# CS301
# Embedded System and Microcomputer Principle

## Lecture 5: Interrupt

2023 Fall

# Recap

- EmbeddedC
  - Operations
    - Bit manipulation
      - &: clear a bit
      - |: set a bit
      - ^: toggle a bit
      - <<, >>
    - else-if vs switch case
    - macro
      - replacement of code
      - function vs macro
  - Data types
    - pointers
      - pointer is address
      - pointer's size: width of address bus
    - array
      - pointer increment/decrement
    - string
      - array of char, terminated by an \0
    - struct
      - memory alignment for members with padding
      - bit field structure
    - union
      - members occupy same memory space
  - storage
    - Scope and Lifetime
      - global variable
        - scope: entire project (static global: current file)
        - lifetime: from declaration till end of process
      - auto variable
        - scope: within function
        - lifetime: from declaration till end of function
      - static local variable
        - scope: within function
        - lifetime: from declaration till end of process
        - value maintains between function invocations
    - volatile
      - value may be changed outside
    - malloc()/free()
      - More costly
      - unpredictable in time

# Outline

- **Subroutine**
- Interrupt

# Recall: Branch

Branch changes the Program Counter (PC) and causes the CPU to execute an instruction other than the next instruction.

- Unconditional Branch: When CPU executes an unconditional branch, it jumps unconditionally (without checking any condition) to the target location.
  - Example: B, (BL, and BX when calling subroutine)
- Conditional Branch: When CPU executes a conditional branch, it checks a condition, if the condition is true then it jumps to the target location; otherwise, it executes the next instruction.
  - Example: BCS, BEQ, BGE, etc

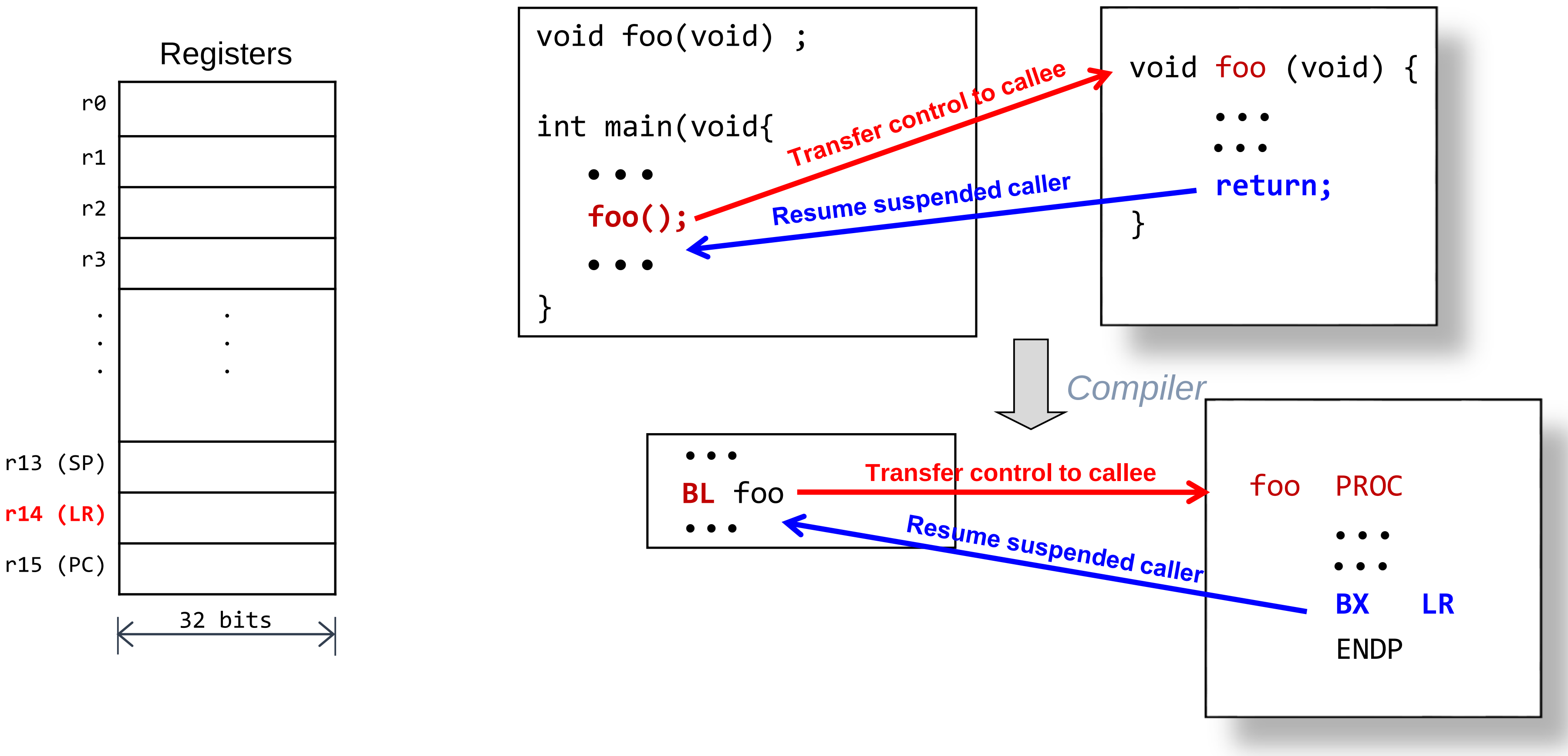
# Subroutine

- Subroutines are often used to perform tasks that need to be performed frequently
- How to call a subroutine?
  - BL: Branch and link
  - Call a subroutine while saving the return address in the link register (LR)
- How to return the control back to the caller?
  - BX: Branch with eXchange
  - Branch to an address specified in a register. The processor copies LR to PC after the program is finished.


Link Register: Stores the return address for function calls

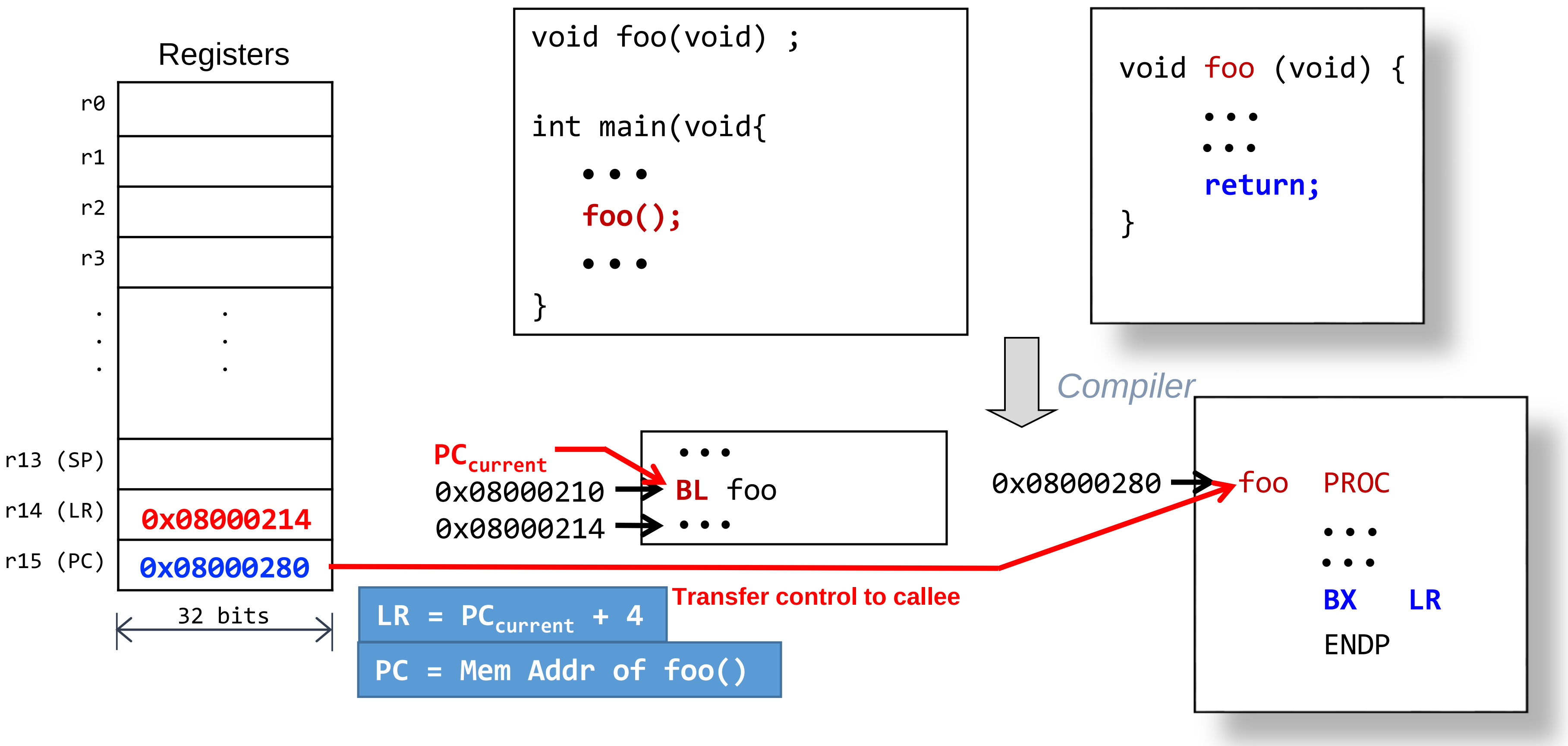
Program Counter: Memory address of the to be executed instruction

# Subroutine


Registers
r0
r1
r2
r3
r13 (SP)
r14 (LR)
r15 (PC)
32 bits
void foo(void) ;
int main(void{
foo();
}
Transfer control to callee
Resume suspended caller
void foo (void) {
return;
}
Compiler
BL foo
Transfer control to callee
Resume suspended caller
foo PROC
BX LR
ENDP

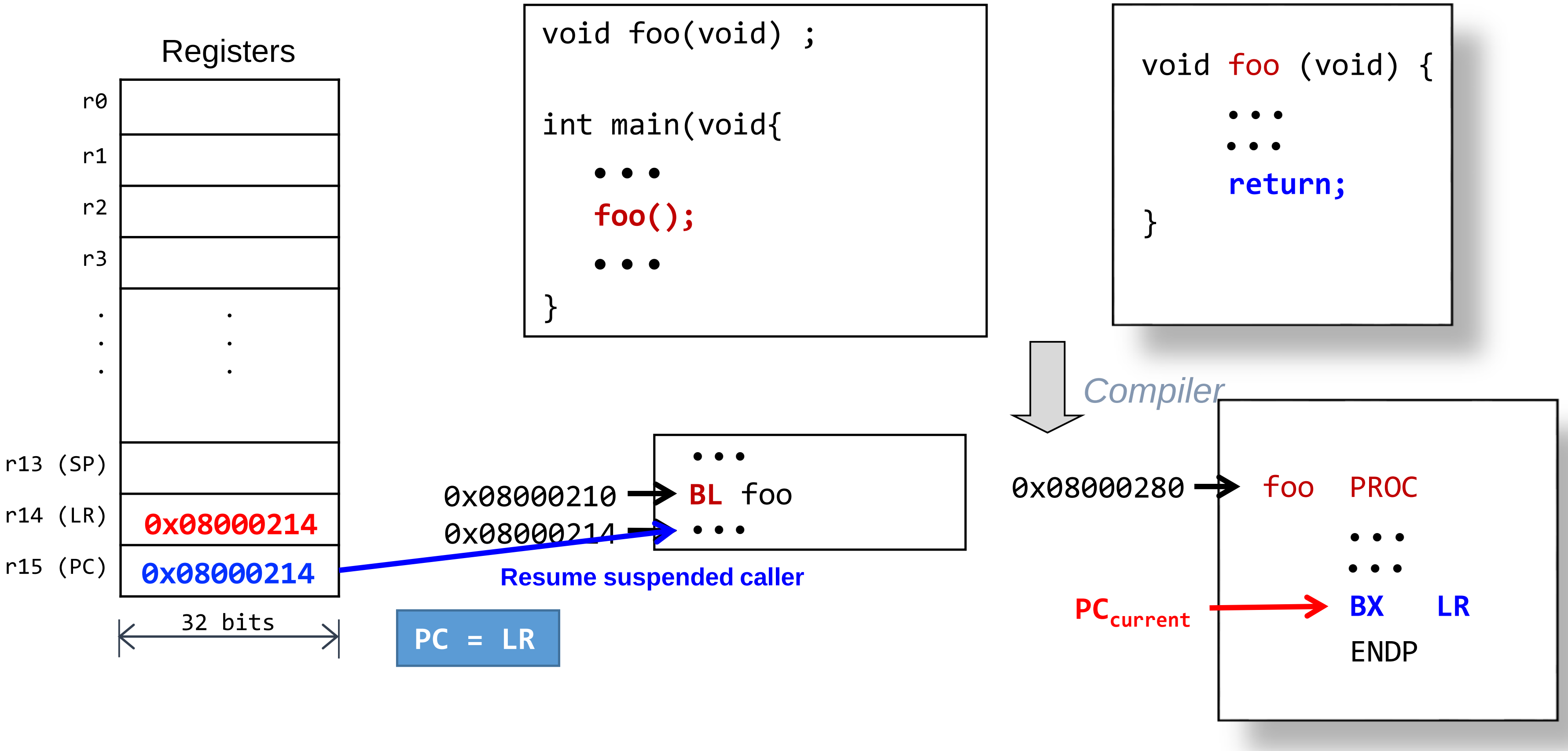
# Call a Subroutine

- BL label
  - Step 1: LR = PC + 4
  - Step 2: PC = label



# Exit a Subroutine

- BX LR
  - MOV PC, LR



# Nested Subroutine

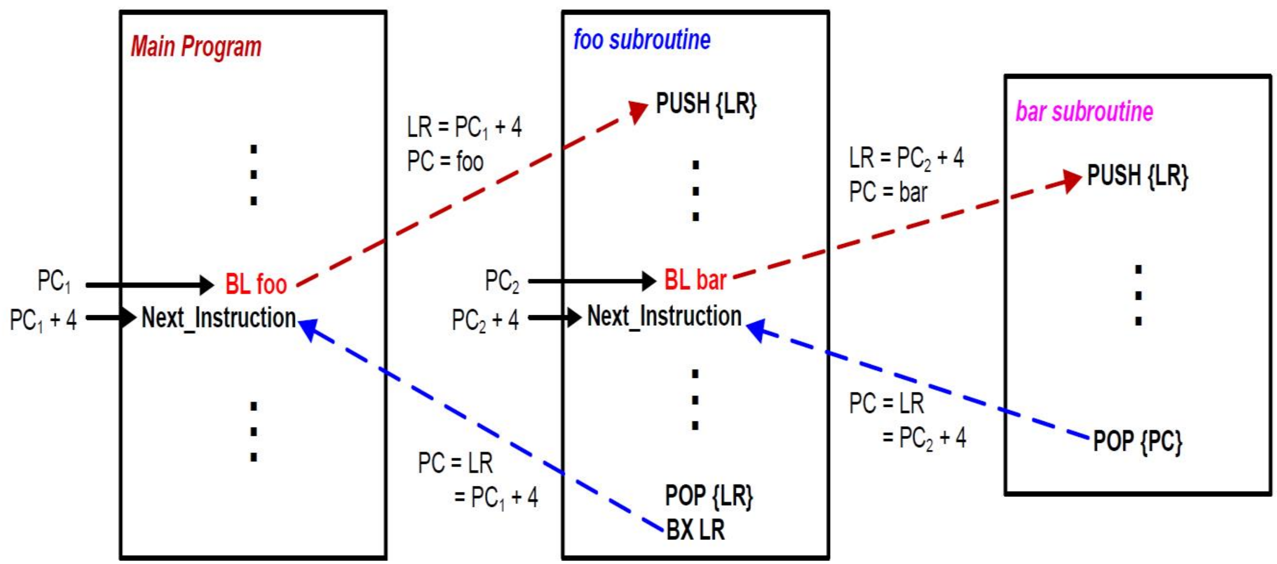
- What happens if a BL occurs in a subroutine?
  - Need to preserve runtime environment via stack

```
void foo(void) ;

int main(void{
   • • •
   foo();
   • • •
}
```

```
void foo (void) {
      • • •
      bar();
      • • •
      return;
}
```

```
void bar (void) {
      • • •
      • • •
      return;
}
```

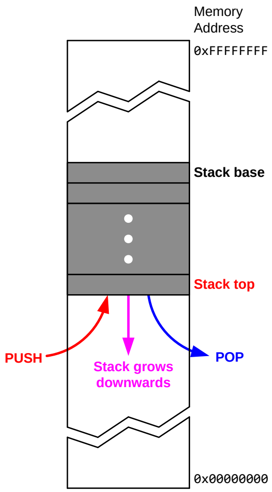


# The Stack

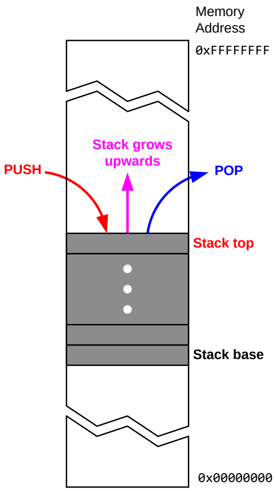
- The stack is a "last in first out" queue
  - that means whatever data was added to the stack ('pushed') last is taken from the stack ('popped') first.
  - E.g. if the 32bits values pushed onto a stack were 0x0000FFFF, 0xFFFF0000, 0xAAAAAAAA in that order then they would be popped from the stack in the reverse order.
- In memory the stack is held as a list:
  - top of stack 0xAAAAAAAA
    0xFFFF0000
  - bottom of stack 0x0000FFFF

# Stack Growth Convention

- Ascending vs Descending

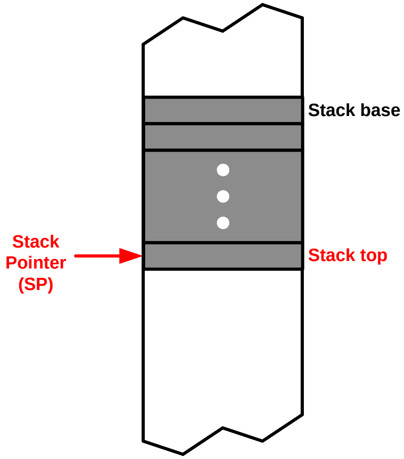



***Descending stack***: Stack grows towards low memory address

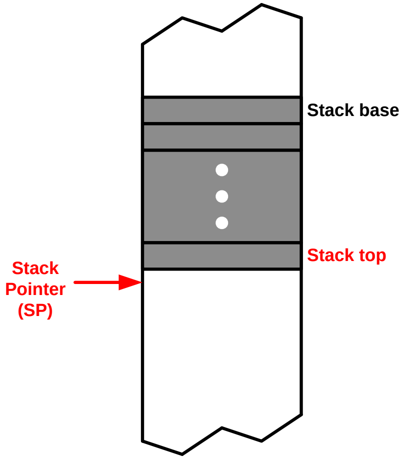



***Ascending stack***: Stack grows towards high memory address

- Full vs Empty



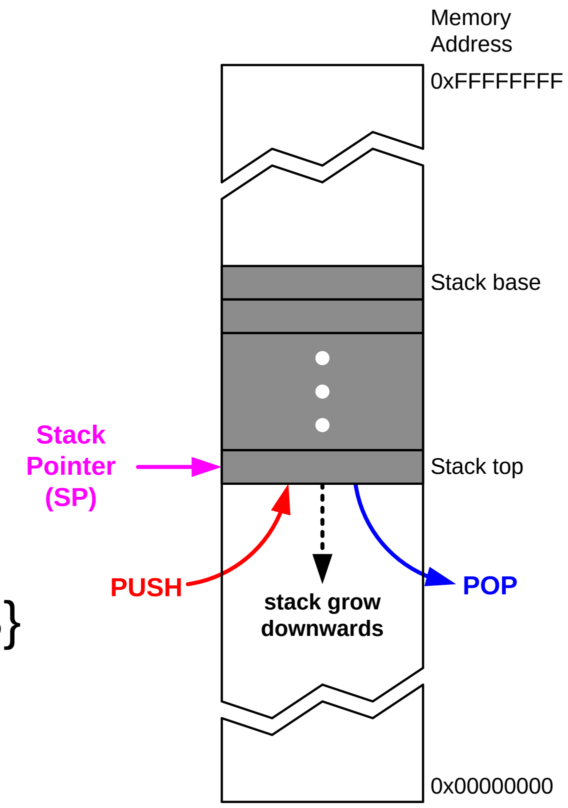

***Full stack***: SP points to the last item pushed onto the stack




***Empty stack***: SP points to the next free space on the stack

# Full Descending Stack

- Cortex-M uses full descending stack
  - The bottom(base) is fixed at a particular memory address
  - The top is identified by a register: SP
- Stack pointer (SP, aka R13)
  - decremented on PUSH
  - SP = SP – 4 * # of registers
  - incremented on POP
  - SP = SP + 4 * # of registers
- Example:
  - PUSH/POP {r0,r6,r3}
  - Equivalent to STMFD/LDMFD sp!, {r0, r6, r3}
    - Store Multiple Full Descending
    - Load Multiple Full Descending



# Stacking & Unstacking

- PUSH {Rd}
  - SP = SP-4 ⟶ descending stack
  - (*SP) = Rd ⟶ full stack
  - Push multiple registers

PUSH {r6, r7, r8} ⟺ PUSH {r8, r7, r6} ⟺ PUSH {r8}
PUSH {r7}
PUSH {r6}

- POP {Rd}
  - Rd = (*SP) ⟶ full stack
  - SP = SP + 4 ⟶ Stack shrinks
  - Pop multiple registers

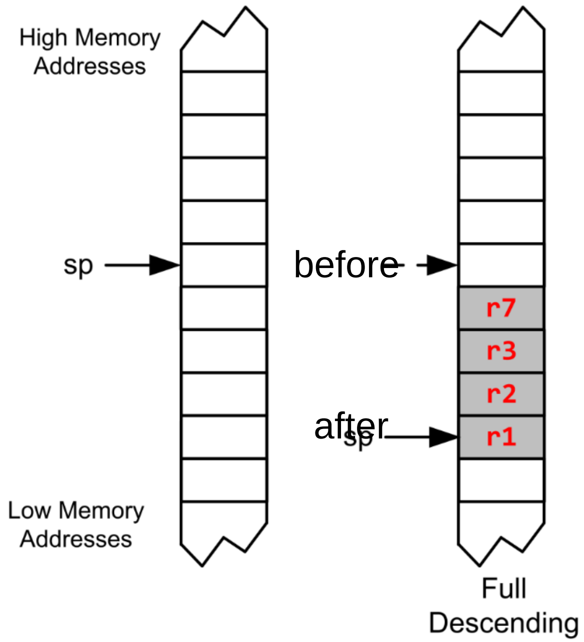
POP {r6, r7, r8} ⟺ POP {r8, r7, r6} ⟺ POP {r6}
POP {r7}
POP {r8}

- The order in which registers listed in the register list does not matter.
- When pushing/popping multiple registers, largest-numbered register is pushed first but popped last.

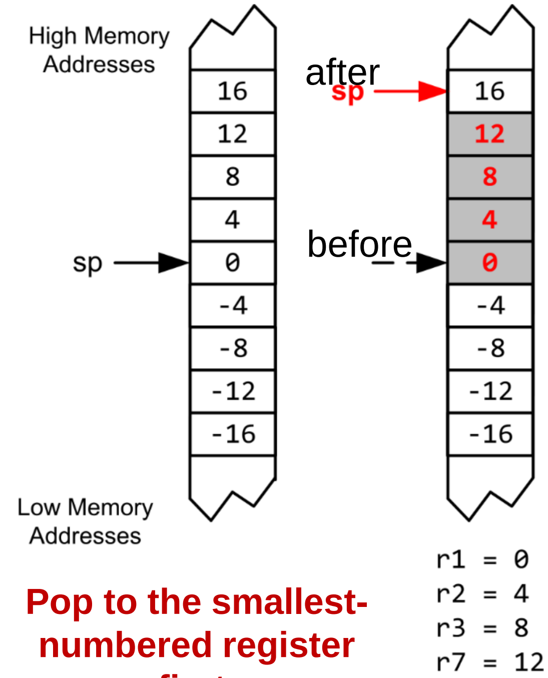
# Stacking & Unstacking

- Largest-numbered register is pushed first but popped last.

PUSH {r3, r1, r7, r2}




POP {r3, r1, r7, r2}




**Pop to the smallest-numbered register first.**

# Example: swap R1 & R2

```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```

| | |
|---|---|
| R1 | 0x11 |
| R2 | 0x22 |
| R13 | 0x80 |

| mem | Addr |
|---|---|
| xxxx | 0x80 |
| xxxx | 0x7C |
| xxxx | 0x78 |

```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```

| | |
|---|---|
| R1 | 0x11 |
| R2 | 0x22 |
| R13 | 0x7C |

| mem | Addr |
|---|---|
| xxxx | 0x80 |
| 0x11 | 0x7C |
| xxxx | 0x78 |

```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```

| | |
|---|---|
| R1 | 0x11 |
| R2 | 0x22 |
| R13 | 0x78 |

| mem | Addr |
|---|---|
| xxxx | 0x80 |
| 0x11 | 0x7C |
| 0x22 | 0x78 |

```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```

| | |
|---|---|
| R1 | 0x22 |
| R2 | 0x22 |
| R13 | 0x7C |

| mem | Addr |
|---|---|
| xxxx | 0x80 |
| 0x11 | 0x7C |
| 0x22 | 0x78 |

```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```

| | |
|---|---|
| R1 | 0x22 |
| R2 | 0x11 |
| R13 | 0x80 |

| mem | Addr |
|---|---|
| xxxx | 0x80 |
| 0x11 | 0x7C |
| 0x22 | 0x78 |

# Outline

- Subroutine
- **Interrupt**

# Handle external events

- Question: Is Pizza delivered?
  - **Polling**:  Open the door every three seconds to check if the delivery person has arrived.
  - **Interrupt**: Do whatever you should do and open the door when the doorbell rings.
- How to execute other codes that Handle external events like I/O, timer, Communication? e.g. turn on LED

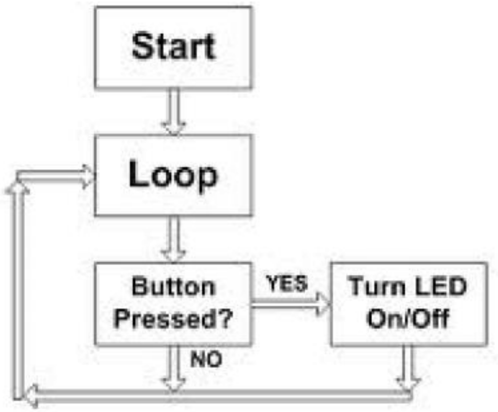
```
// Polling method
while (1) {
    read_button_input;
    if (pushed)
        exit;
}

turn_on_LED;
```

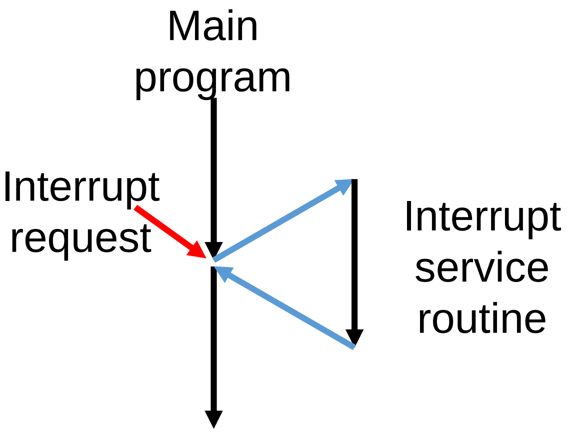
```
// Interrupt method
interrupt_handler(){
 turn_on_LED;
 exit;
}
```

# Polling vs Interrupt

- Polling:
  - The periodic/continuous checking of external events
  - consumes a significant amount of CPU processing time
  - The polling process needs to be combined with other functional code.
  - Since CPU needs to handle other events, critical events may be missed.

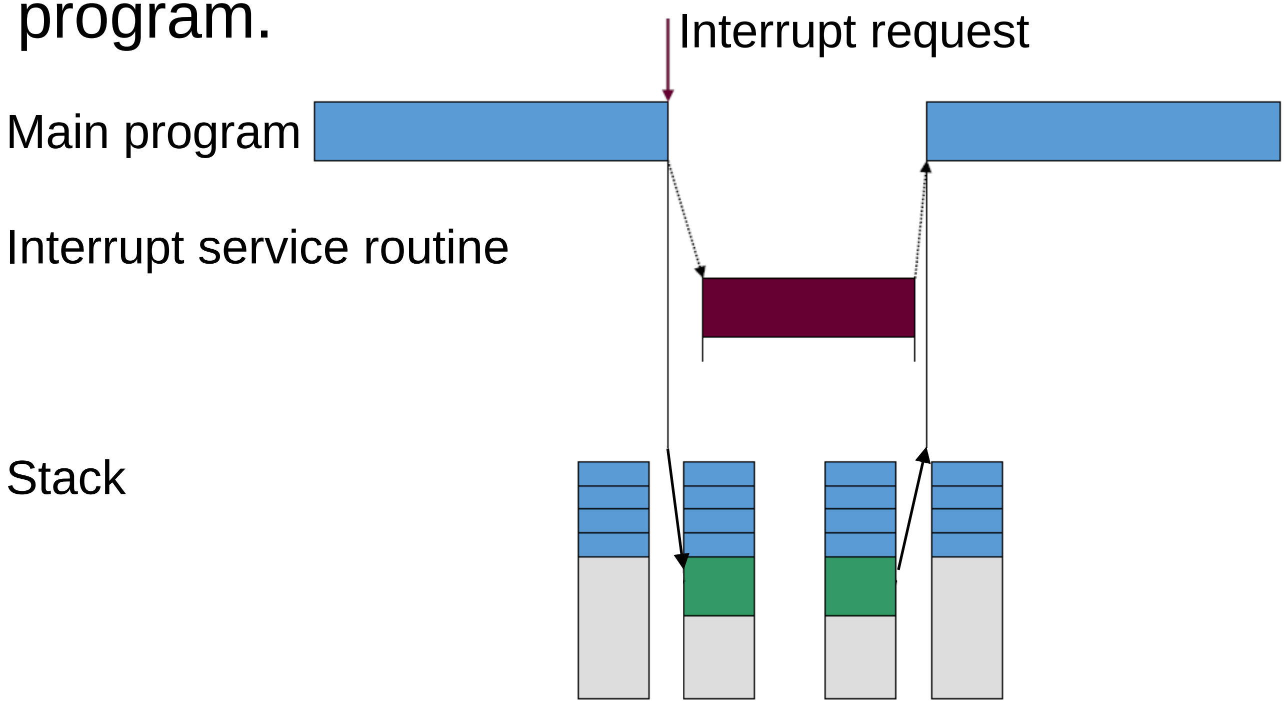



- Interrupt:
  - The hardware determines whether an external event has occurred and notifies the CPU
  - A dedicated interrupt service routine (ISR) is used to handle the event.



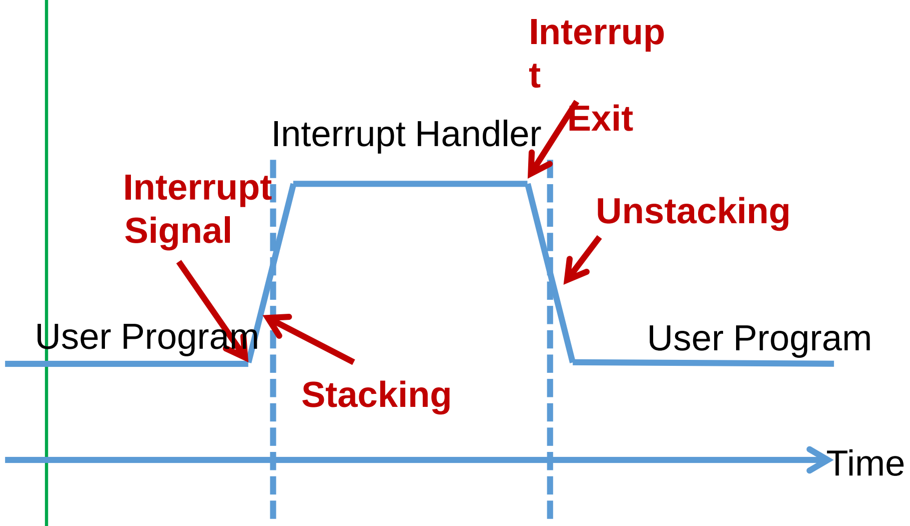
# How CPU Processes the Interrupts

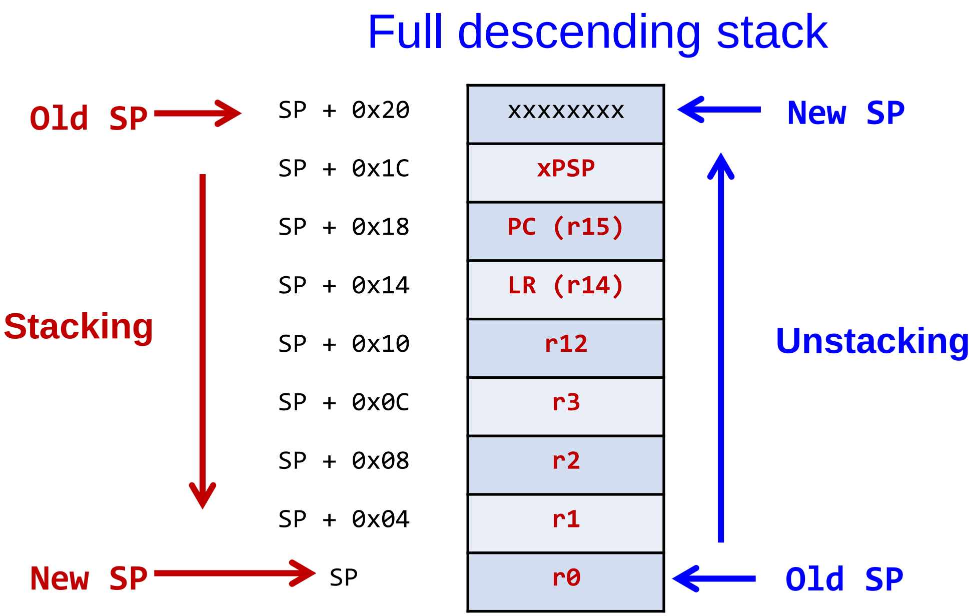
- Finish processing current instruction.
- Save return address and register context to stack.
- Run interrupt service routine.
- Restore return address and register context from stack.
- Resume main program.



# Automatic Stacking & Unstacking

- Stacking: hardware automatically pushes eight register into the stack(xPSR,PC,LR,r12,r3,r2,r1,r0)
- Unstacking: hardware automatically pops these eight register off the stack



# Interrupt Service Routines (ISR)

- Subroutines used to service an interrupt are called ISR.
- Each interrupt has an ISR
- ISR is like a subroutine from program's perspective, but is invoked by the hardware at an unpredictable time
  - Not by the control of the program's logic
- Difference with Subroutine:
  - Program has total control of when to call and jump to a subroutine

# Interrupt Vector Table

- Question: Where to Put ISR Code?
  - Locations of ISRs should be fixed so that the processor can easily find them
  - But, different ISRs may have different lengths
    → hard to track their starting addresses
- Cortex-M solution:
  - A table in memory contains addresses of ISR, the table is called interrupt vector table
  - Processor obtains the subroutine address from the vector table and directs the execution to the ISR. (loads PC with this fixed, pre-defined address)

# Interrupt Vector Table

| Position | Priority | Type of priority | Acronym | Description | Address |
|---|---|---|---|---|---|
| | - | - | - | Reserved | 0x0000_0000 |
| | -3 | fixed | Reset | Reset | 0x0000_0004 |
| | -2 | fixed | NMI | Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector. | 0x0000_0008 |
| | -1 | fixed | HardFault | All class of fault | 0x0000_000C |
| | 0 | settable | MemManage | Memory management | 0x0000_0010 |
| | 1 | settable | BusFault | Prefetch fault, memory access fault | 0x0000_0014 |
| | 2 | settable | UsageFault | Undefined instruction or illegal state | 0x0000_0018 |
| | - | - | - | Reserved | 0x0000_001C - 0x0000_002B |
| | 3 | settable | SVCall | System service call via SWI instruction | 0x0000_002C |
| | 4 | settable | Debug Monitor | Debug Monitor | 0x0000_0030 |
| | - | - | - | Reserved | 0x0000_0034 |
| | 5 | settable | PendSV | Pendable request for system service | 0x0000_0038 |
| | 6 | settable | SysTick | System tick timer | 0x0000_003C |
| 0 | 7 | settable | WWDG | Window watchdog interrupt | 0x0000_0040 |
| 1 | 8 | settable | PVD | PVD through EXTI Line detection interrupt | 0x0000_0044 |
| 2 | 9 | settable | TAMPER | Tamper interrupt | 0x0000_0048 |
| 3 | 10 | settable | RTC | RTC global interrupt | 0x0000_004C |
| ... | | | | | |
| 53 | 60 | settable | UART5 | UART5 global interrupt | 0x0000_0114 |
| 54 | 61 | settable | TIM6 | TIM6 global interrupt | 0x0000_0118 |
| 55 | 62 | settable | TIM7 | TIM7 global interrupt | 0x0000_011C |
| 56 | 63 | settable | DMA2_Channel1 | DMA2 Channel1 global interrupt | 0x0000_0120 |
| 57 | 64 | settable | DMA2_Channel2 | DMA2 Channel2 global interrupt | 0x0000_0124 |
| 58 | 65 | settable | DMA2_Channel3 | DMA2 Channel3 global interrupt | 0x0000_0128 |
| 59 | 66 | settable | DMA2_Channel4_5 | DMA2 Channel4 and DMA2 Channel5 global interrupts | 0x0000_012C |

Table 63. Vector table for other STM32F10xx devices, reference manual

System Exceptions
Defined by ARM

Peripheral Interrupts
Defined by chip vendor

| Interrupt Number | Memory Address | Memory Contents (32 bits) |
|---|---|---|
| ⋮ | ⋮ | ⋮ |
| 13 | 0x00000074 | DMA1_Channel3_IRQHandler |
| 12 | 0x00000070 | DMA1_Channel2_IRQHandler |
| 11 | 0x0000006C | DMA1_Channel1_IRQHandler |
| 10 | 0x00000068 | EXTI4_IRQHandler |
| 9 | 0x00000064 | EXTI3_IRQHandler |
| 8 | 0x00000060 | EXTI2_IRQHandler |
| 7 | 0x0000005C | EXTI1_IRQHandler |
| 6 | 0x00000058 | EXTI0_IRQHandler |
| 5 | 0x00000054 | RCC_IRQHandler |
| 4 | 0x00000050 | FLASH_IRQHandler |
| 3 | 0x0000004C | RTC_WKUP_IRQHandler |
| 2 | 0x00000048 | TAMPER_STAMP_IRQHandler |
| 1 | 0x00000044 | PVD_IRQHandler |
| 0 | 0x00000040 | WWDG_IRQHandler |
| -1 | 0x0000003C | SysTick_Handler |
| -2 | 0x00000038 | PendSV_Handler |
| -3 | 0x00000034 | Reserved |
| -4 | 0x00000030 | DebugMon_Handler |
| -5 | 0x0000002C | SVC_Handler |
| -6 | 0x00000028 | Reserved |
| -7 | 0x00000024 | Reserved |
| -8 | 0x00000020 | Reserved |
| -9 | 0x0000001C | Reserved |
| -10 | 0x00000018 | UsageFault_Handler |
| -11 | 0x00000014 | BusFault_Handler |
| -12 | 0x00000010 | MemManage_Handler |
| -13 | 0x0000000C | HardFault_Handler |
| -14 | 0x00000008 | NMI_Handler |
| | 0x00000004 | Reset_Handler |
| | 0x00000000 | Top_of_Stack |

System Exceptions

```
void DMA1_Channel1_IRQHandler () {
  ...
}
```

```
void EXTI1_Handler () {
  ...
}
```

```
void EXTI0_Handler () {
  ...
}
```

```
void SysTick_Handler () {
  ...
}
```

```
void SVC_Handler () {
  ...
}
```

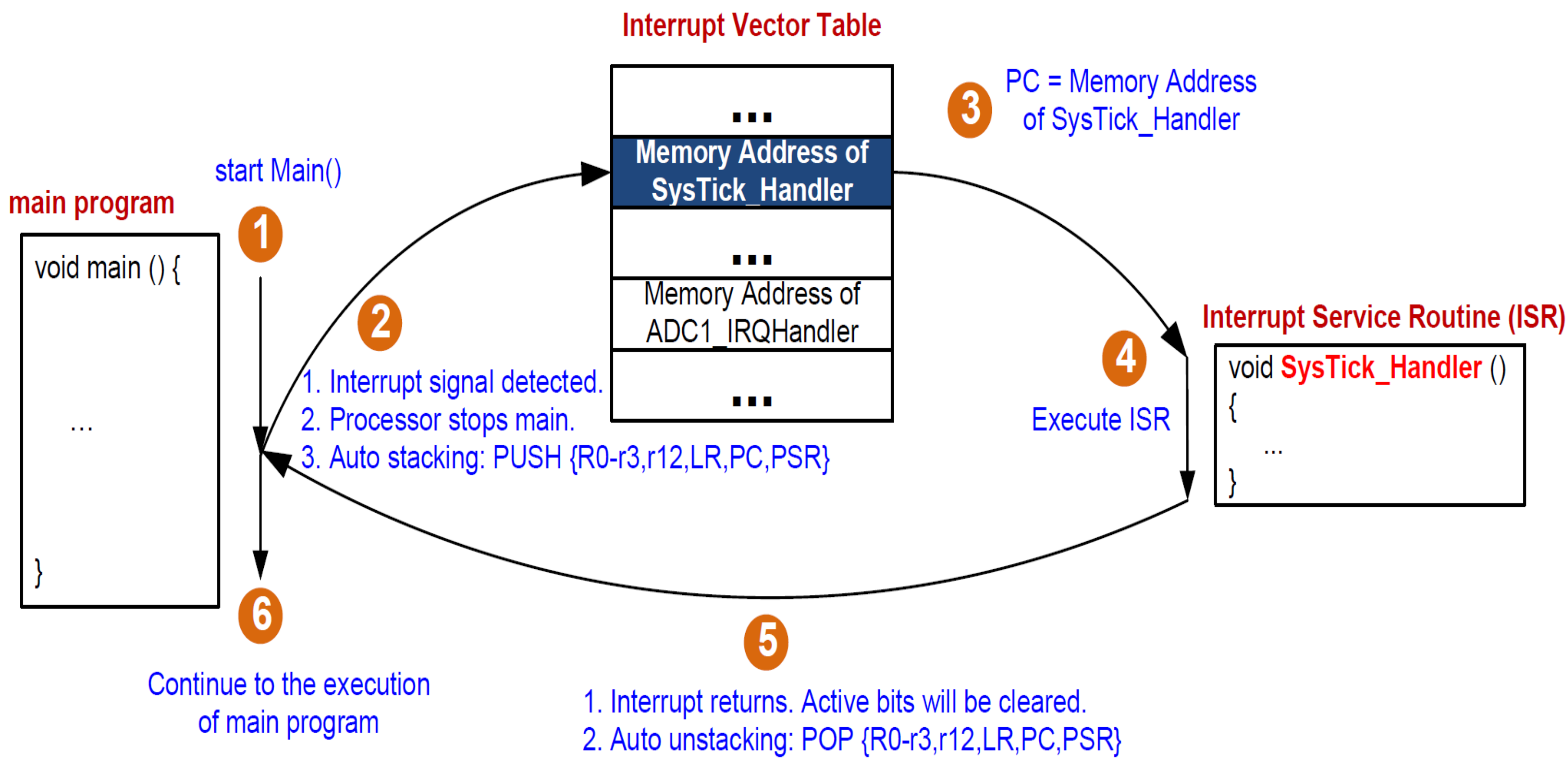
```
void Reset_Handler () {
  ...
  main();
  ...
}
```

Reset_Handler: Value to initialize the Program Counter (PC)

Top_of_Stack: Value to initialize the Stack Pointer (SP)

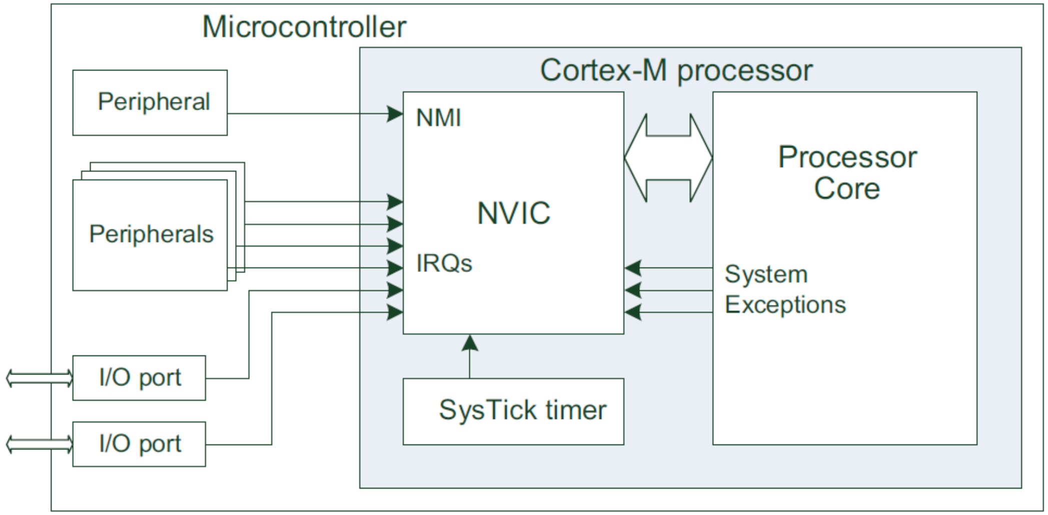
# An Interrupt Process Example

- Example: SysTick interrupt process



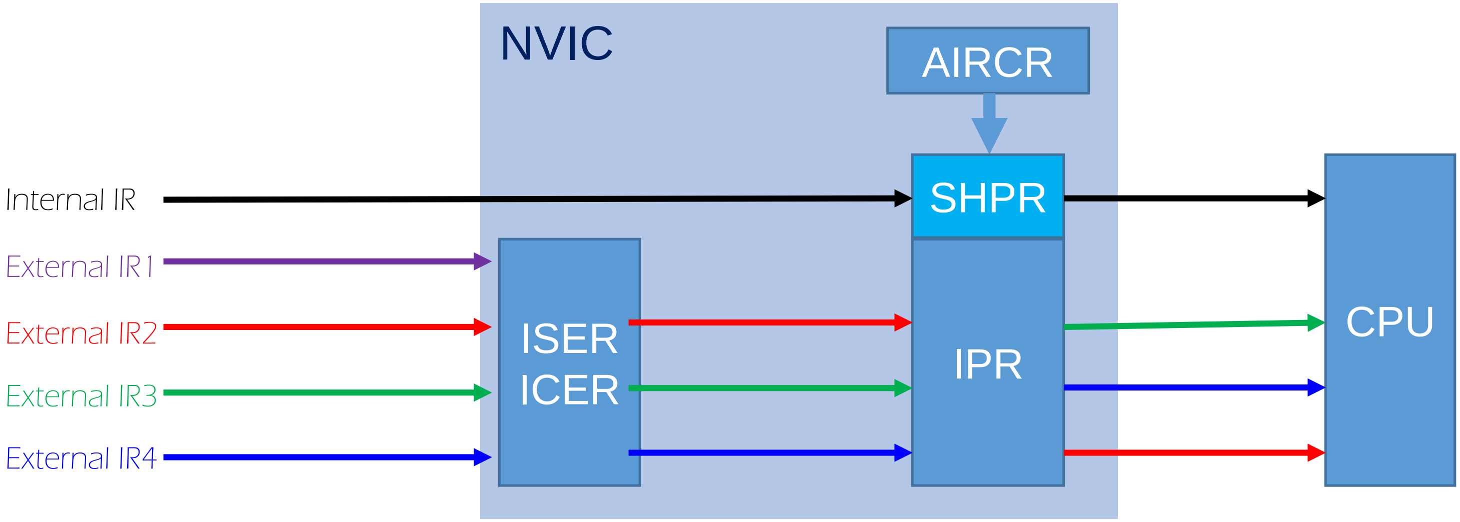
# Interrupts management

- Types of Interrupts
  - Interrupts from peripherals modules
  - External pin interrupts ( IRQ0 to IRQ15)
  - Software interrupts
  - Non Maskable interrupts (Reset pin)
- Interrupts are managed by Nested Vectored Interrupt Controller (NVIC)
  - NVIC receives interrupt requests from various sources



# NVIC Registers

- Program AIRCR register to set priority group
- Program IPRx register to set priority value
  - Fixed priority for Reset, HardFault, and NMI.
  - Adjustable for all the other interrupts
- Enable/Disable interrupts with ISER/ICER registers



# Interrupt Priority

- Priority:
  - preempt priority(抢占优先级) number: determines the order of execution among different interrupt sources
  - sub-priority(响应优先级) number: Resolves conflicts between interrupts of the same preempt priority level
  - natural-priority(自然优先级): The priority within the interrupt vector table.
- Principles
  - Smaller value = higher priority
  - higher preemption priority can interrupt a lower preemption priority
  - When preemption priorities are the same, higher sub-priority executes first, but cannot preempt each other
  - When preemption and sub-priorities are the same, the one with the higher natural priority executes first.
- Reset has highest priority

# Priority Group Register (AIRCR)

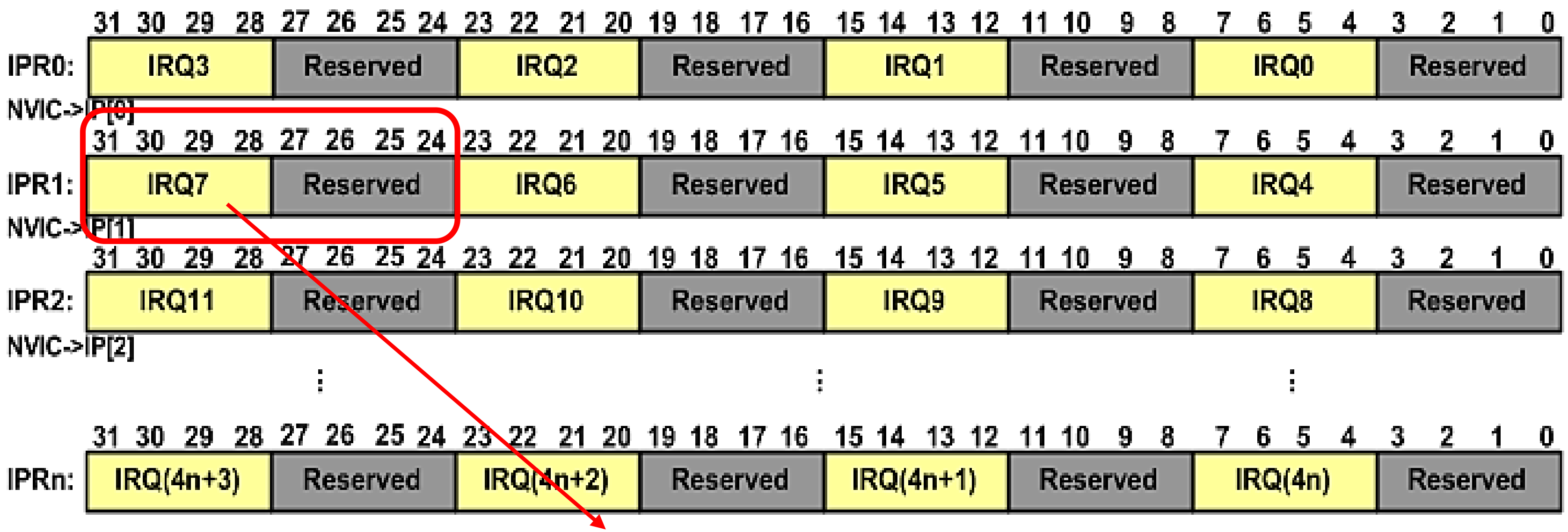
- Priority group setting (# of bit partition for preempt and sub-priority) can be adjusted by AIRCR register
- preempt priority and sub-priority value can be set by IPRx register

| Priority group | AIRCR[10:8] | IPRx bit[7:4] partition | Result |
|---|---|---|---|
| **0** | 111 | None ： [7:4] | 0 bit for preempt priority, 4 bits for sub priority |
| **1** | 110 | [7] ： [6:4] | 1 bit for preempt priority, 3 bits for sub priority |
| **2** | 101 | [7:6] ： [5:4] | 2 bit for preempt priority, 2 bits for sub priority |
| **3** | 100 | [7:5] ： [4] | 3 bit for preempt priority, 1 bits for sub priority |
| **4** | 011 | [7:4] ： None | 4 bit for preempt priority, 0 bits for sub priority |

Default: 2 bits for preemption, 2 for sub-priority

```
HAL_NVIC_SetPriorityGrouping(n);
```

# Interrupt Priority Register (IPRx)




EXTI1_IRQn = 7

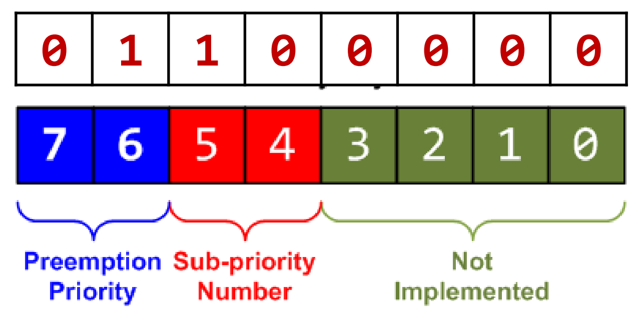
- Using the IPR register

```
NVIC->IP[7] = (6 << 4) & 0xff;
```

- Using the function

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn,
uint32_t PreemptPriority, uint32_t SubPriority);

HAL_NVIC_SetPriority(7, 1, 2);
```




With default priority group

# Example

- Example: Determine execution order of the following interrupt

| IRQ # | Priority Level | Interrupt | Preemption | Sub-Priority | Execution Order |
|---|---|---|---|---|---|
| 3 | 10 | RTC | 2 | 1 | 2 |
| 6 | 13 | EXTI0 | 3 | 0 | 4 |
| 7 | 14 | EXTI1 | 2 | 0 | 1 |
| -1 | 6 | Systick | 3 | 0 | 3 |

- EXTI1 and RTC can obtain priority execution during EXTI0 and Systick interrupts, as the preempt priority is higher

# Enabling Peripheral Interrupts (ISER)

- Example: Enable TIM7 Interrupt

EN0 (ISER[0]):

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

EN1 (ISER[1]):

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 |

TIM7_IRQn = 44

- Using the ISER register

```
NVIC->ISER[1] = 1 << 12;     // Enable TIM7 (bit 12 of ISER[1])
```

- Using the function

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);

HAL_NVIC_EnableIRQ(TIM7_IRQn); // Enable TIM7, IRQn = 44
```

# Disabling Peripheral Interrupts (ICER)

- Example: Disable TIM7 Interrupt

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DIS0 (ICER[0]): | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DIS1 (ICER[1]): | 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 |

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 |

TIM7_IRQn = 44

- Using the ICER register

```
NVIC->ICER[1] = 1 << 12;     // Disable TIM7 (bit 12 of ISER[1])
```

- Using the function

```
void HAL_NVIC_DisableIRQ(IRQn_Type IRQn);

HAL_NVIC_DisableIRQ(TIM7_IRQn); // Disable TIM7, IRQn = 44
```

# Interrupt vs. subroutine call

| BL | Interrupt |
|---|---|
| Jumps to any location | Jumps to a fixed location |
| BL is used by the programmer in the sequence of instruction | hardware interrupt can come in at any time |
| cannot be masked | Can be masked (disabled) |
| Saves only LR register (Value of PC) | Saves CPSR, PC, LR, R12, R3, R2, R1, and R0. |
| CPU mode remains unchanged | CPU goes to Handler mode |
| On return, restores LR register (Value of PC) | On return, restores CPSR, R15, R14, R12, R3–R0. |