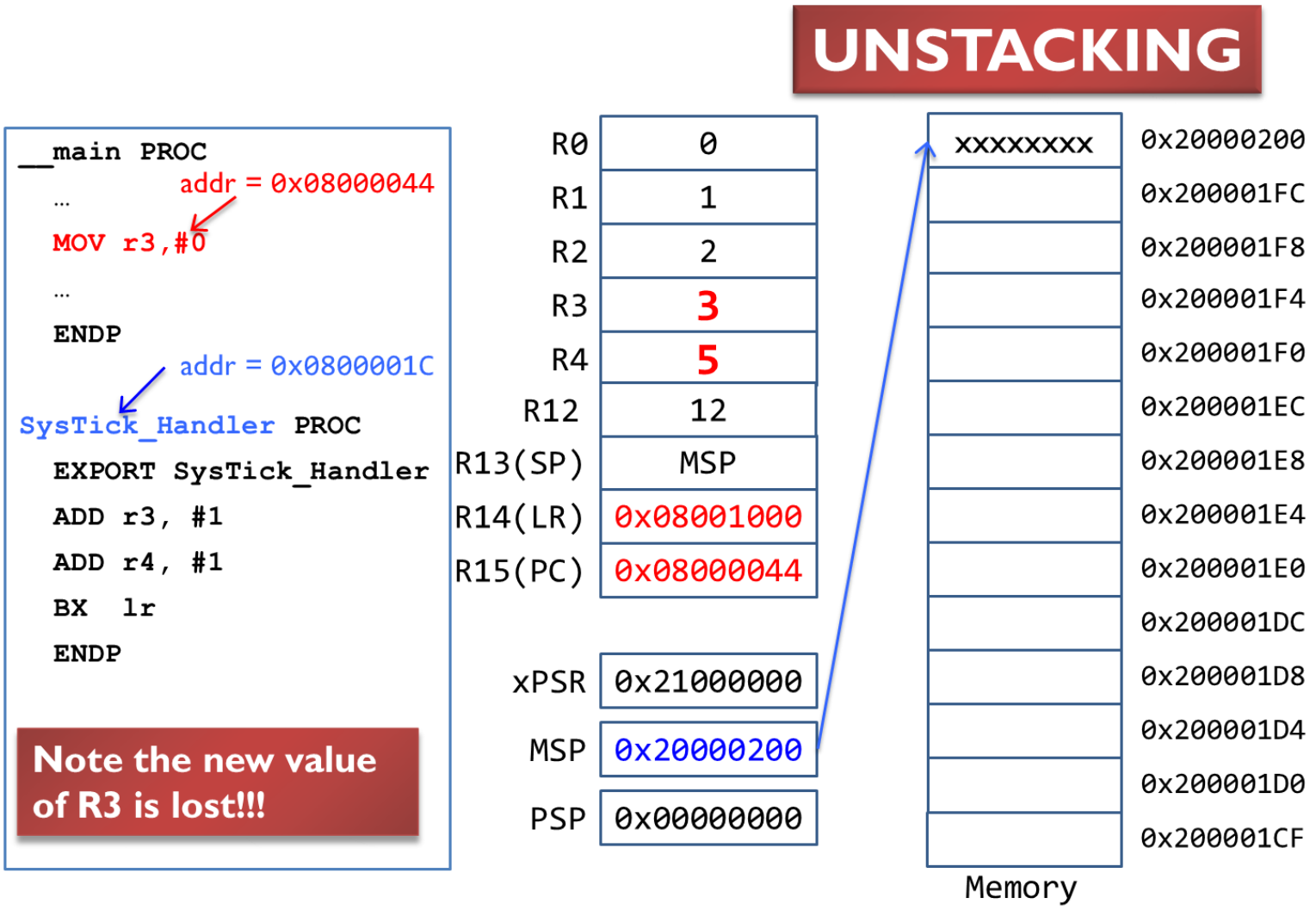
LR = 0xFFFFFFF9 to indicate MSP is used.



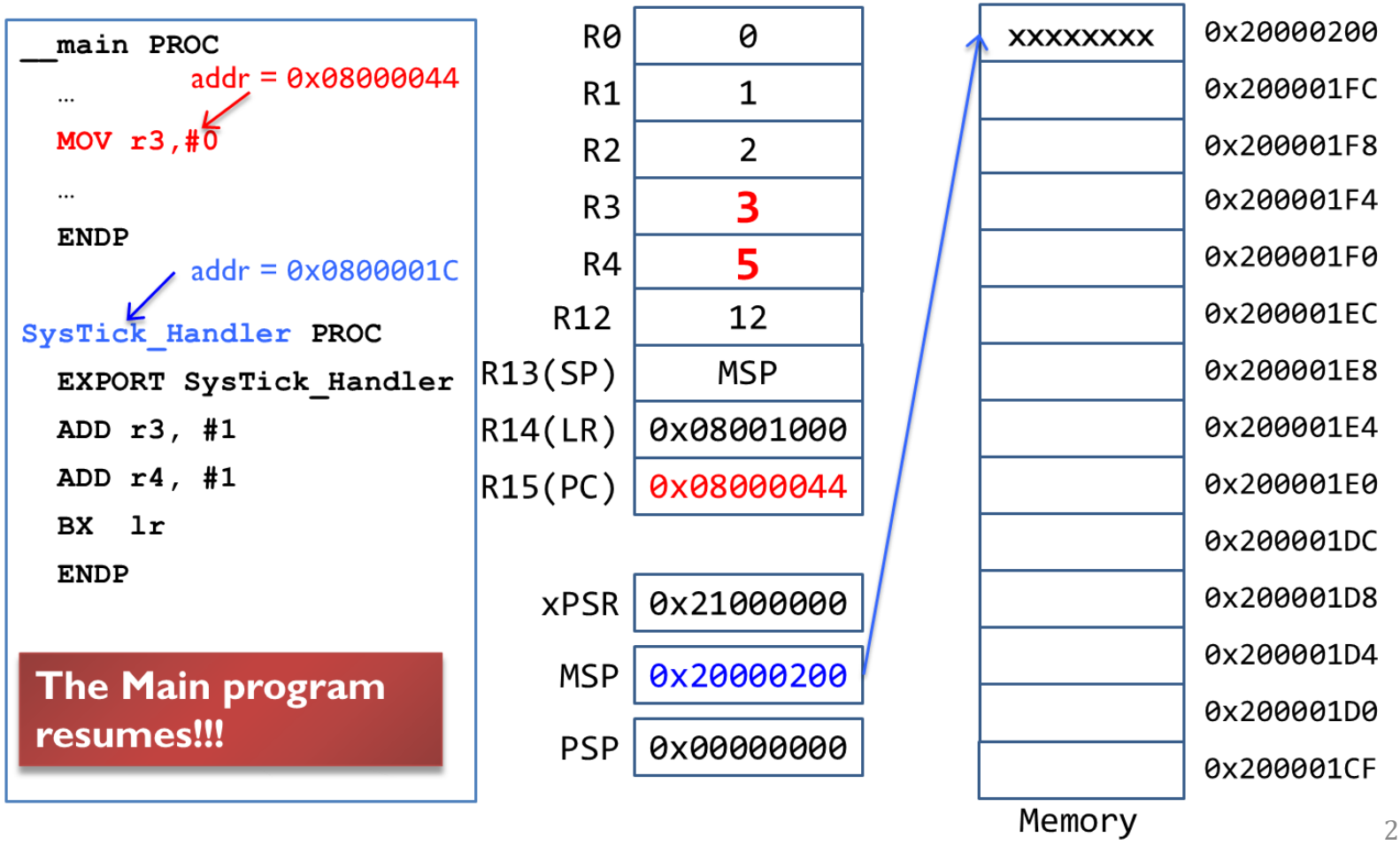
Interrupt: Stacking & Unstacking



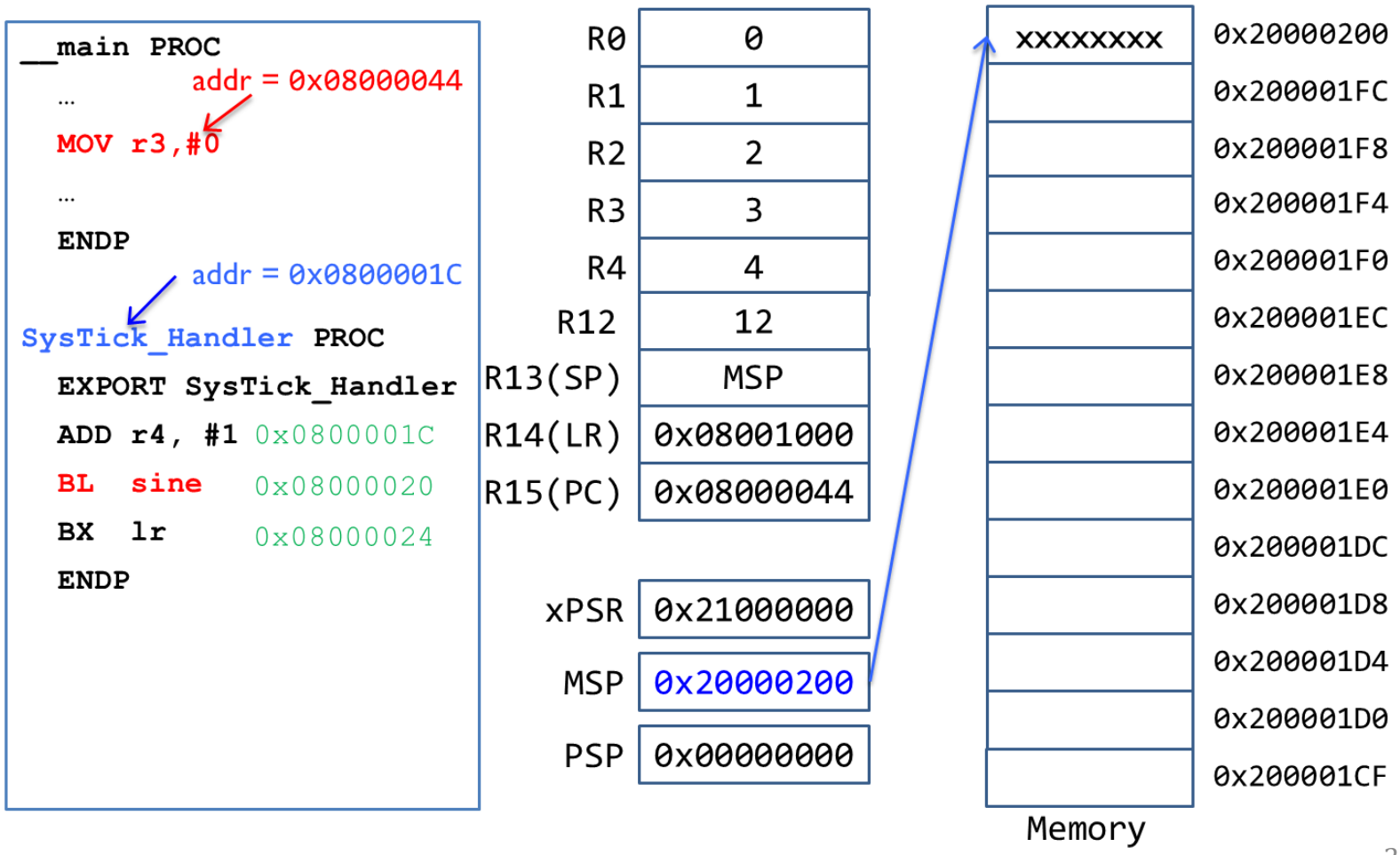
Interrupt: Stacking & Unstacking



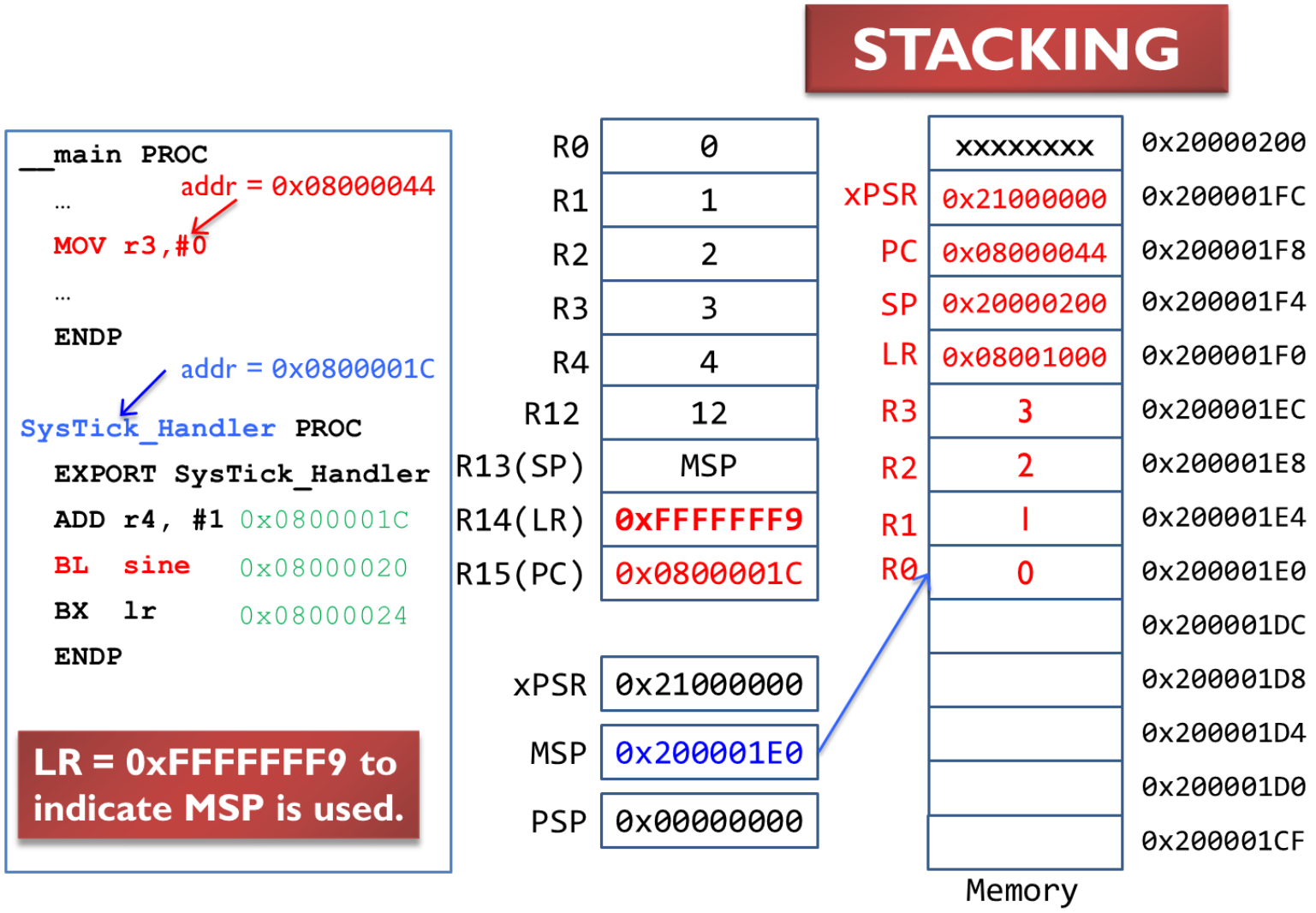
Interrupt: Stacking & Unstacking



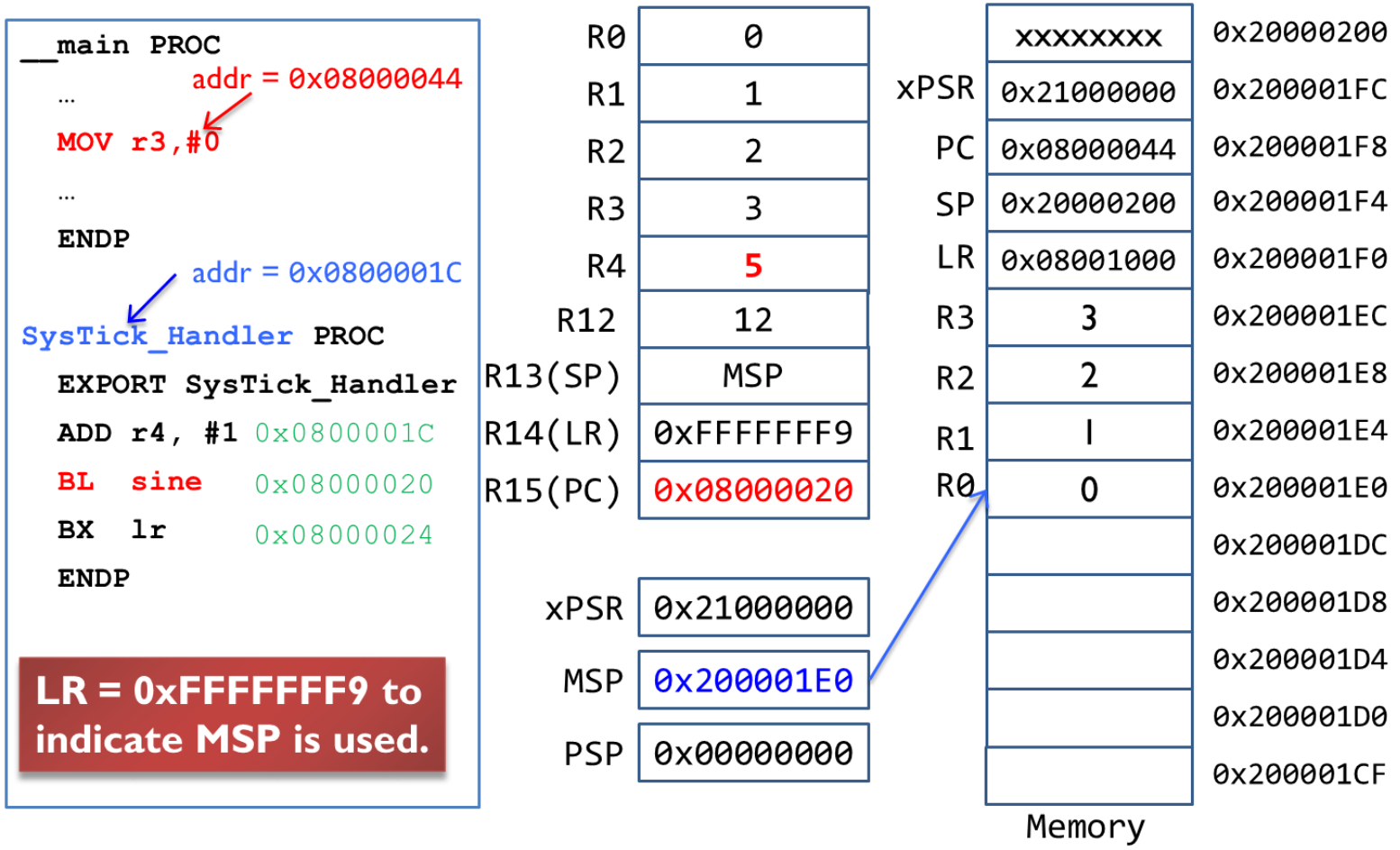
Interrupt: Stacking & Unstacking



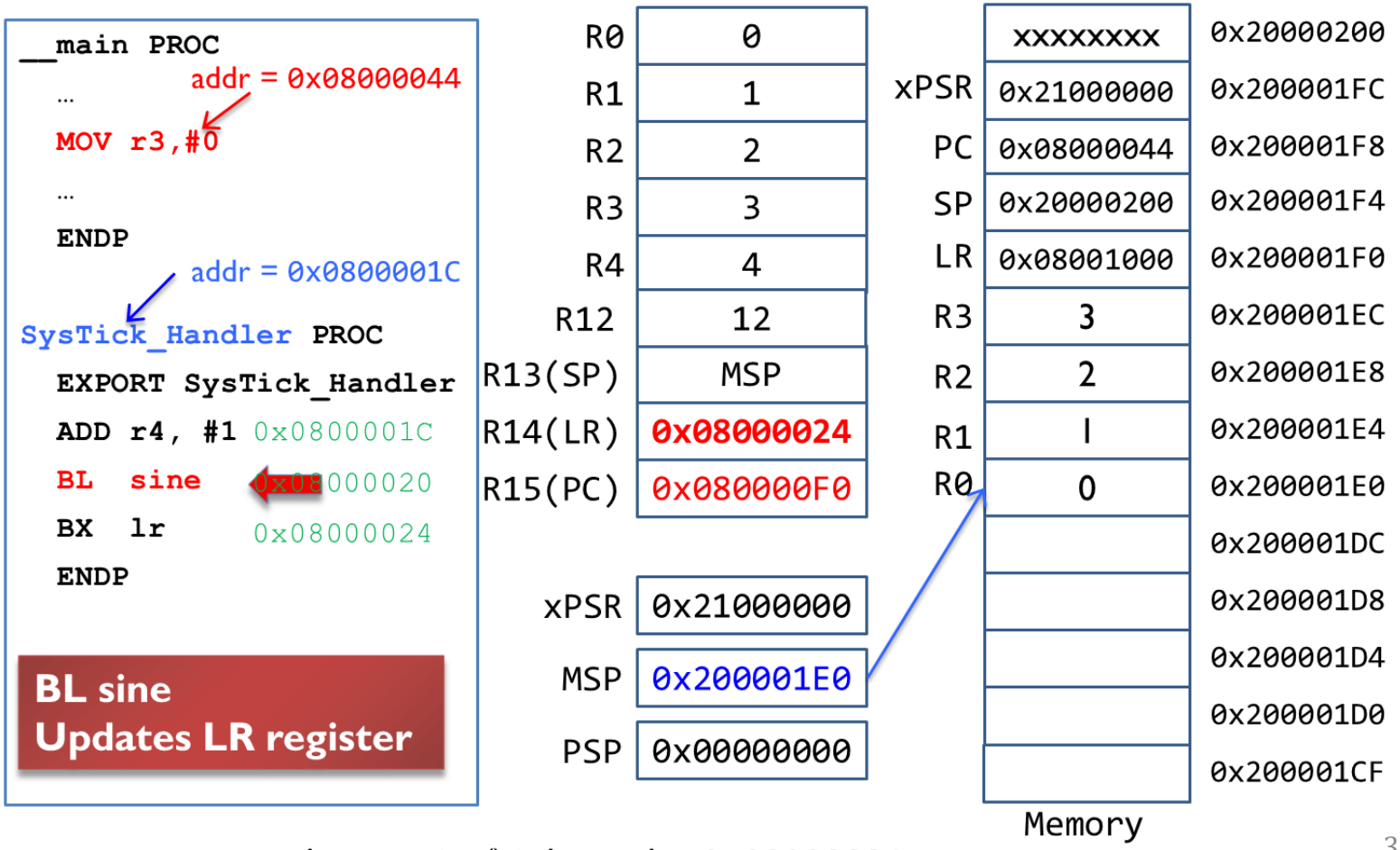
Interrupt: Stacking & Unstacking



Interrupt: Stacking & Unstacking

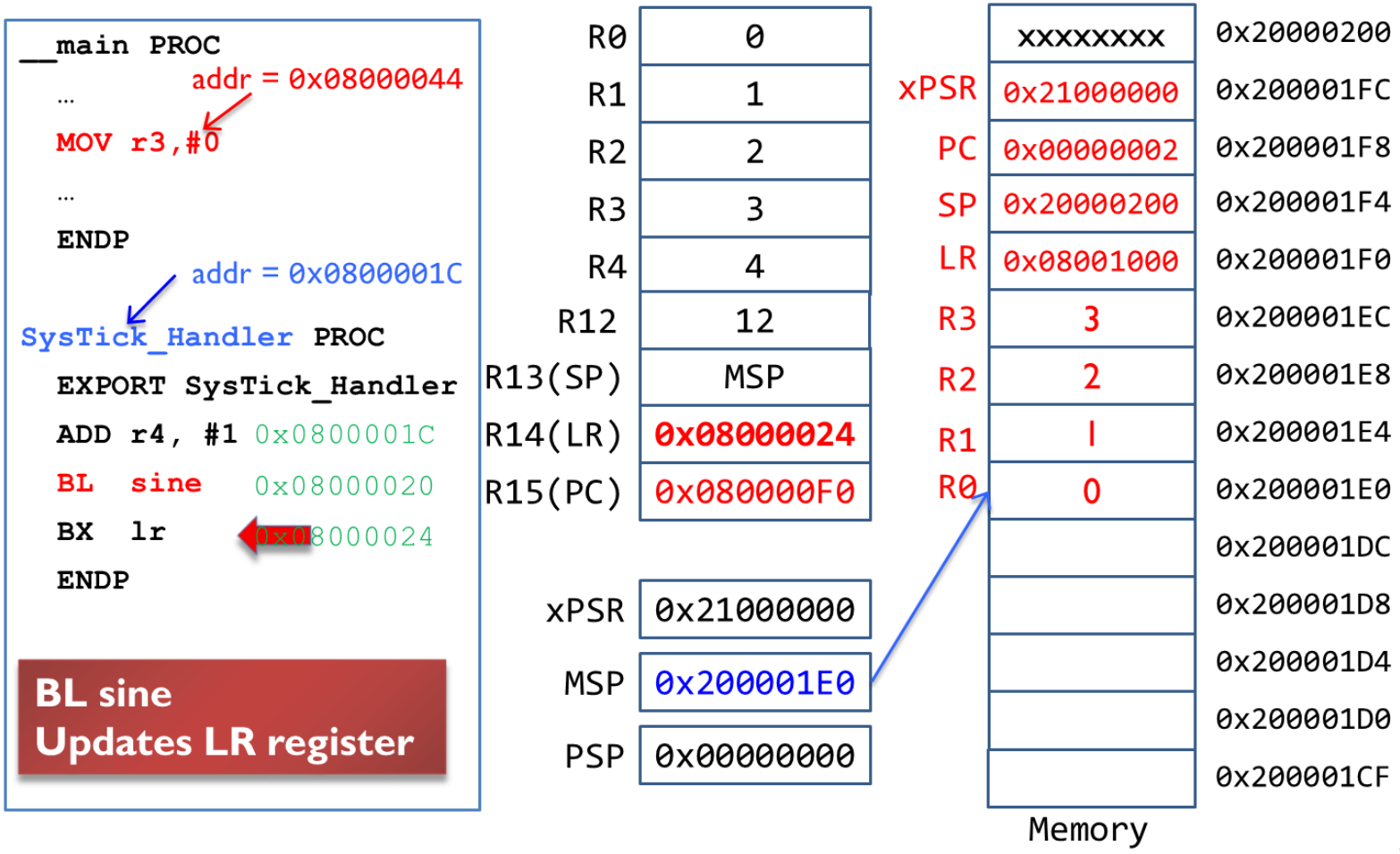


Interrupt: Stacking & Unstacking

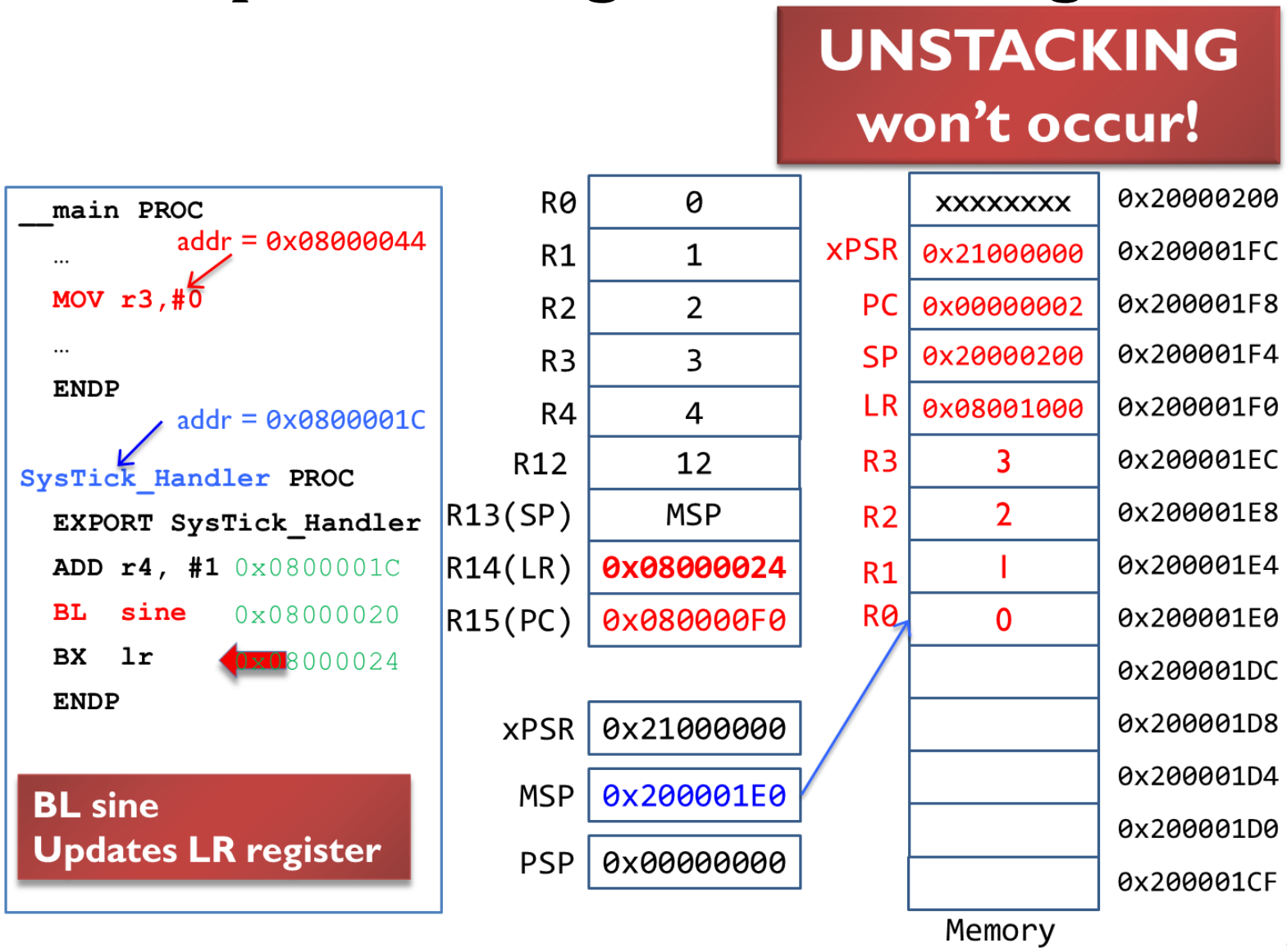


Assume sine() is located at 0x08000024.

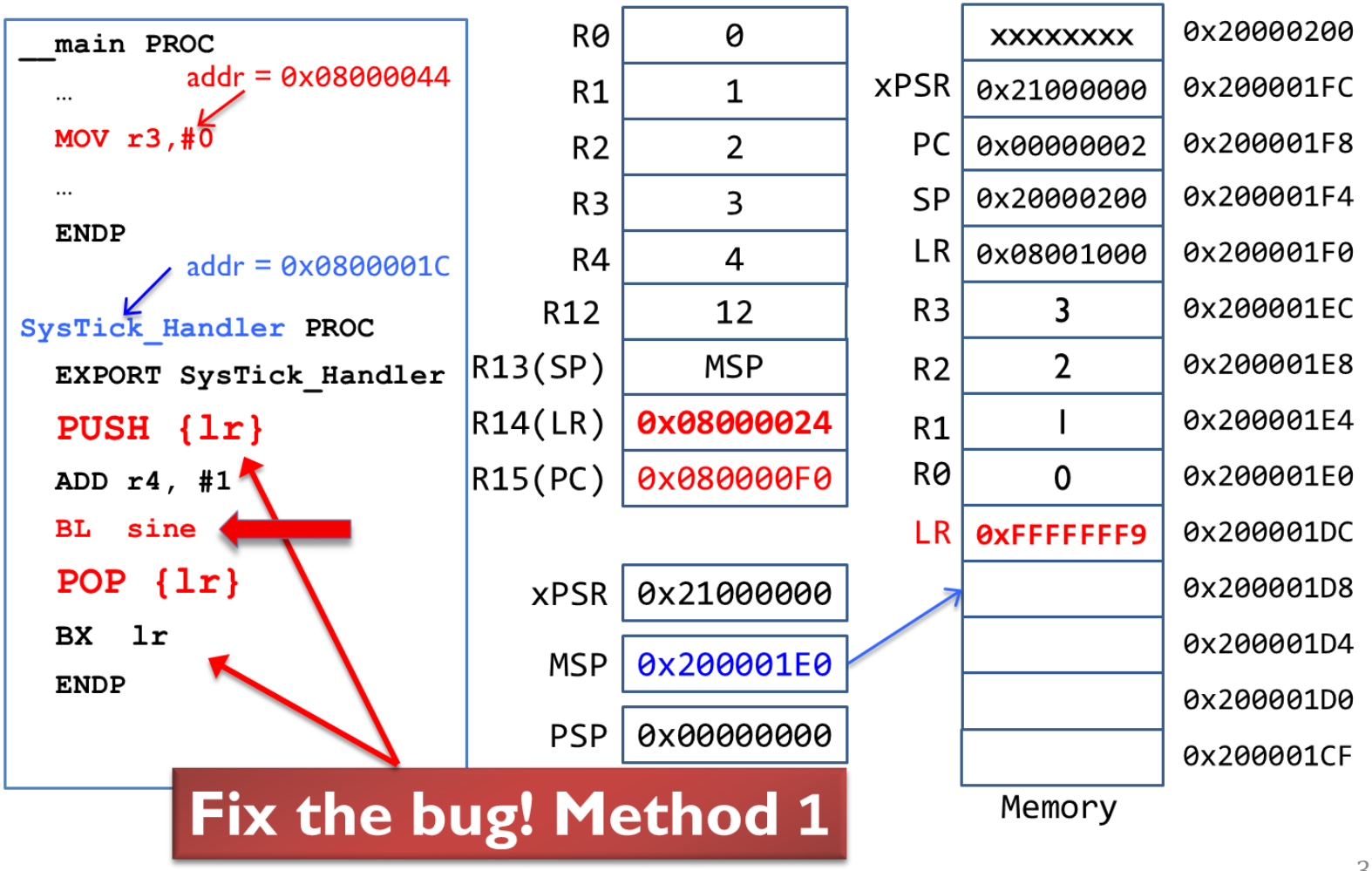
Interrupt: Stacking & Unstacking



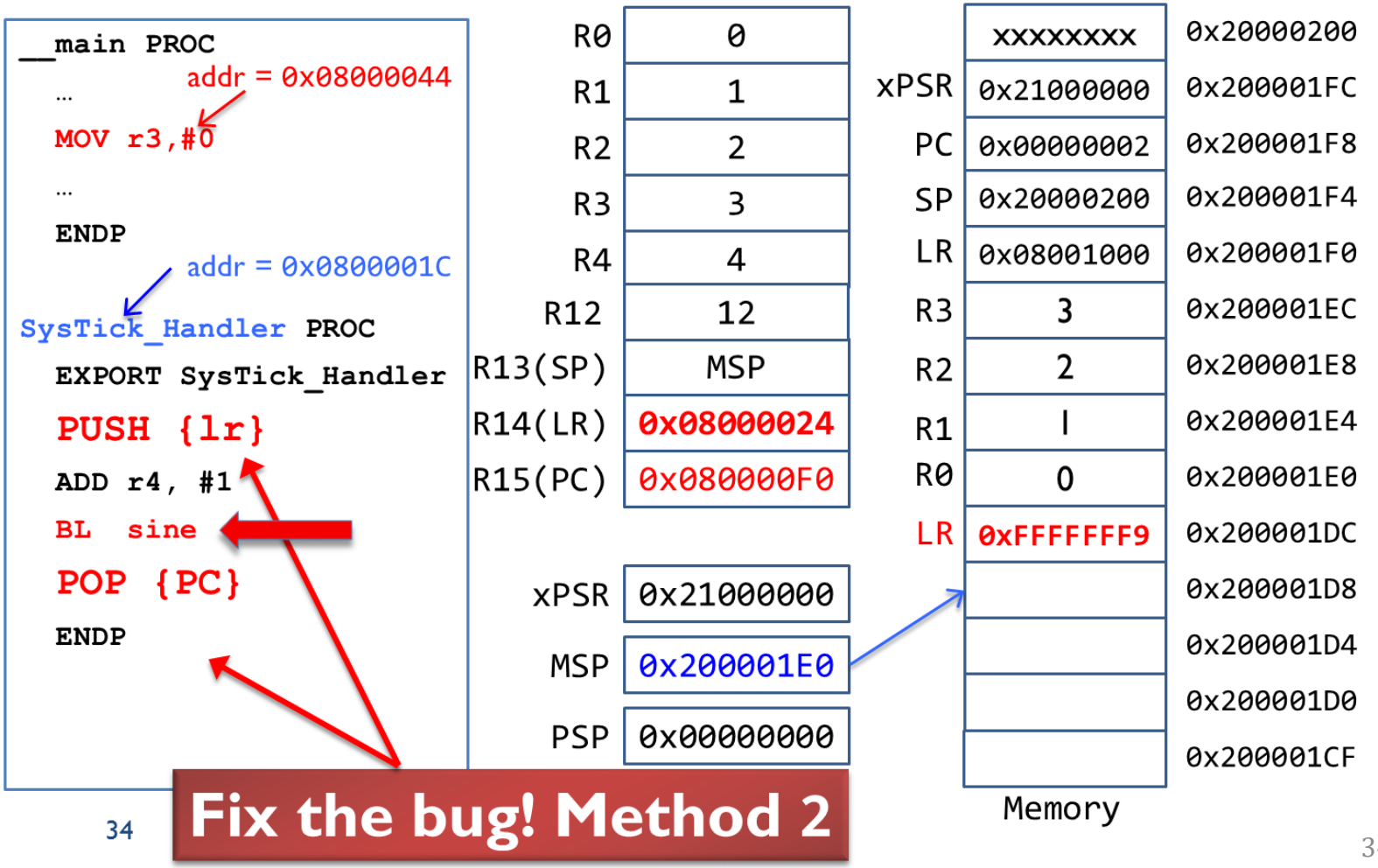
Interrupt: Stacking & Unstacking



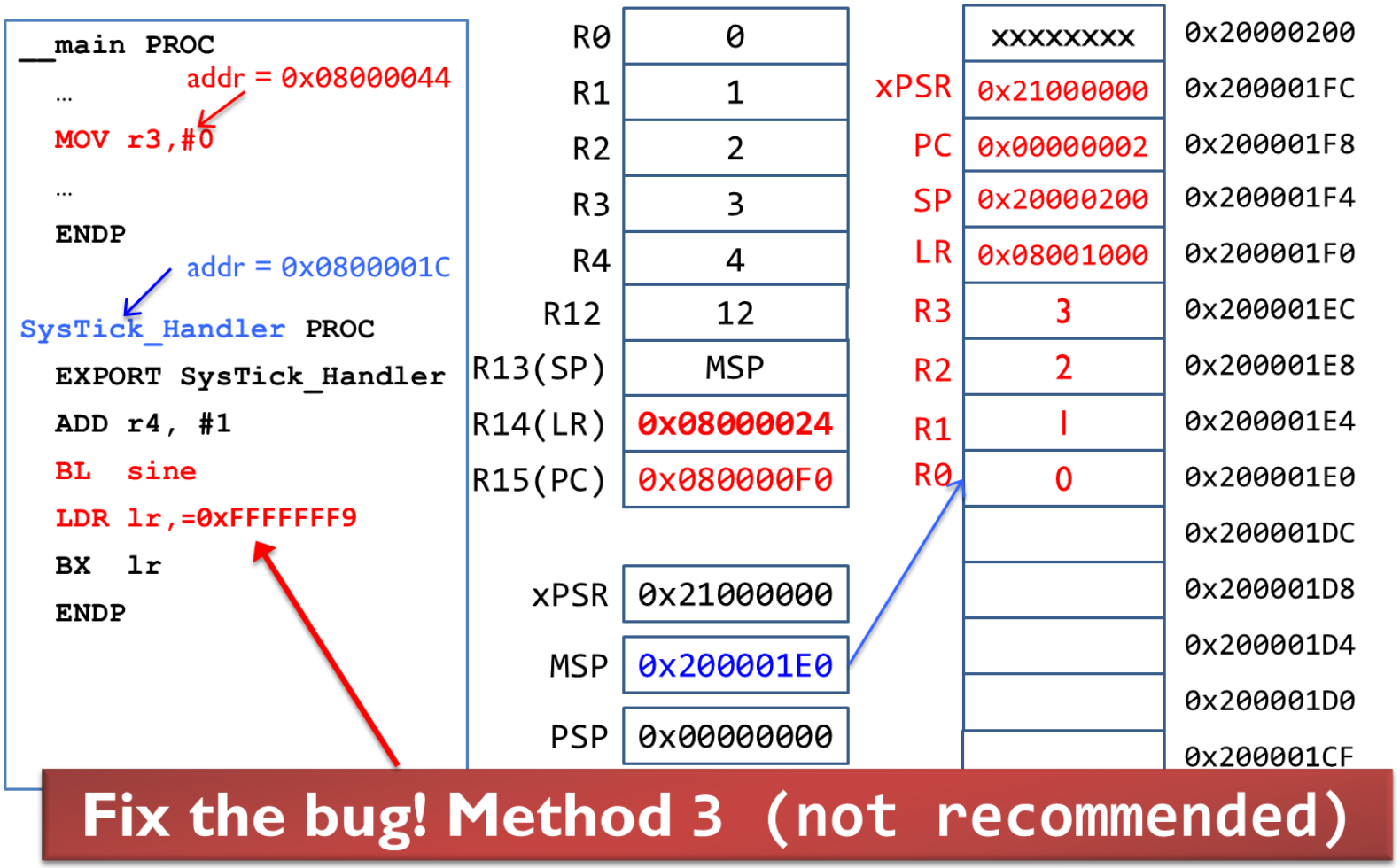
Interrupt: Stacking & Unstacking



Interrupt: Stacking & Unstacking

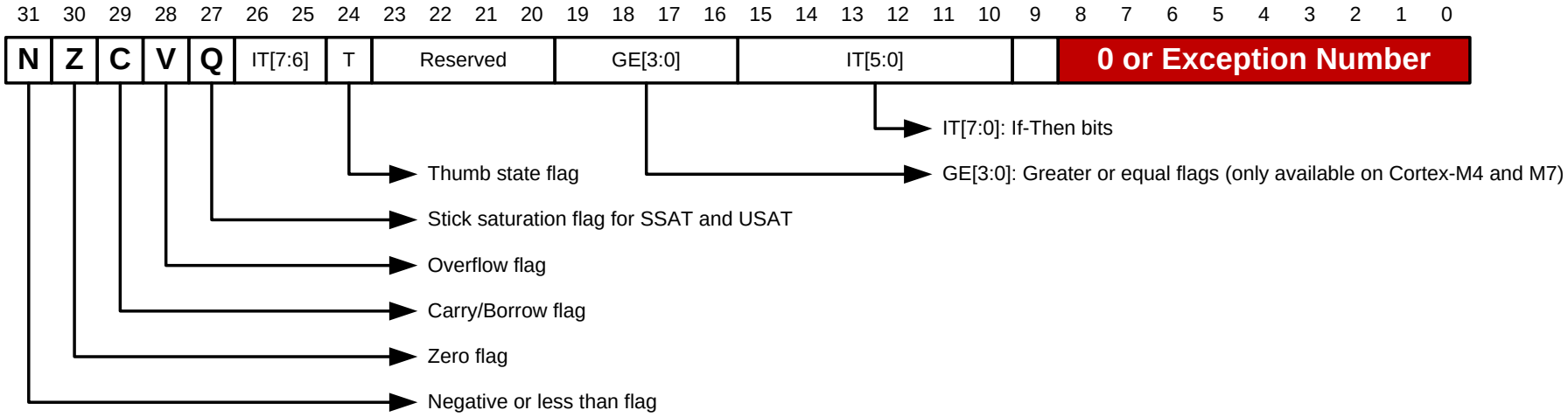


Interrupt: Stacking & Unstacking



Fix the bug! Method 3 (not recommended)

Interrupt Number in PSR



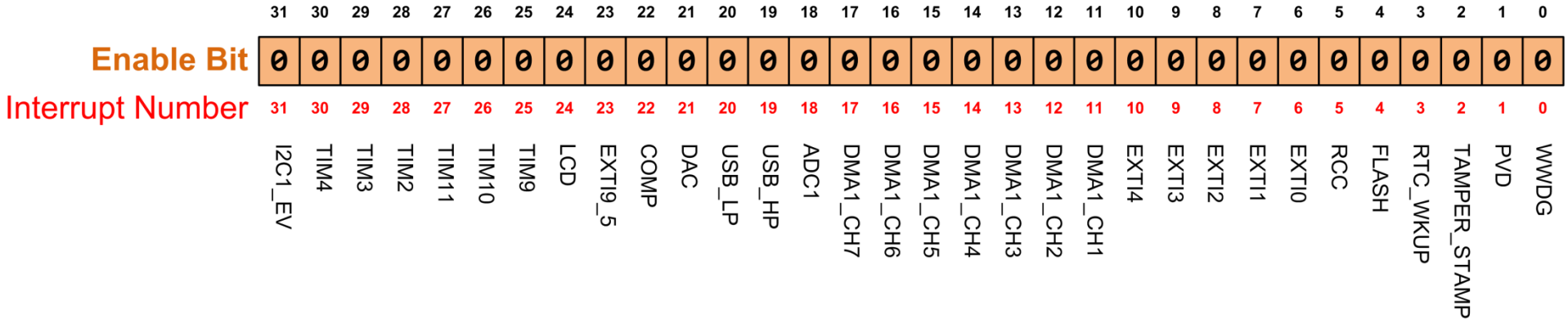
Enable an Interrupt/Exception

- Enable a system exception
  - Some are always enabled (cannot be disabled)
  - No centralized registers for enabling/disabling
  - Each are control by its corresponding components, such as SysTick module

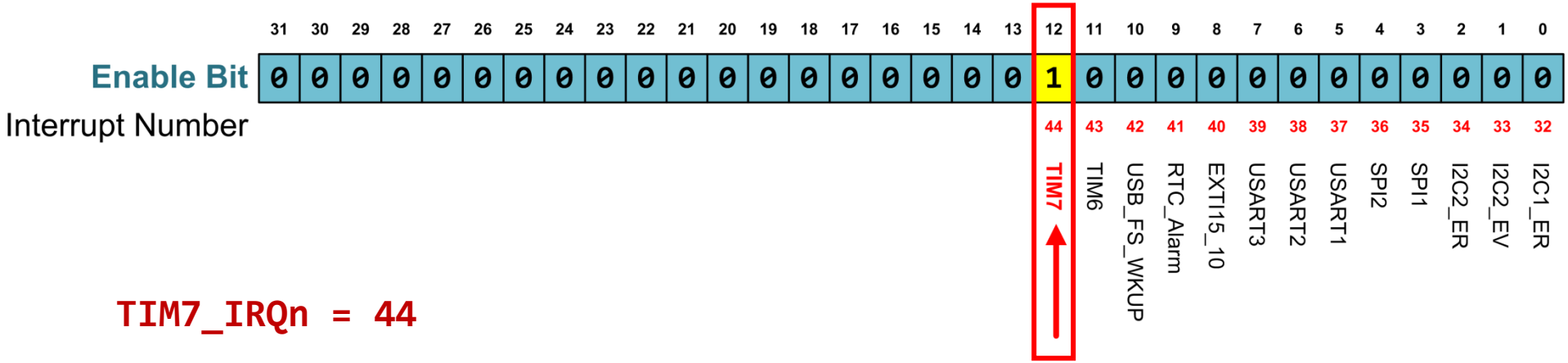
- Enable a peripheral interrupt
  - Centralized register arrays for enabling/disabling
  - NVIC's **`ISER`** 0~15 registers for enabling
    - Interrupt Set Enable Register
  - NVIC's **`ICER`** 0~15 registers for disabling
    - Interrupt Clear Enable Register

Enabling Peripheral Interrupts

Interrupt Set Enable Register 0 (ISER0)



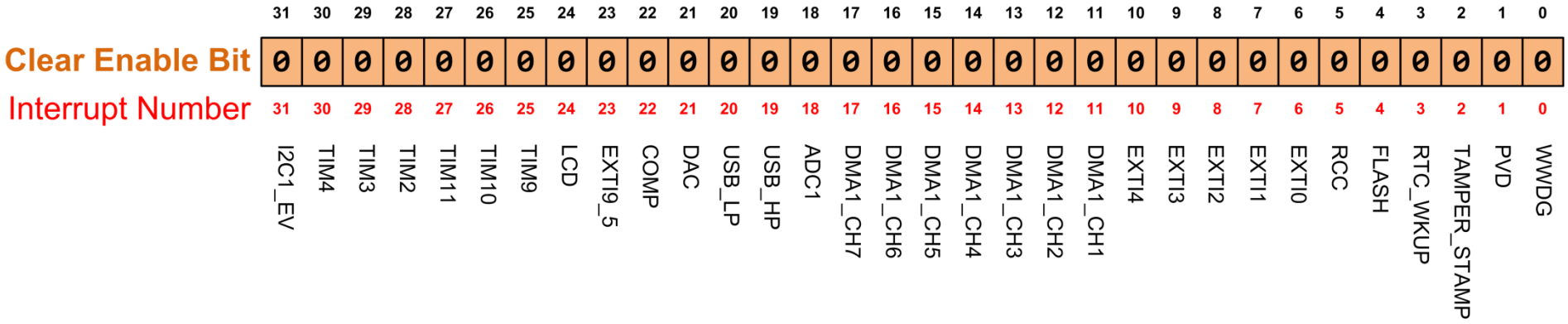
Interrupt Set Enable Register 1 (ISER1) *Address of ISER1 = Address of ISER0 + 4*



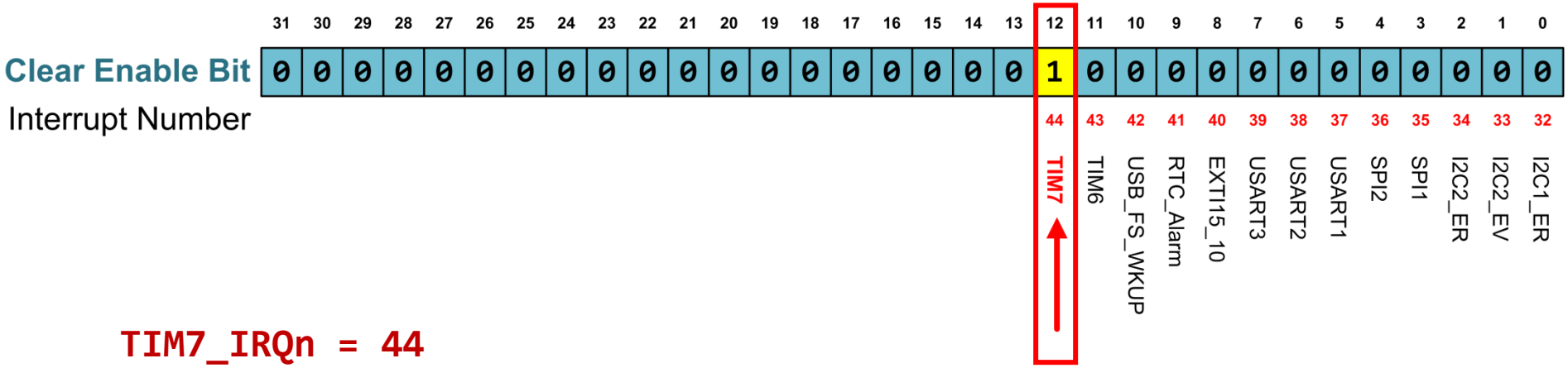

```
TIM7_IRQn = 44

NVIC->ISER[1] = 1 << 12;      // Enable Timer 7 interrupt
```

Disabling Peripheral Interrupts

Interrupt Clear Enable Register 0 (ICER0)



Interrupt Clear Enable Register 1 (ICER1) *Address of ICER1 = Address of ISER0 + 4*




```
TIM7_IRQn = 44

NVIC->ICER[1] = 1 << 12;      // Diable Timer 7 interrupt
```

Priority Management

Interrupt Priority

- Inverse Relationship:
  - Lower priority value means higher urgency.
    - Priority of Interrupt A = 5,
    - Priority of Interrupt B = 2,
    - B has a higher priority/urgency than A.

- Fixed priority for Reset, HardFault, and NMI.

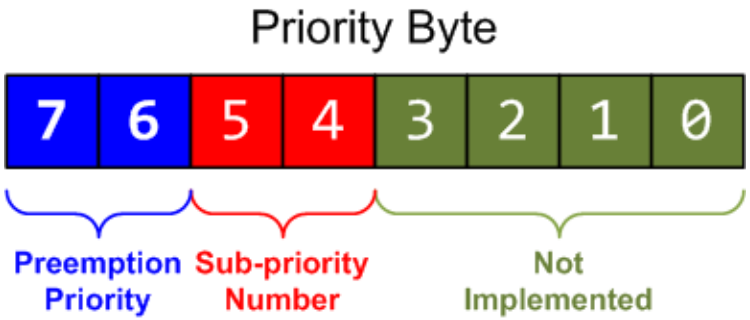
Exception	**IRQn**	**Priority**
Reset	N/A	-3 (the highest)
Non-maskable Interrupt (NMI)	14	-2 (2nd highest)
Hard Fault	13	-1

- Adjustable for all the other interrupts

Interrupt Priority

- Interrupt priority is configured by Interrupt Priority Register (IPR) 0~123
  - 124 IPRs * 4 priority configuration per IPR = 496

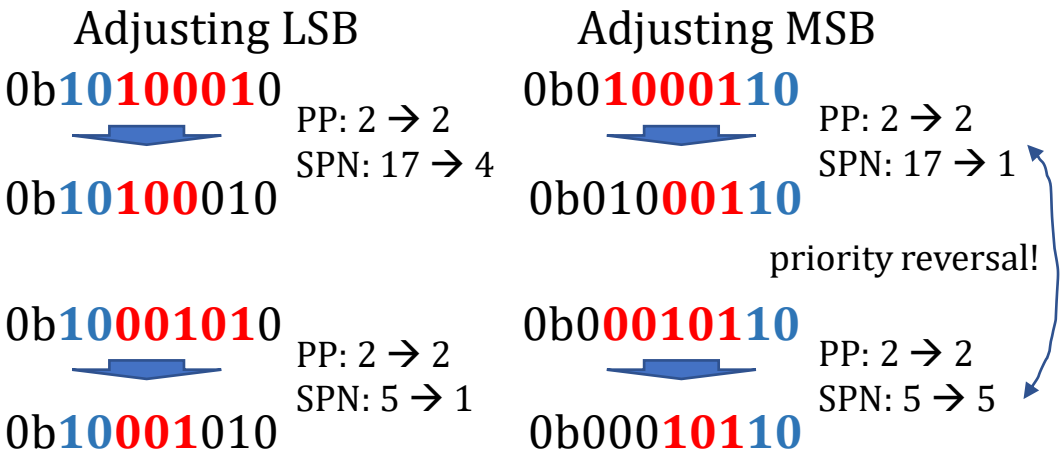
    (= Total # of interrupts supported on ARMv7-M)
- Each priority consists of two fields, including preempt priority number and sub-priority number.
  - The preempt priority number defines the priority for preemption.
  - The sub-priority number determines the order when multiple interrupts are pending with the same preempt priority number.

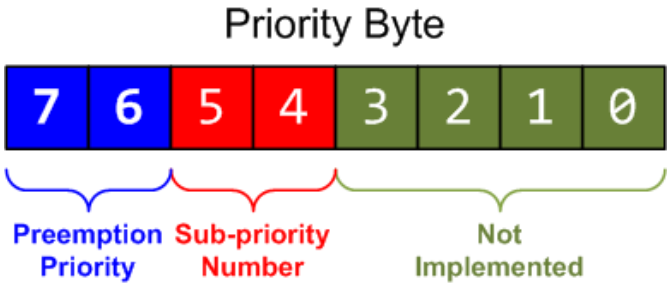


default setting

Interrupt Priority

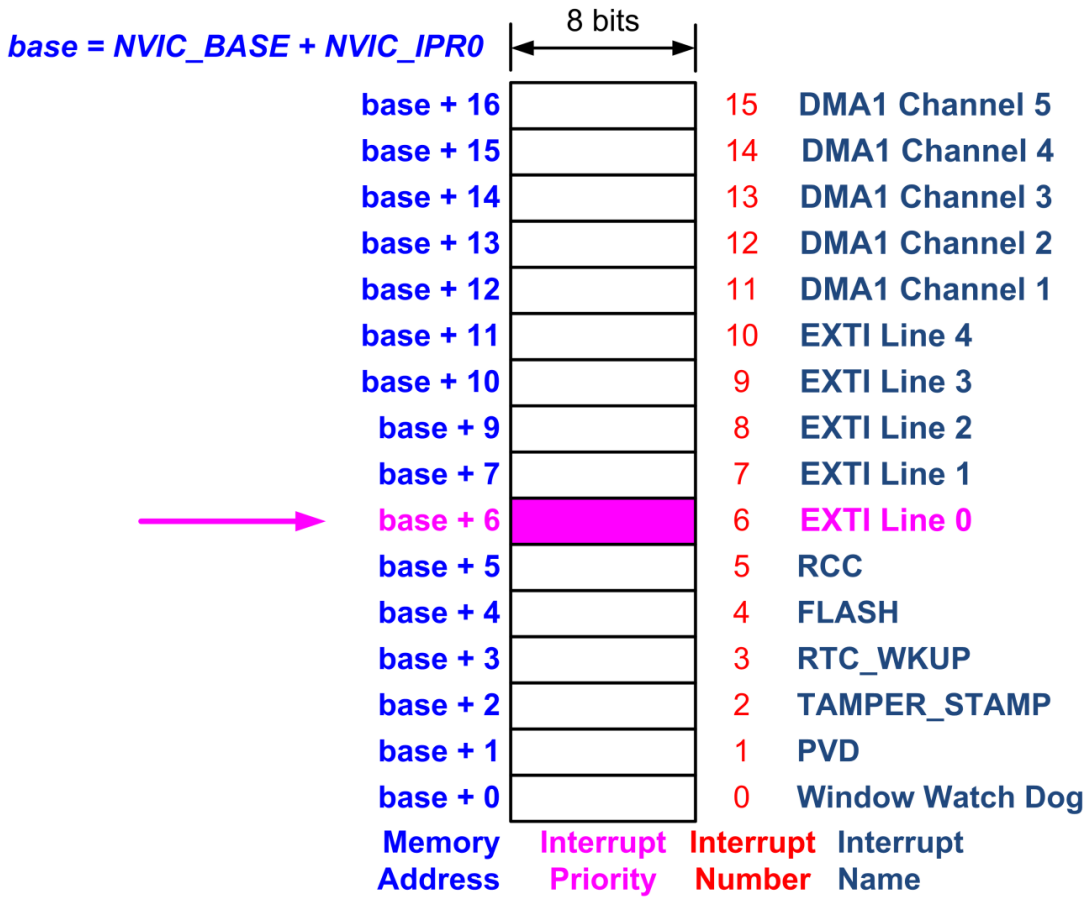
- The lengths of the Preemption Priority-field and the Sub-priority Number-field are configurable.
  - The position of LSB is adjusted.
- Why adjusting LSB?
  - Easier porting!
  - Programs implemented on many-bit priority-level device can run on small-bit priority level device without inversion of priority.
  - example: 2/5 bits priority fields → 2/3 bits priority fields





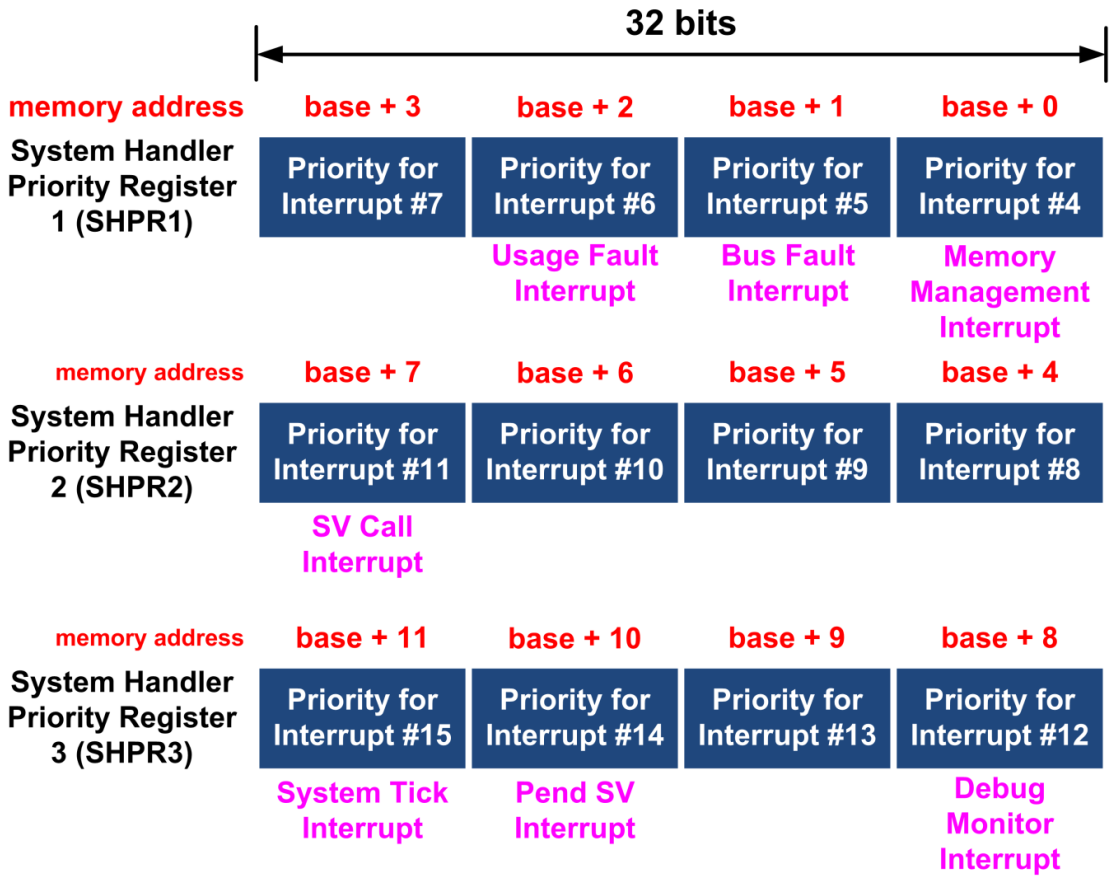
default setting

Priority of Peripheral Interrupts




```
// Set the priority for EXTI 0 (Interrupt number 6)
NVIC->IP[6] = 0xF0;
```

Priority of System Interrupts




```
// Set the priority of a system interrupt IRQn
SCB->SHP[(IRQn) & 0xF) - 4] = (priority << 4) & 0xFF;
```

Exception-masking registers

- PRIMASK: Used to disable all exceptions except Non-maskable interrupt (NMI) and hard fault.
  - Write 1 to PRIMASK to disable all interrupts except NMI and hard fault exception

```
MOV R0, #1
MSR PRIMASK, R0
```

  - Write 0 to PRIMASK to enable all interrupts

```
MOV R0, #0
MSR PRIMASK, R0
```

- FAULTMASK: Like PRIMASK but change the current priority level to -1, so that even hard fault handler is blocked
- BASEPRI: Disable interrupts only with priority lower than a certain level
  - Example, disable all exceptions with priority value higher than 0x60

```
MOV R0, #0x60
MSR BASEPRI, R0
```