香港中文大學
The Chinese University of Hong Kong

CENG2400 Embedded System Design

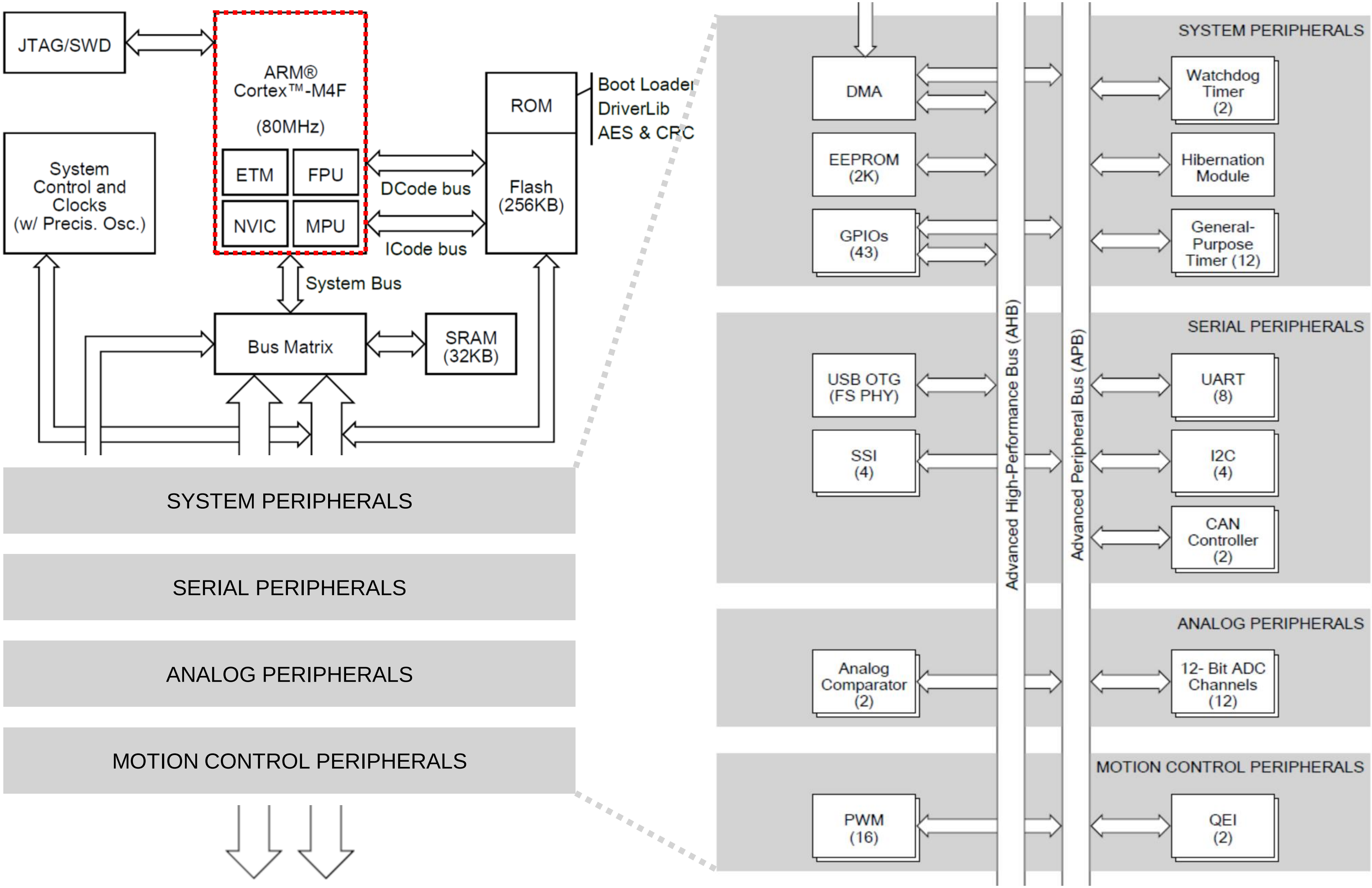
Lecture 05: Processor Core

Ming-Chang YANG

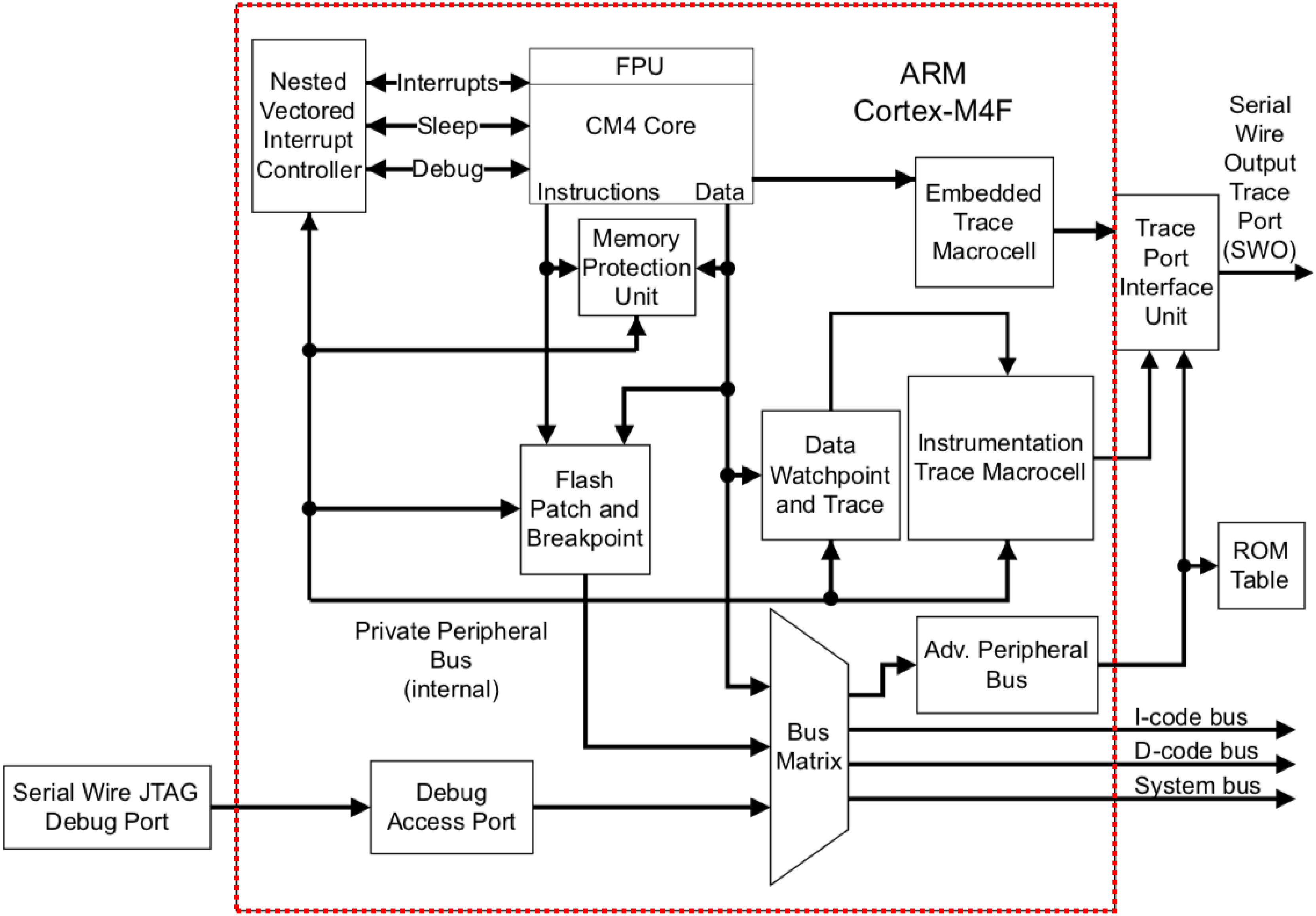
mcyang@cse.cuhk.edu.hk

Thanks to Prof. Q. Xu and Drs. K. H. Wong, Philip Leong, Y.S. Moon, O. Mencer, N. Dulay, P. Cheung for some of the slides used in this course!

Recall: Tiva™ TM4C123GH6PM (MCU)



Recall: ARM Cortex-M4F (μProcessor)

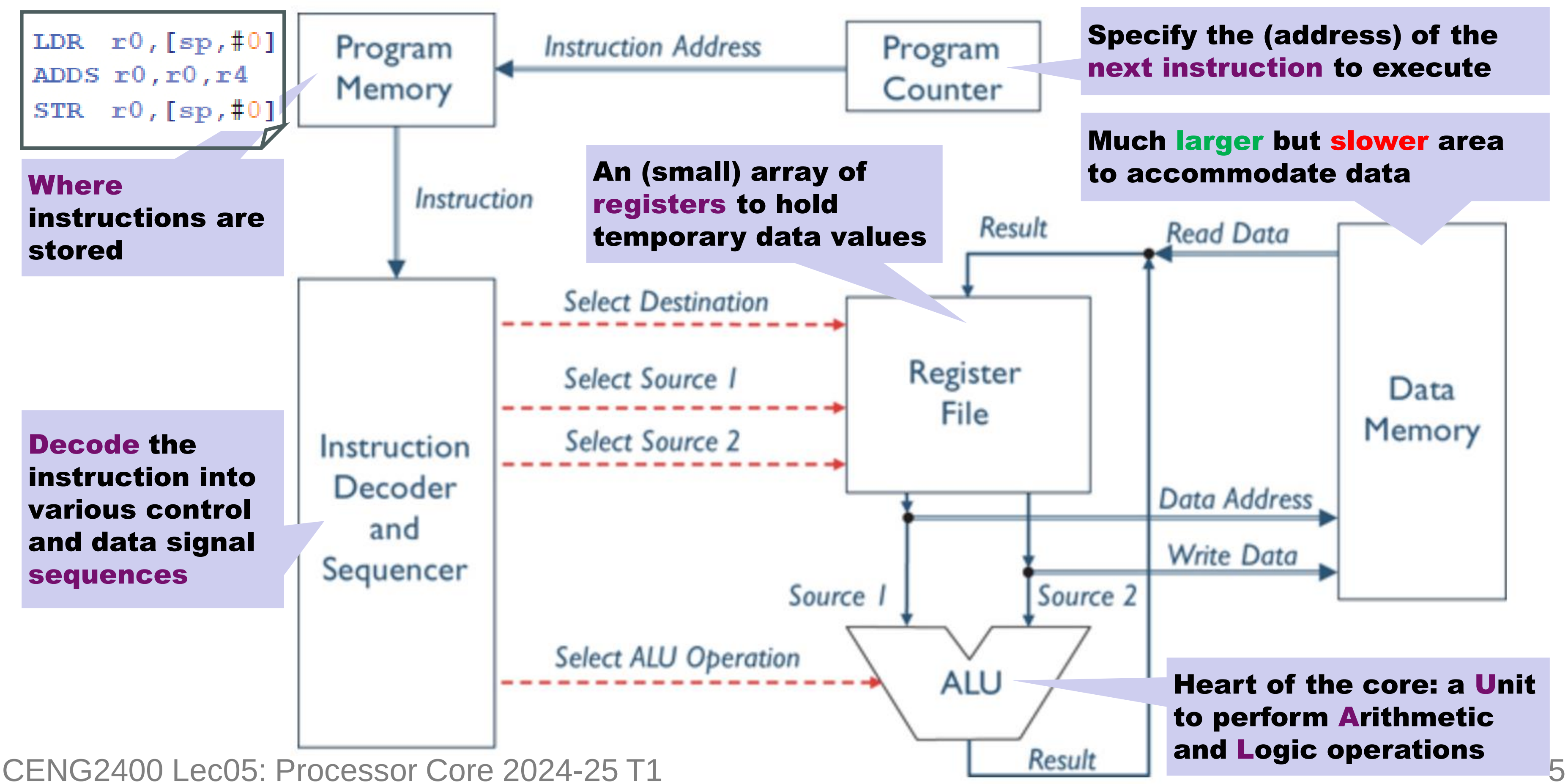

ARM
Cortex-M4F
Nested Vectored Interrupt Controller
Interrupts
Sleep
Debug
FPU
CM4 Core
Instructions
Data
Memory Protection Unit
Embedded Trace Macrocell
Serial Wire Output Trace Port (SWO)
Trace Port Interface Unit
Data Watchpoint and Trace
Instrumentation Trace Macrocell
Flash Patch and Breakpoint
ROM Table
Private Peripheral Bus (internal)
Adv. Peripheral Bus
Bus Matrix
I-code bus
D-code bus
System bus
Serial Wire JTAG Debug Port
Debug Access Port

Outline

- **Processor Core**
  - A Simplified View
  - Sequential Execution
  - Architecture Profile
  - Programming Model: Load/Store
  - Registers
  - Memory
  - Instructions
    - Condition Code Flags
    - Branch & Subroutine
  - Exception/Interrupt Handling

Processor Core: A Simplified View (1/2)

- Core executes **instructions** that constitute a program.
  - Each instruction specifies an **operation** to perform and which **operands** (i.e., parameters) to be involved.



```
LDR  r0,[sp,#0]
ADDS r0,r0,r4
STR  r0,[sp,#0]
```

Where instructions are stored

Specify the (address) of the next instruction to execute

Much larger but slower area to accommodate data

An (small) array of registers to hold temporary data values

Decode the instruction into various control and data signal sequences

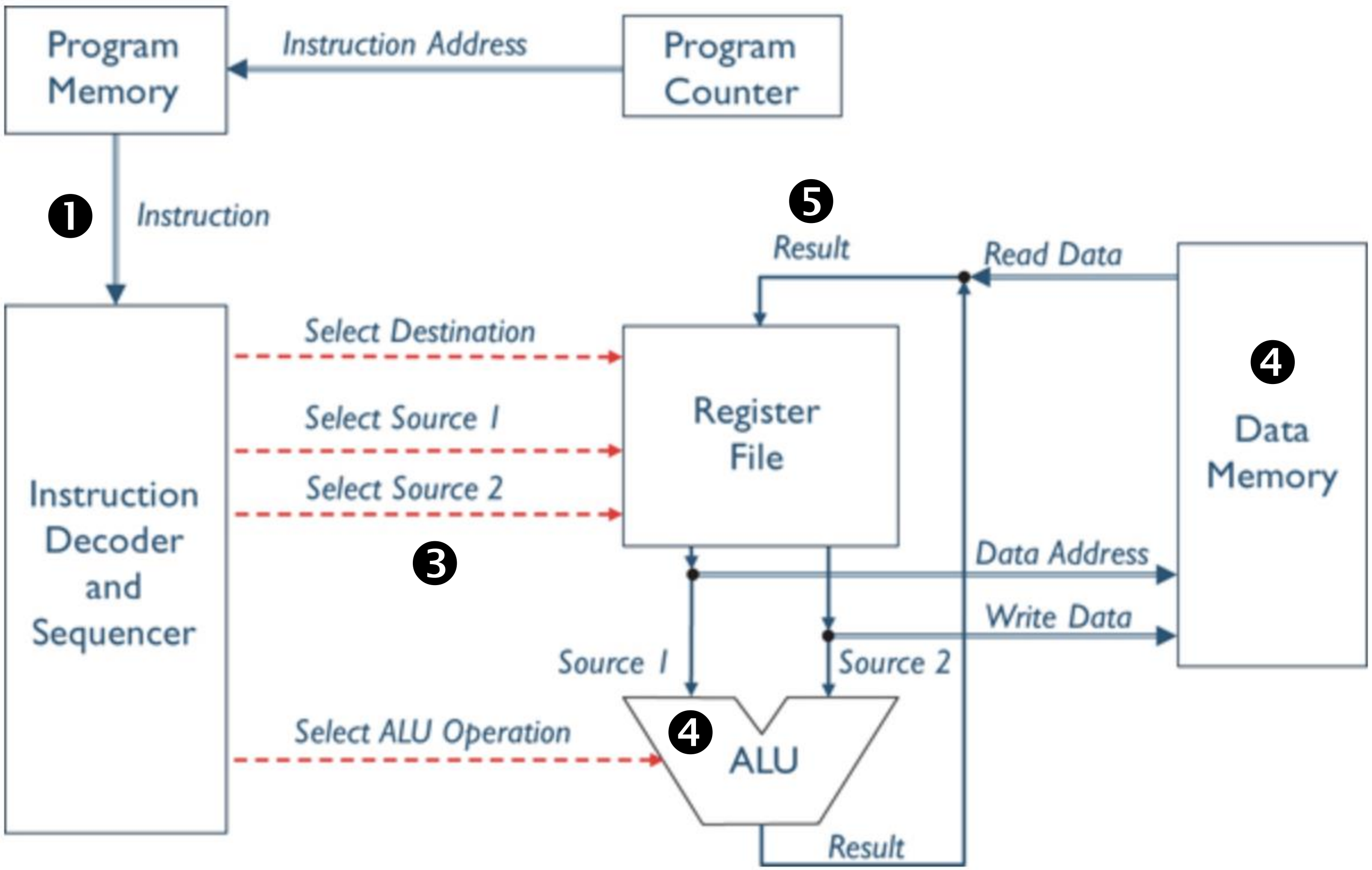
Heart of the core: a Unit to perform Arithmetic and Logic operations

Processor Core: A Simplified View (2/2)

- The typical sequence to process an instruction:
  - ❶ Fetch instruction; ❷ Decode instruction; ❸ Get source operands; ❹ Perform an operation *or* access memory; ❺ Write back the result.

Note: Not all the steps are necessary for every instruction.

```
LDR  r0,[sp,#0]
ADDS r0,r0,r4
STR  r0,[sp,#0]
```

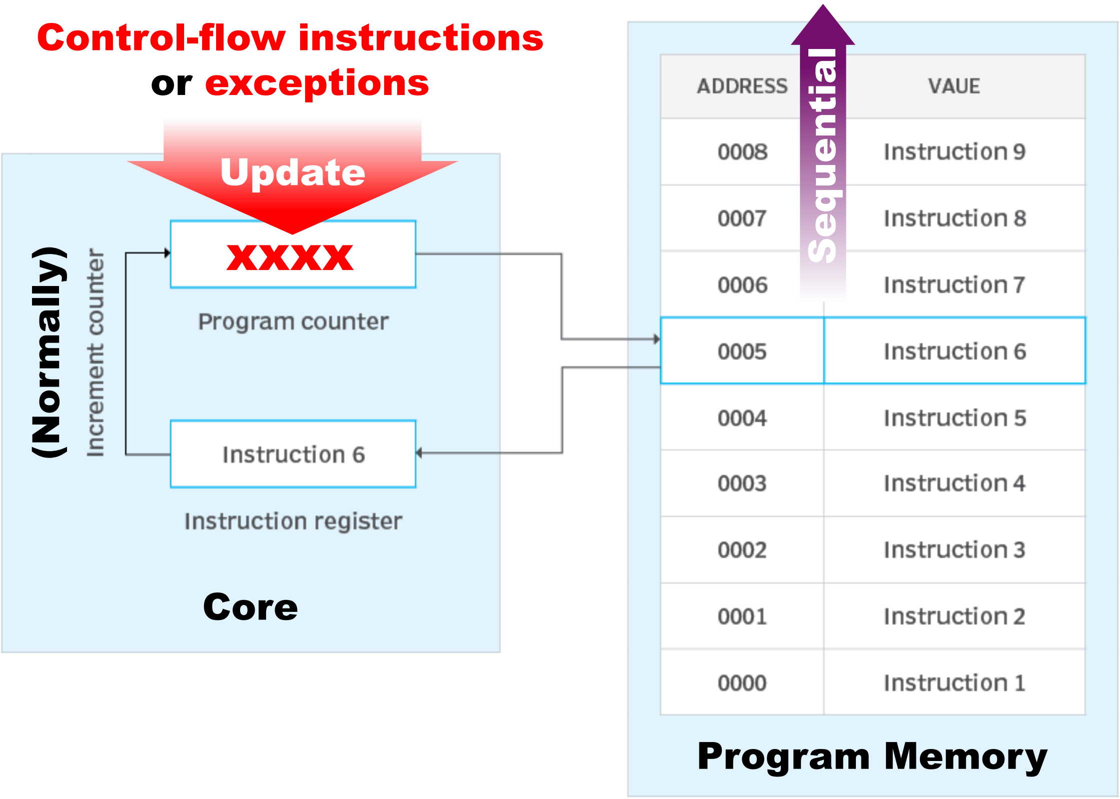


Outline

- **Processor Core**
  - A Simplified View
  - Sequential Execution
  - Architecture Profile
  - Programming Model: Load/Store
  - Registers
  - Memory
  - Instructions
    - Condition Code Flags
    - Branch & Subroutine
  - Exception/Interrupt Handling

Sequential Execution

- Programs normally follow a sequential execution.
  - By advancing to the instruction located immediately after.
- However, control-flow instructions (such as `branch`, `call`/`return`) or exceptions will break the seq. flow.
  - This makes the flow jump to a different location via the update of program counter.
    - This enables loops, condition tests, subroutine calls, and many other behaviors.



ADDRESS	VAUE
0008	Instruction 9
0007	Instruction 8
0006	Instruction 7
0005	Instruction 6
0004	Instruction 5
0003	Instruction 4
0002	Instruction 3
0001	Instruction 2
0000	Instruction 1

Outline

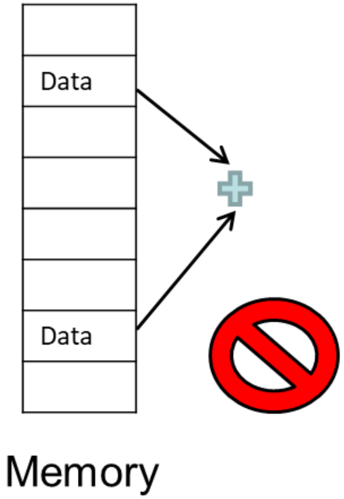
- **Processor Core**
  - A Simplified View
  - Sequential Execution
  - Architecture Profile
  - Programming Model: Load/Store
  - Registers
  - Memory
  - Instructions
    - Condition Code Flags
    - Branch & Subroutine
  - Exception/Interrupt Handling

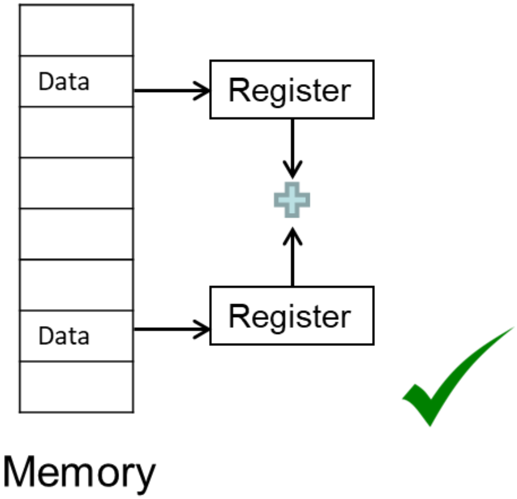
Architecture Profile

- The Cortex-M4F processor implements the ARMv7-M architecture profile:
  - **Armv7-A**: The application profile for systems supporting the Arm and Thumb instruction sets, and requiring virtual address support in the memory management model.
  - **Armv7-R**: The real-time profile for systems supporting the Arm and Thumb instruction sets, and requiring physical address only support in the memory management model.
  - **Armv7-M**: The microcontroller profile for systems supporting only the Thumb instruction set, and where overall size and deterministic operation for an implementation are more important than absolute performance.

Programming Model: Load/Store Arch.

- **Load/Store Architecture**: *Data must be `Loaded` into registers before processing and perhaps `stored` back to memory afterward.*
  - That is, only the data located in registers can be processed, while data in memory cannot be processed directly.
  - This architecture simplifies the hardware, generally increasing speed and reducing power consumption.



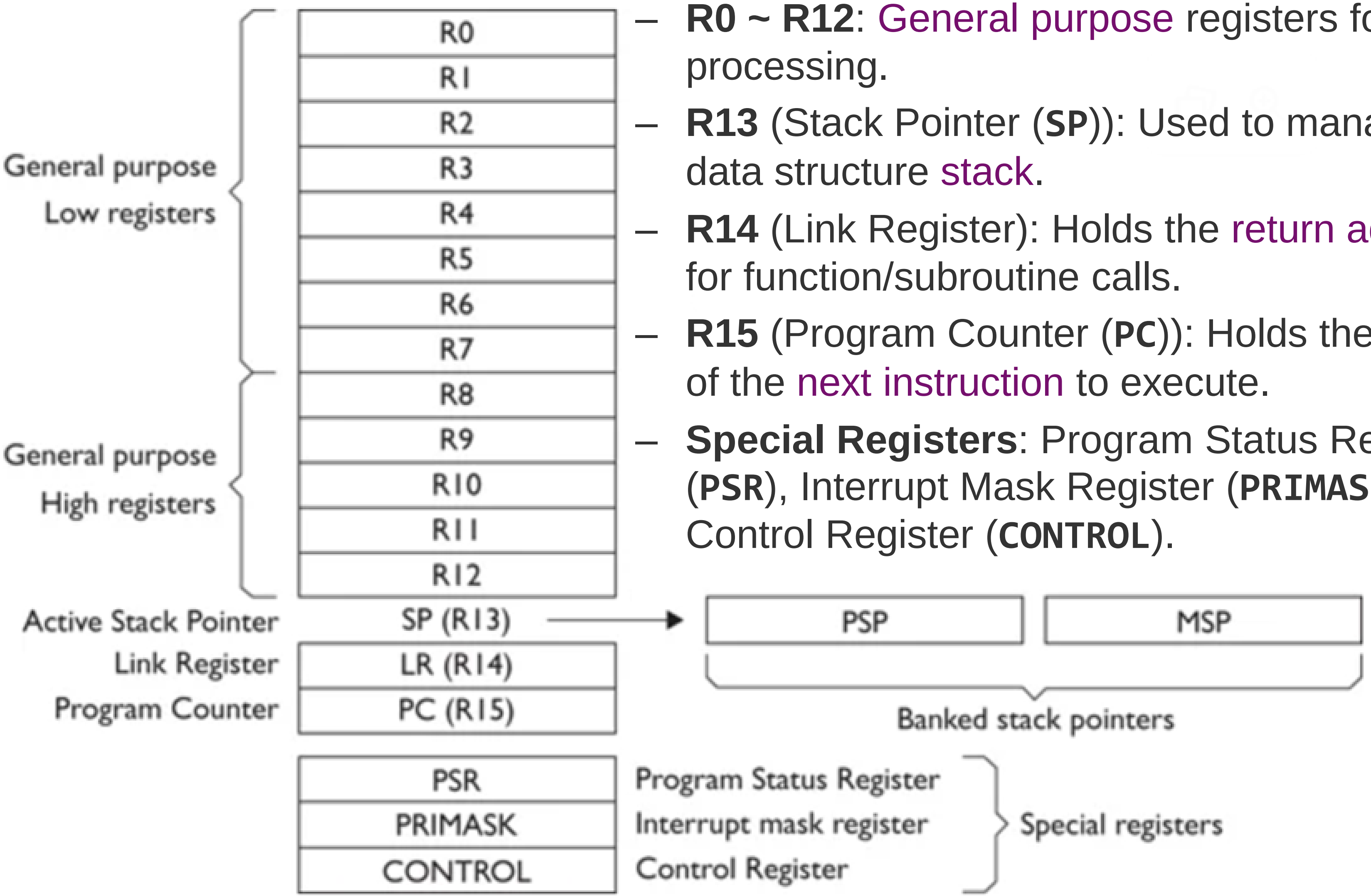


Outline

- **Processor Core**
  - A Simplified View
  - Sequential Execution
  - Architecture Profile
  - Programming Model: Load/Store
  - Registers
  - Memory
  - Instructions
    - Condition Code Flags
    - Branch & Subroutine
  - Exception/Interrupt Handling

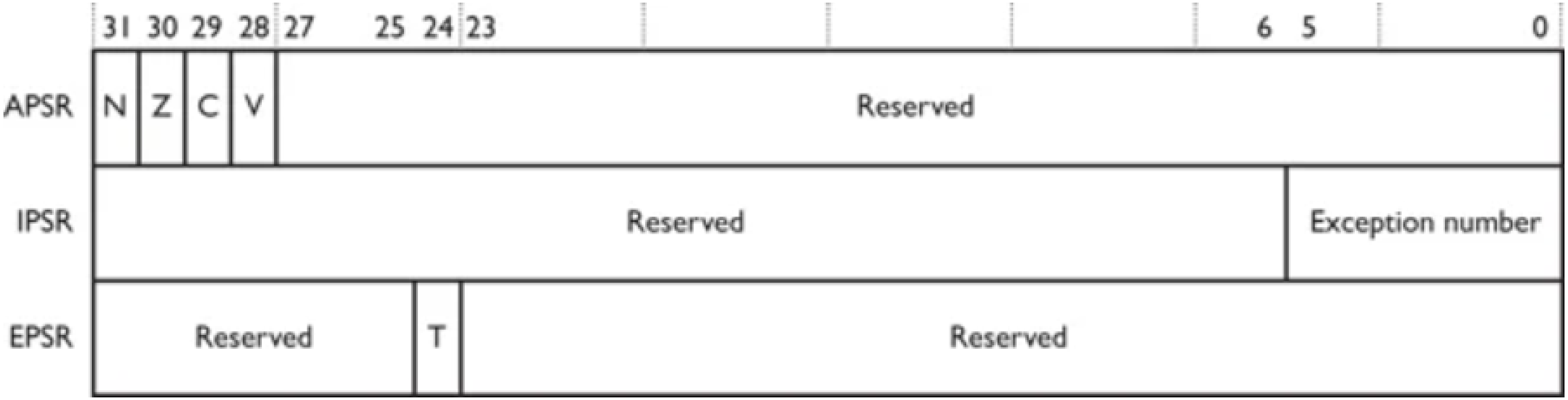
Registers

- ARM features multiple 32-bit registers:
  - R0 ~ R12: General purpose registers for data processing.
  - R13 (Stack Pointer (**SP**)): Used to manage the data structure stack.
  - R14 (Link Register): Holds the return address for function/subroutine calls.
  - R15 (Program Counter (**PC**)): Holds the address of the next instruction to execute.
  - **Special Registers**: Program Status Register (**PSR**), Interrupt Mask Register (**PRIMASK**), and Control Register (**CONTROL**).



Program Status Register (PSR)

- **PSR** is one register but has three different views:
  - **Application PSR View**: Shows the condition code flag bits (Negative, Zero, Carry, and Overflow), which are set by the instructions based on their result.
  - **Interrupt PSR View**: Holds the exception number of the currently executing exception handler.
  - **Execution PSR View**: Indicates whether the CPU is operating in Thumb mode.



Outline

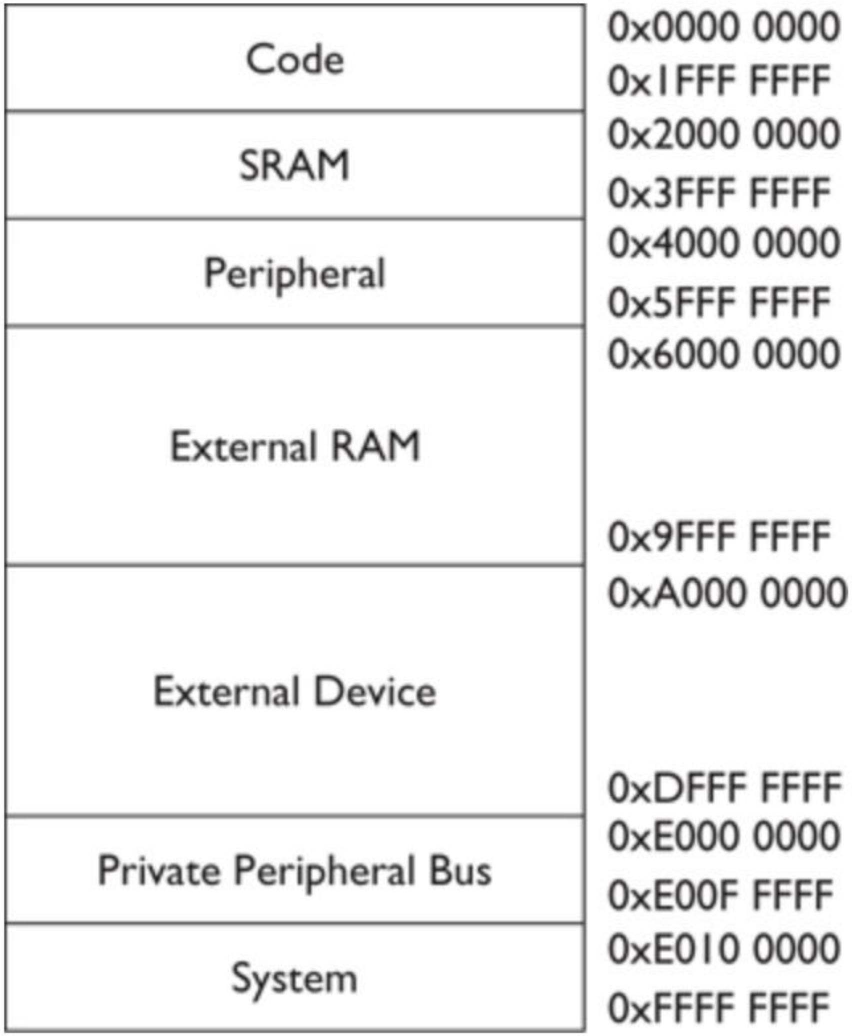
- **Processor Core**
  - – A Simplified View
  - – Sequential Execution
  - – Architecture Profile
  - – Programming Model: Load/Store
  - – Registers
  - – Memory
  - – Instructions
    - Condition Code Flags
    - Branch & Subroutine
  - – Exception/Interrupt Handling

Memory – Memory Map

- Cortex-M has a 32-bit address space:
  - – Allow up to 2^{32} locations to be addressed, each specifies a particular **byte**.
    - This is called byte-addressable.

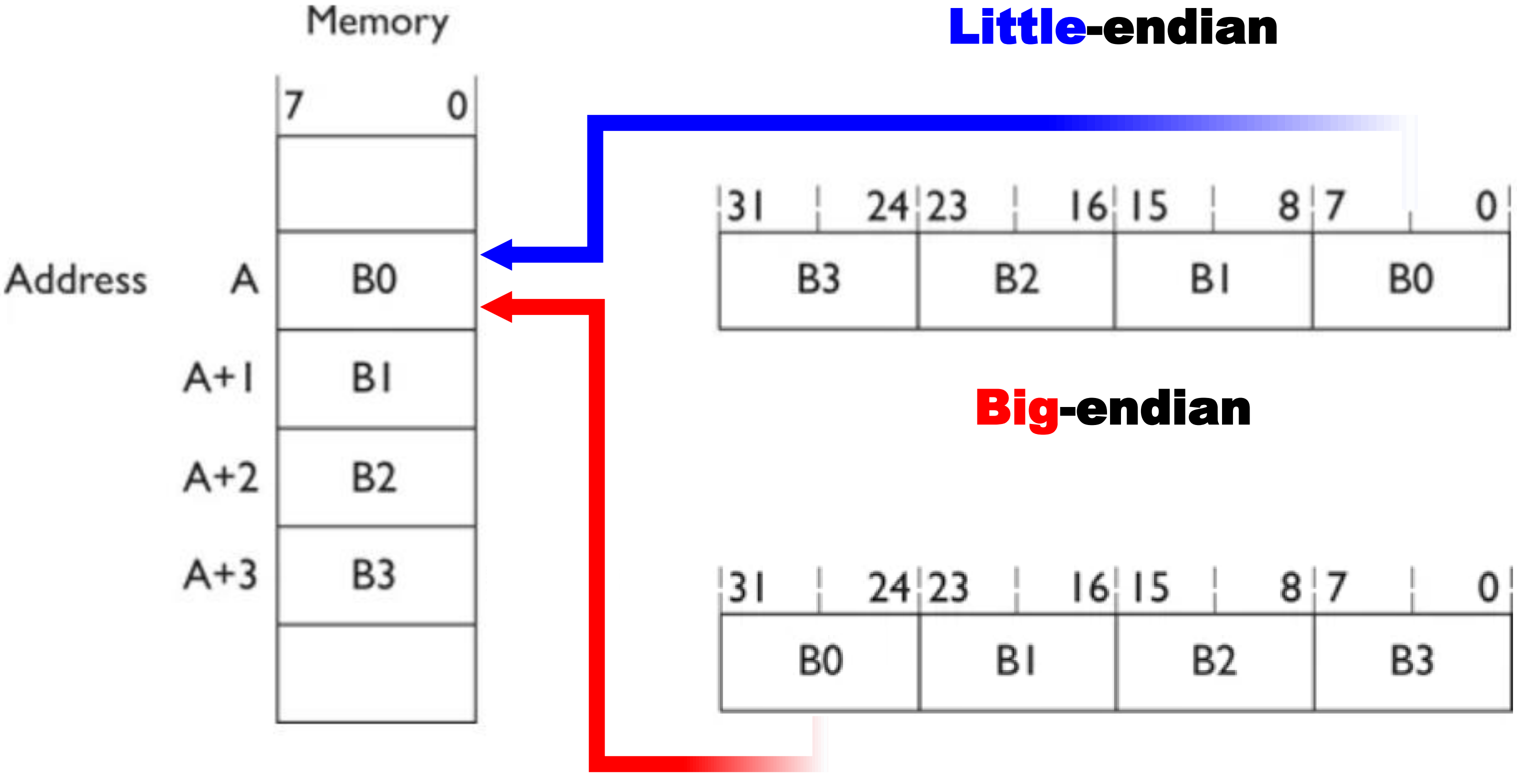
> **Byte-Addressable:** Memory in which each address identifies a single byte.

  - – This address space is divided into various regions for different uses.



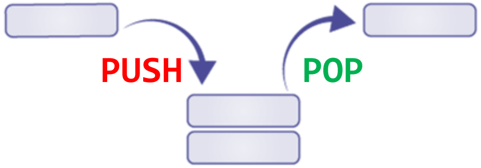
Memory – Endianness

- **Endianness** describes the order in which "multi-byte" values are stored in memory at a range of addresses.
  - **Little-endian**: lowest address holds least-significant byte.
  - **Big-endian**: lowest address holds most-significant byte.

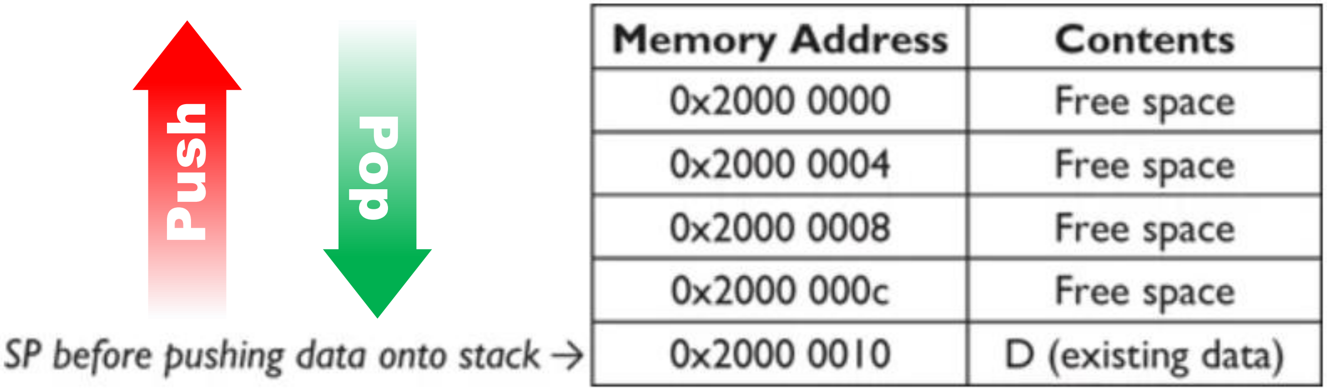


Memory – Processor Stack

- A **processor stack** allows the program to allocate a location (in the memory), use and free it conveniently.
  - Stacks follows a “Last-In, First-Out” data manipulation.
    - **PUSH**: Adding data onto the stack.
    - **POP**: Removing data from the top of the stack.



  - **Stack Pointer (SP)** points to the last item on the stack.
    - Pushing data decreases the SP value by the number of data bytes.
    - Popping data increases SP value by the number of bytes removed.
    - Note: For ARM Cortex-M, all pushes and pops use 32-bit data items.



Memory Address	Contents
0x2000 0000	Free space
0x2000 0004	Free space
0x2000 0008	Free space
0x2000 000c	Free space
0x2000 0010	D (existing data)

SP before pushing data onto stack →

Class Exercise 5.1

- Assuming that SP is `0x20000010` initially, what is its value after executing a PUSH operation?

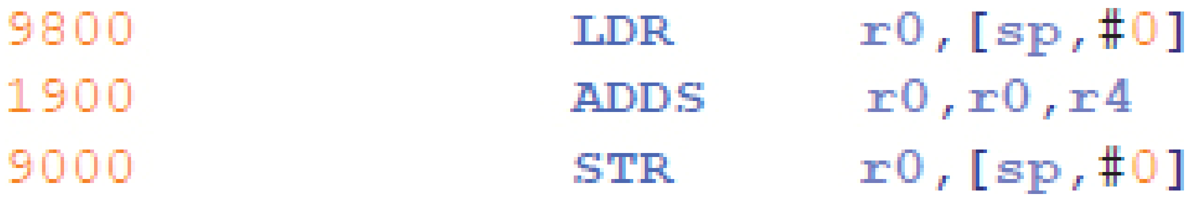
Outline

- **Processor Core**
  - A Simplified View
  - Sequential Execution
  - Architecture Profile
  - Programming Model: Load/Store
  - Registers
  - Memory
  - Instructions
    - Condition Code Flags
    - Branch & Subroutine
  - Exception/Interrupt Handling

Instructions

- An **instruction** specifies which operation to perform, and operands (i.e., parameters) for it.
  - **Machine Language**: Code in which each instruction is represented as a numerical value, which can be processed directly by the processor core.
  - **Assembly Language**: Human-readable representation of machine code.

Machine Language **Assembly Language**

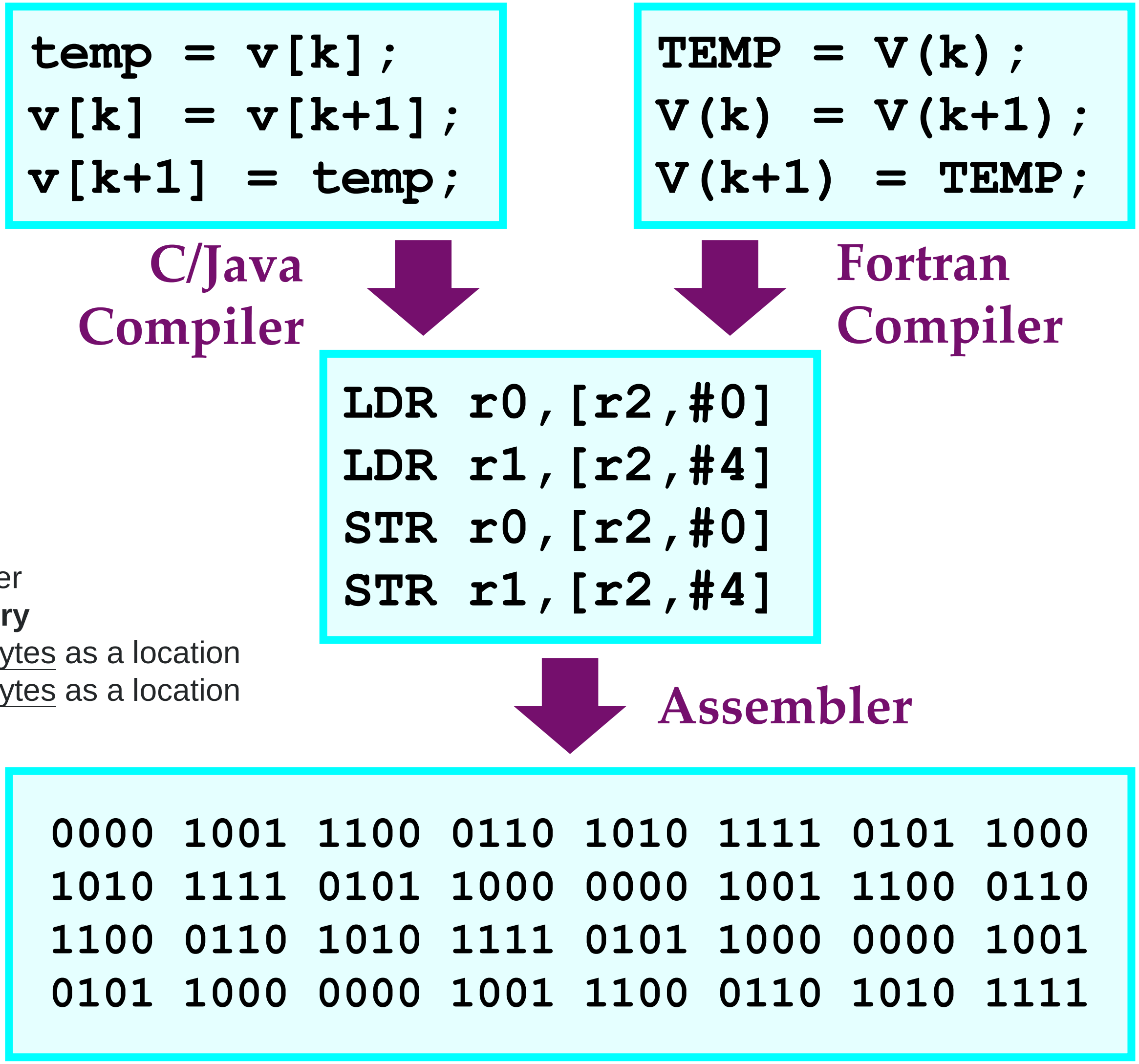


```
9800        LDR     r0,[sp,#0]
1900        ADDS    r0,r0,r4
9000        STR     r0,[sp,#0]
```

> **Assembler**: Software tool which translates assembly language code into machine code.

An Example of “Language Translation”

High-level Language



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
TEMP = V(k);
V(k) = V(k+1);
V(k+1) = TEMP;
```

C/Java Compiler

Fortran Compiler

```
LDR r0,[r2,#0]
LDR r1,[r2,#4]
STR r0,[r2,#0]
STR r1,[r2,#4]
```

Assembler

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Assembly Language

`LDR`: loads a word from **memory** into a register
`STR`: saves a word from a register into **memory**
`[r2,#0]`: treats the value of register r2 + 0 bytes as a location
`[r2,#4]`: treats the value of register r2 + 4 bytes as a location

Machine Language

https://gerardnico.com/code/lang/machine
https://clip2art.com/explore/Boy%20clipart%20teacher/

Assembly Instruction Format

- **`<operation> <operand1> <operand2> <operand3>`**
  - There may be fewer operands.
  - First operand is typically destination.
  - Other operands are sources.

  - Where can the operands be located?
    - In a general-purpose register (**`Rn`**);
    - In memory (*only for **`Load/store`** for accessing the memory and **push/pop** for manipulating the processor stack*);
    - As an immediate value (**`#`**) encoded in instruction word;
    - As a PC-relative addressing (*for **`branch`***).

Instruction Set Examples

- **Data Processing:**

```
MOV  r2, r5                 ; r2 = r5
ADD  r5, #0x24              ; r5 = r5 + 36
ADD  r2, r3, r4, LSL #2 ; r2 = r3 + (r4 * 4)
LSL  r2, #3                 ; r2 = r2 * 8
MOVT r9, #0x1234            ; upper half-word of r9 = #0x1234
MLA  r0, r1, r2, r3         ; r0 = (r1 * r2) + r3
```

- **Memory Access:**

```
STRB r2, [r10, r1]           ; store lower byte in r2 at
                               address {r10 + r1}
LDR r0, [r1, r2, LSL #2] ; load r0 with data at address
                               {r1 + r2 * 4}
```

- **Program Flow:**

```
BL   <label>                ; PC relative branch to <label>, and
                              return address stored in LR (r14)
```

Full vs. Thumb Instruction Set

- In ARM, the **Full** instruction set uses 32 bits to represent each instruction.
  - This makes programs larger and raises costs, which are often crucial for embedded systems.

- The ARMv7-M profile only supports the **Thumb** instruction set, a subset of the full instruction set.
  - Most instructions are represented as 16-bit half-words.
  - This reduces program memory requirements significantly and usually allows instructions to be fetched faster.
  - The limitation is that there are fewer bits available to represent the operation and operands: some 32-bit instructions and advanced features are not available.

Outline

- **Processor Core**
  - A Simplified View
  - Sequential Execution
  - Architecture Profile
  - Programming Model: Load/Store
  - Registers
  - Memory
  - Instructions
    - Condition Code Flags
    - Branch & Subroutine
  - Exception/Interrupt Handling

Condition Code Flags (1/2)

- Instructions typically generate numerical results, such as *positive*, *negative*, or *zero*.

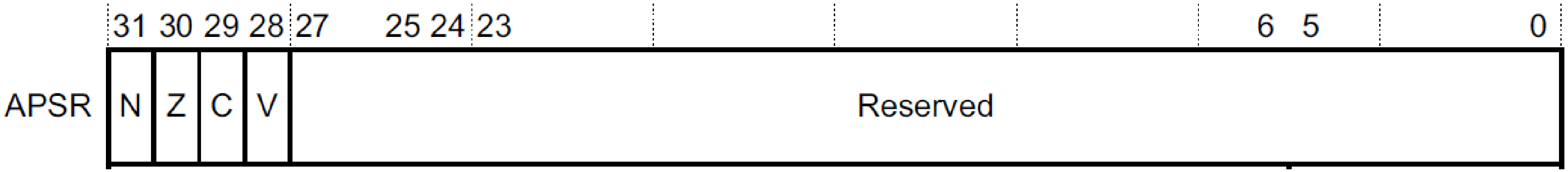
- **Condition Code Flags**: keep the information about the results of the “most recent” instruction.
  - The subsequent instruction may use them conveniently for different purposes.

Common Condition Flags

N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise; otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out occurs; otherwise, cleared to 0

Condition Code Flags (2/2)

- **Recall**: **PSR** is one register with three different views:
  - **Application PSR View (APSR)**: Shows the condition code flag bits (Negative, Zero, Carry, and Overflow), which are set by the instructions based on their result.

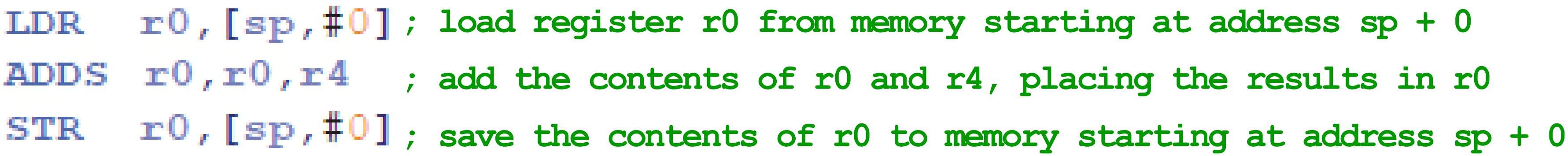


- Some instructions have two variants: one which updates the condition codes and one which does not.
  - “**S**” suffix indicates the instruction updates **APSR**.
    - For example: **ADD** vs. **ADDS**

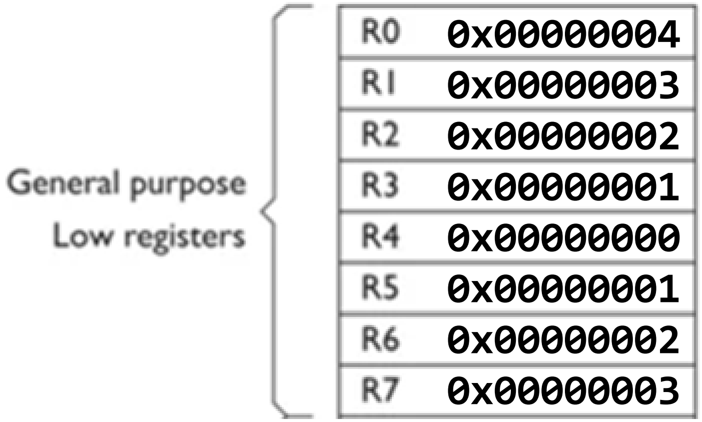
- **CMN** and **CMP** also set the condition code flags.

Class Exercise 5.2

- Consider the contents of registers and memory below:
  - ① Determine the content of `R0` after the execution of the following assembly code snippet.
  - ② Determine the values of condition code flags (`N`, `Z`, `C`, `V`) after the execution of the `ADDS r0,r0,r4` instruction.

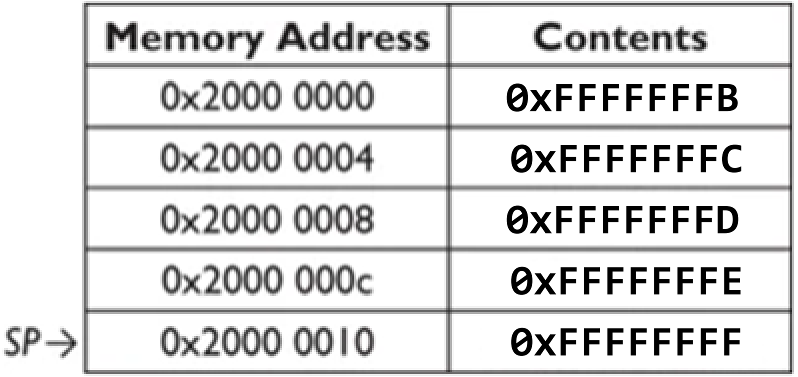


```
LDR   r0,[sp,#0] ; load register r0 from memory starting at address sp + 0
ADDS  r0,r0,r4   ; add the contents of r0 and r4, placing the results in r0
STR   r0,[sp,#0] ; save the contents of r0 to memory starting at address sp + 0
```



General purpose
Low registers

R0	0x00000004
R1	0x00000003
R2	0x00000002
R3	0x00000001
R4	0x00000000
R5	0x00000001
R6	0x00000002
R7	0x00000003



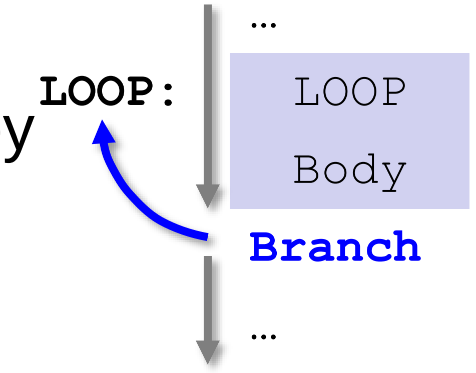
Memory Address	Contents
0x2000 0000	0xFFFFFFFB
0x2000 0004	0xFFFFFFFC
0x2000 0008	0xFFFFFFFD
0x2000 000c	0xFFFFFFFE
SP→ 0x2000 0010	0xFFFFFFFF

Outline

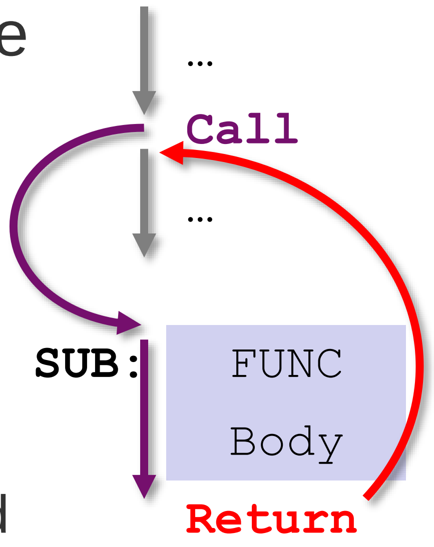
- **Processor Core**
  - A Simplified View
  - Sequential Execution
  - Architecture Profile
  - Programming Model: Load/Store
  - Registers
  - Memory
  - Instructions
    - Condition Code Flags
    - Branch & Subroutine
  - Exception/Interrupt Handling

Branch & Subroutine Call/Return

- **Branch**:
  - The instruction jumping to any instruction by loading its memory address into PC.



- **Subroutine/Function Call**
  - **Subroutine**: a block of instructions that will be executed each time when calling.
  - **Subroutine/Function Call**: when a program *branches* to and back from a subroutine.
    - `Call` branches to the subroutine.
    - `Return` branches back to the caller.
    - The special register Link Register (**LR**) can be used to save the return address.



Branch

- **Unconditional Branch**

  `B <label>`

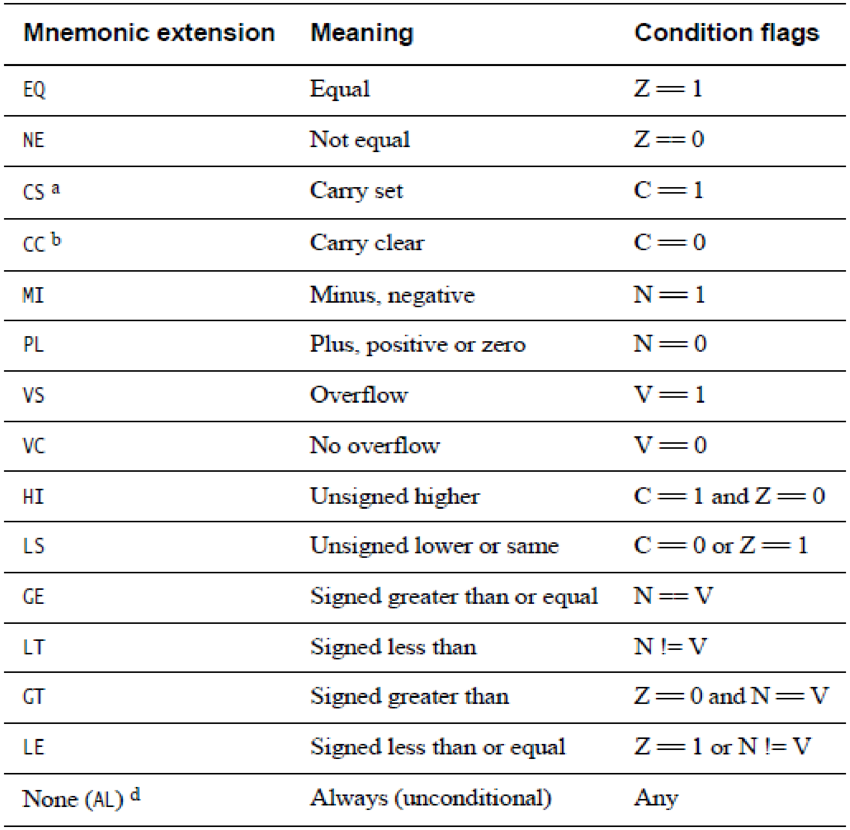
  – Target address must be within ±32 MB/2 KB of branch instruction (for Full/Thumb instruction set, respectively)

- **Conditional Branch**

  `B<cond> <label>`

  – `<cond>` specifies the condition (in the form of mnemonic extension).

  – Target address must be within of branch instruction.

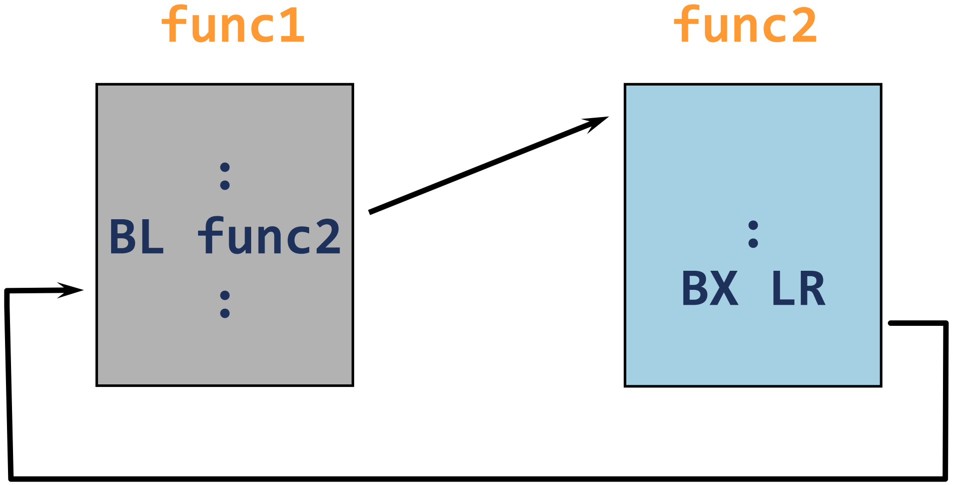


Mnemonic extension	Meaning	Condition flags
EQ	Equal	Z == 1
NE	Not equal	Z == 0
CS [a]	Carry set	C == 1
CC [b]	Carry clear	C == 0
MI	Minus, negative	N == 1
PL	Plus, positive or zero	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
HI	Unsigned higher	C == 1 and Z == 0
LS	Unsigned lower or same	C == 0 or Z == 1
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	Z == 0 and N == V
LE	Signed less than or equal	Z == 1 or N != V
None (AL) [d]	Always (unconditional)	Any

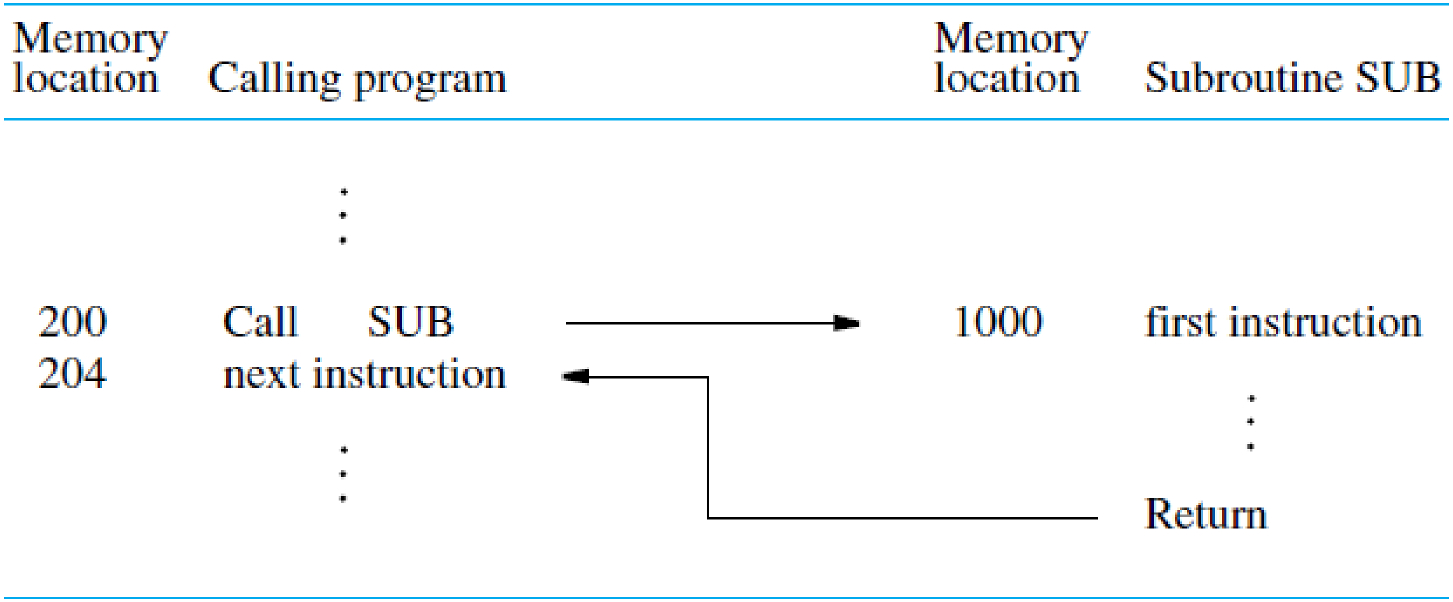
Subroutine Call & Return

- The **call** requires two steps
  - ① Store the return address
  - ② Branch to the address of the required subroutine
  - – Note: These steps are carried out in one instruction, **BL**, where the return address is stored in the Link Register (**LR**).
- The **return** is by branching to the address in **LR**.

```
void func1 (void)
{
        :
        func2();
        :
}
```

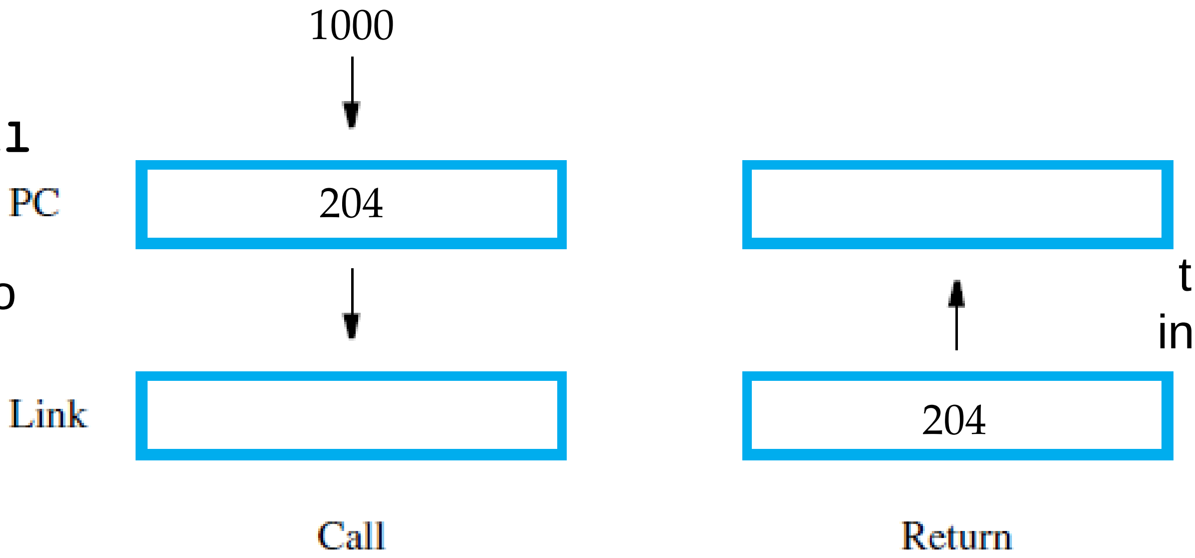


An Use Example of Link Register



② Branch to the target address specified by **`Call`**

① Keep [PC] into the link register.



Branch back to the address kept in the link register by **`Return`**

Working Example

```c
int a = 25;
int b = 45;
while(a != b )
{
  if(a < b)
  {
    a = a + 5;
  else
  {
    b = b + 5;
  }
}
```

```asm
  MOV R1, #25 ; int a = 25
  MOV R2, #45 ; int b = 45
while_loop:
  CMP R1, R2   ; compare R1 and R2
  BEQ end_loop ; if R1 == R2, jump to end_loop
  ; if (a < b)
    CMP R1, R2
    BLT incr_a ; if R1 < R2, jump to incr_a
  ; else
    ADD R2, R2, #5 ; b = b + 5
    B while_loop   ; unconditional branch
  incr_a:
    ADD R1, R1, #5 ; a = a + 5
    B while_loop   ; unconditional branch
end_loop:
```

Class Exercise 5.3

- Consider the code snippets below. Determine if it is fine to remove the “second” **`CMP R1, R2`** instruction in the assembly code snippet.

```
int a = 25;
int b = 45;
while(a != b )
{
  if(a < b)
  {
    a = a + 5;
  else
  {
    b = b + 5;
  }
}
```

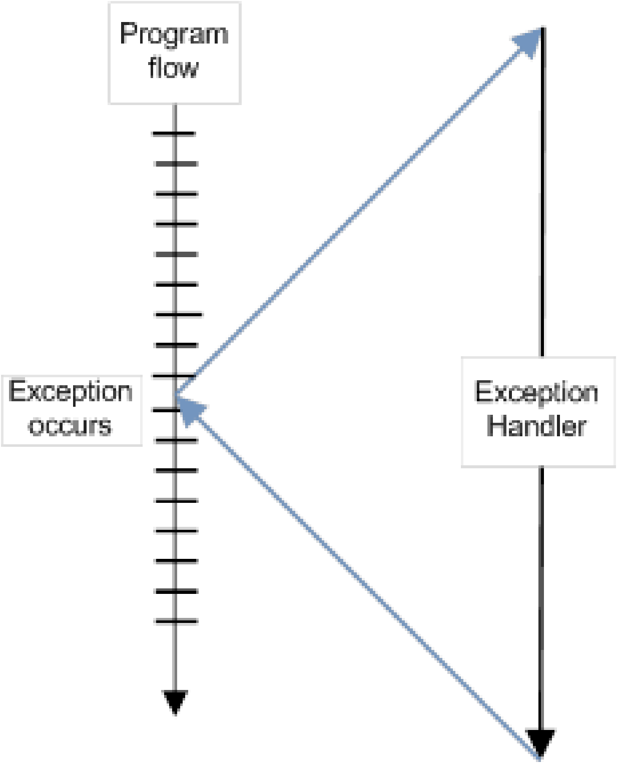
```
  MOV R1, #25 ; int a = 25
  MOV R2, #45 ; int b = 45
while_loop:
  CMP R1, R2   ; compare R1 and R2
  BEQ end_loop ; if R1 == R2, jump to end_loop
  ; if (a < b)
    CMP R1, R2
    BLT incr_a ; if R1 < R2, jump to incr_a
  ; else
    ADD R2, R2, #5 ; b = b + 5
    B while_loop   ; unconditional branch
  incr_a:
    ADD R1, R1, #5 ; a = a + 5
    B while_loop   ; unconditional branch
end_loop:
```

Outline

- **Processor Core**
  - A Simplified View
  - Sequential Execution
  - Architecture Profile
  - Programming Model: Load/Store
  - Registers
  - Memory
  - Instructions
    - Condition Code Flags
    - Branch & Subroutine
  - Exception/Interrupt Handling

Exception vs. Interrupt?

- Exceptions and interrupts are critical tools.
  - They make an embedded system responsive while supporting concurrent operation of hardware and software.
- In ARM, interrupts are *a type of* exception.
  - An exception is an event (other than branch or jump instructions) that causes the normal sequential execution to be modified.
  - An interrupt is an exception that is not caused directly by program execution.
    - Usually, hardware external to the processor core signals an interrupt, such as a switch being pressed.

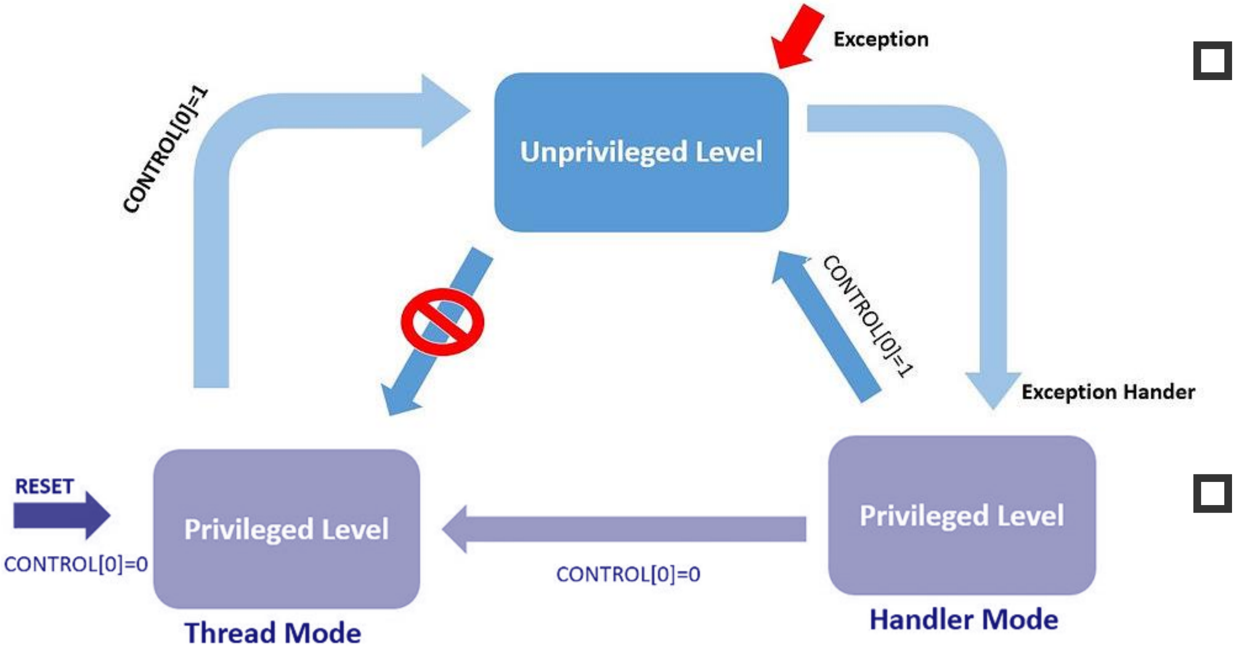


Operating Modes & Privilege Levels (1/2)

- Cortex-M4 processors can operate in two modes with two levels (specified in the **CONTROL** special register):

  ① **Thread Mode**: for executing application code.
  - The processor can run in either **Unprivileged** or **Privileged Level**.

  ② **Handler Mode**: for servicing exceptions and interrupts.
  - The processor is always executed in **Privileged Level**.

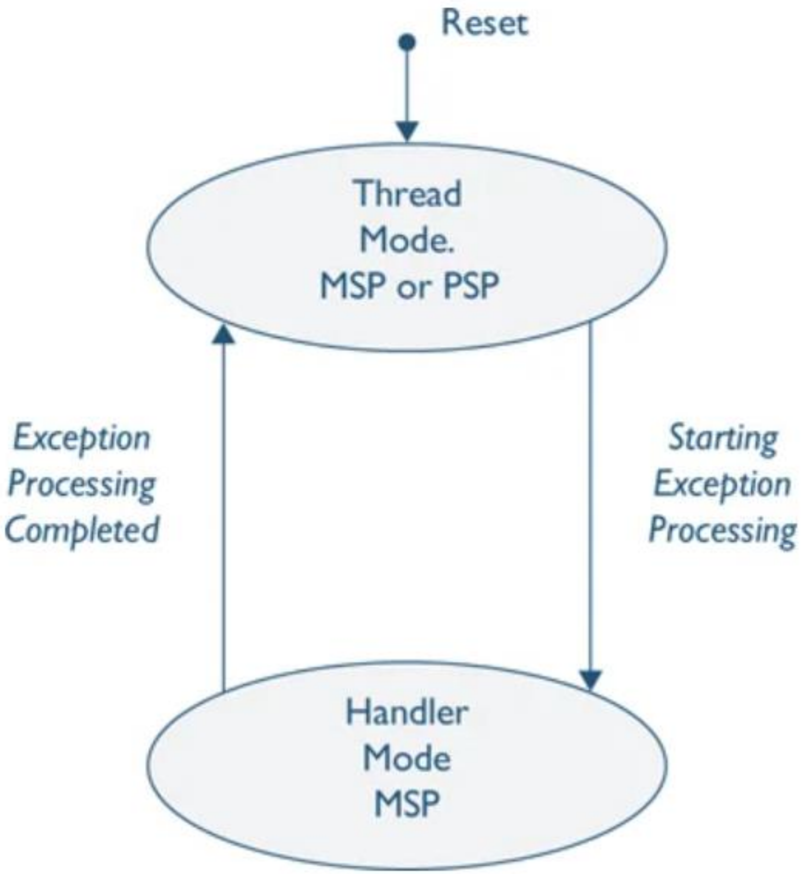


□ **Unprivileged Level** restricts access to certain system resources for security reasons
  - E.g., the processor cannot access the special registers.

□ **Privileged Level** allows access to all system resources.

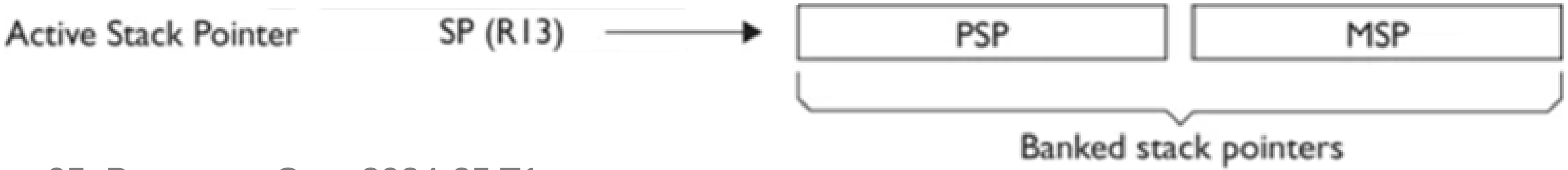
Operating Modes & Privilege Levels (2/2)

- The operating mode also determines which stack pointer (**SP**) is used.
  - **Thread**: **SP** refers to **PSP/MSP**.
  - **Handler**: **SP** refers to **MSP**.
- Why? The OS can protect the main stack and prevent applications from accessing/corrupting it.



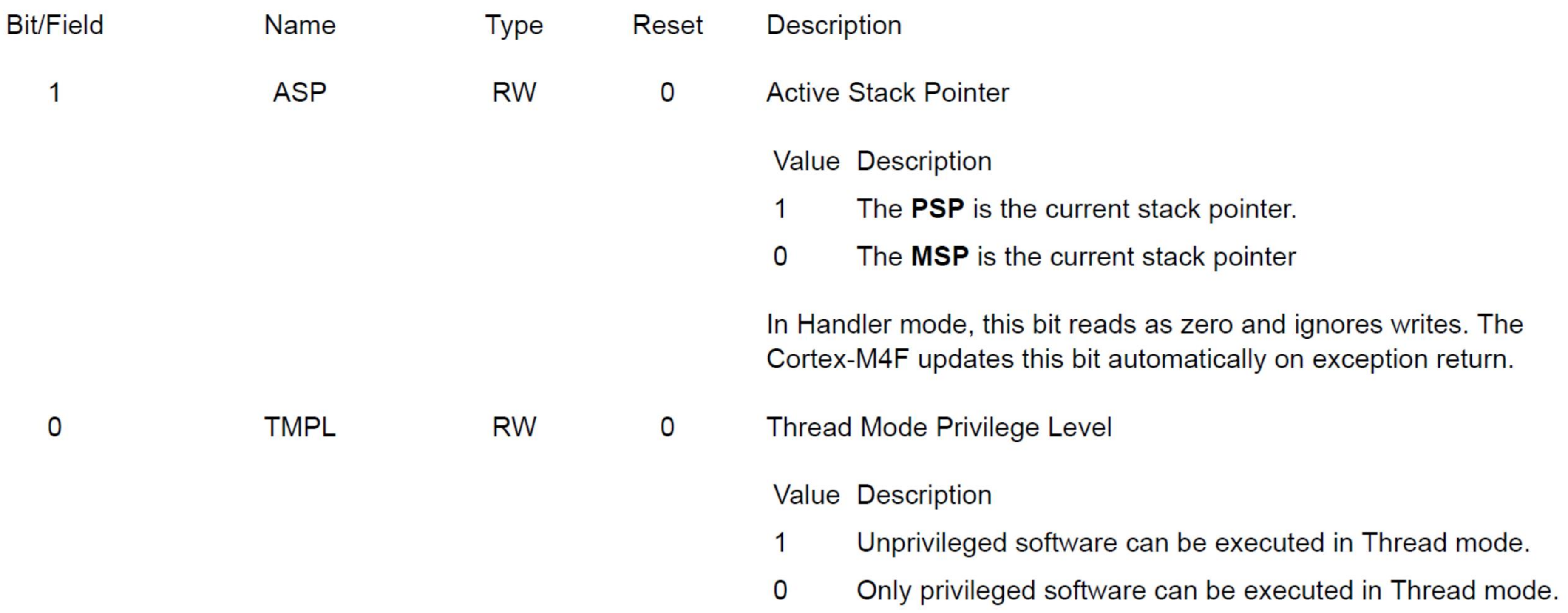
Process SP (CONTROL[1]=1)

Main SP (CONTROL[1]=0)



Control Register (CONTROL)

- The **CONTROL** register specifies the stack used and the privilege level for software execution.
  - This register is only accessible in **Privileged Level**.



Bit/Field	Name	Type	Reset	Description
1	ASP	RW	0	Active Stack Pointer Value Description 1 The **PSP** is the current stack pointer. 0 The **MSP** is the current stack pointer In Handler mode, this bit reads as zero and ignores writes. The Cortex-M4F updates this bit automatically on exception return.
0	TMPL	RW	0	Thread Mode Privilege Level Value Description 1 Unprivileged software can be executed in Thread mode. 0 Only privileged software can be executed in Thread mode.

Note: It also indicates whether the FPU state is active.

Exception Handling Procedure

- The processor services an exception (or an interrupt) with the following steps:

Entering a Handler

① **Pause** the program (i.e., complete executing the current instruction; otherwise, it may take many cycles);

② **Save** context information (such as registers and which instruction to execute next in the current program);

③ **Determine** which handler to run (each type of exception or interrupt can have a separate handler);

Executing a Handler

④ **Execute** the code for the handler;

Exiting a Handler

⑤ **Restore** the context information that was saved earlier;

⑥ **Resume** executing the current program where it left off.

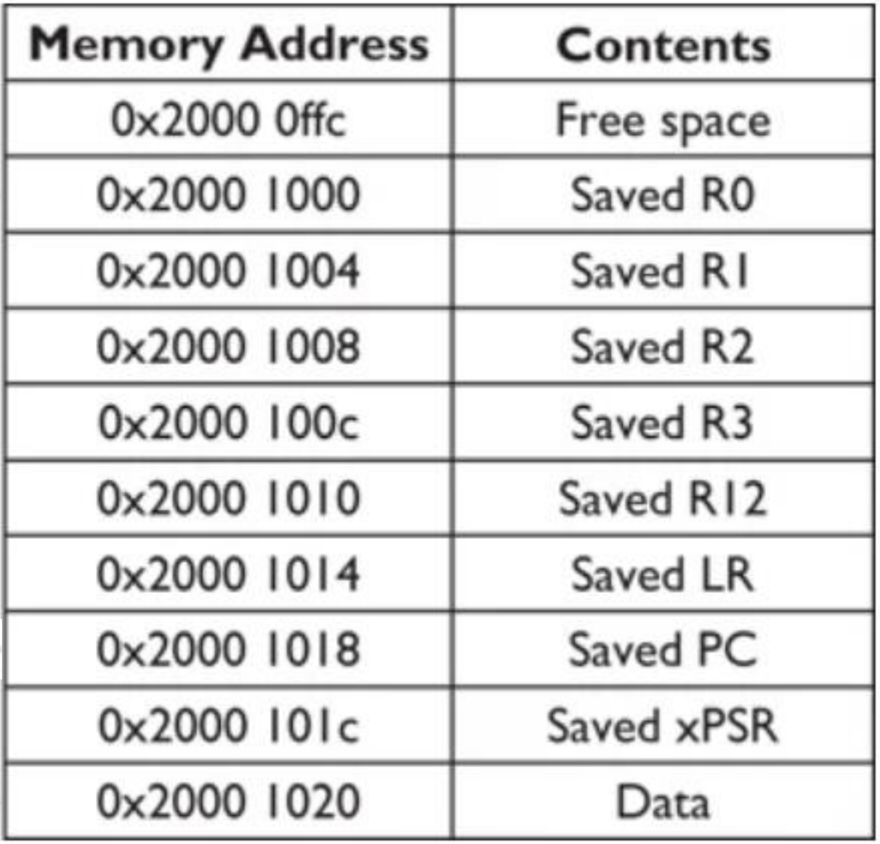
② & ⑤: How to “save” & “store” context?

- The processor stack can help:
  - We can keep critical processor registers that hold execution context for the suspended program in the processor stack.

SP upon entering exception handler
SP before exiting exception handler

PC holds the address of the “next” instruction to the suspended program

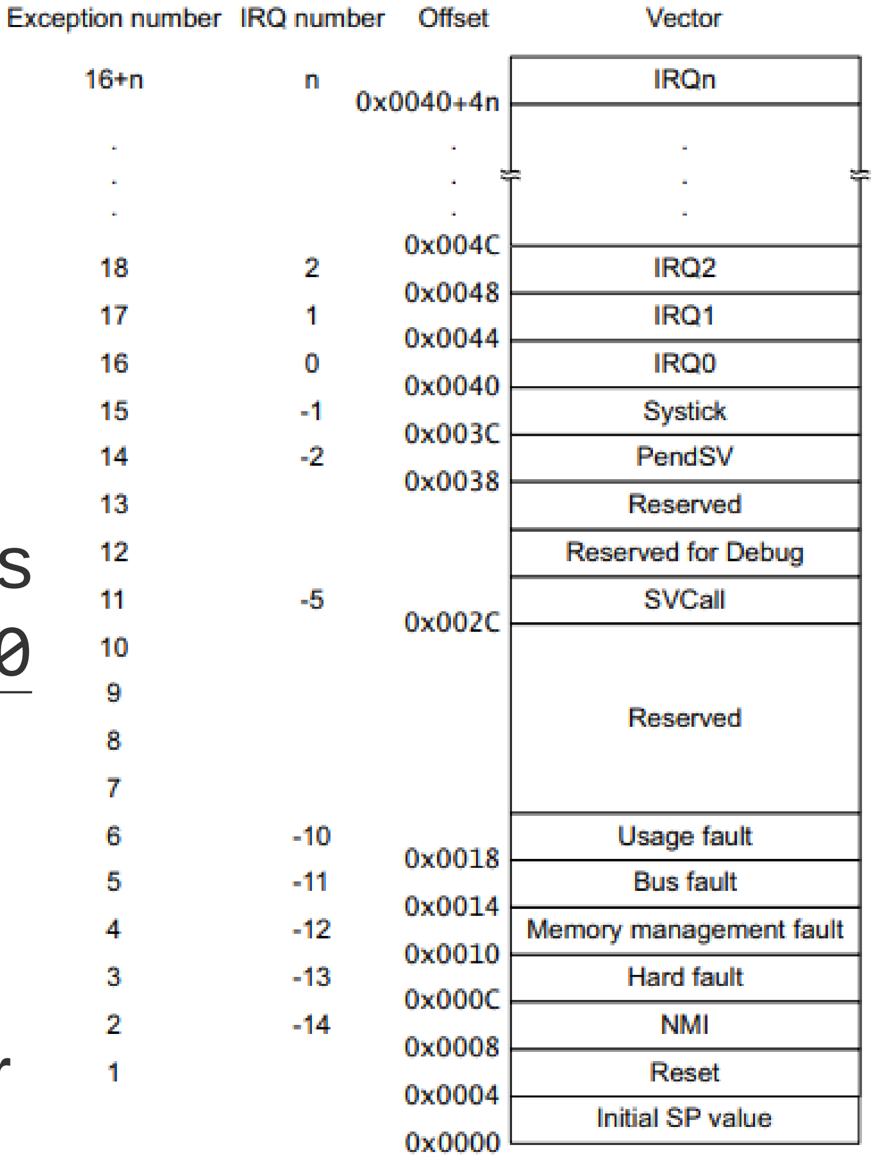
SP before entering exception handler
SP after exiting exception handler



Memory Address	Contents
0x2000 0ffc	Free space
0x2000 1000	Saved R0
0x2000 1004	Saved R1
0x2000 1008	Saved R2
0x2000 100c	Saved R3
0x2000 1010	Saved R12
0x2000 1014	Saved LR
0x2000 1018	Saved PC
0x2000 101c	Saved xPSR
0x2000 1020	Data

③: How to “determine” the handler? (1/2)

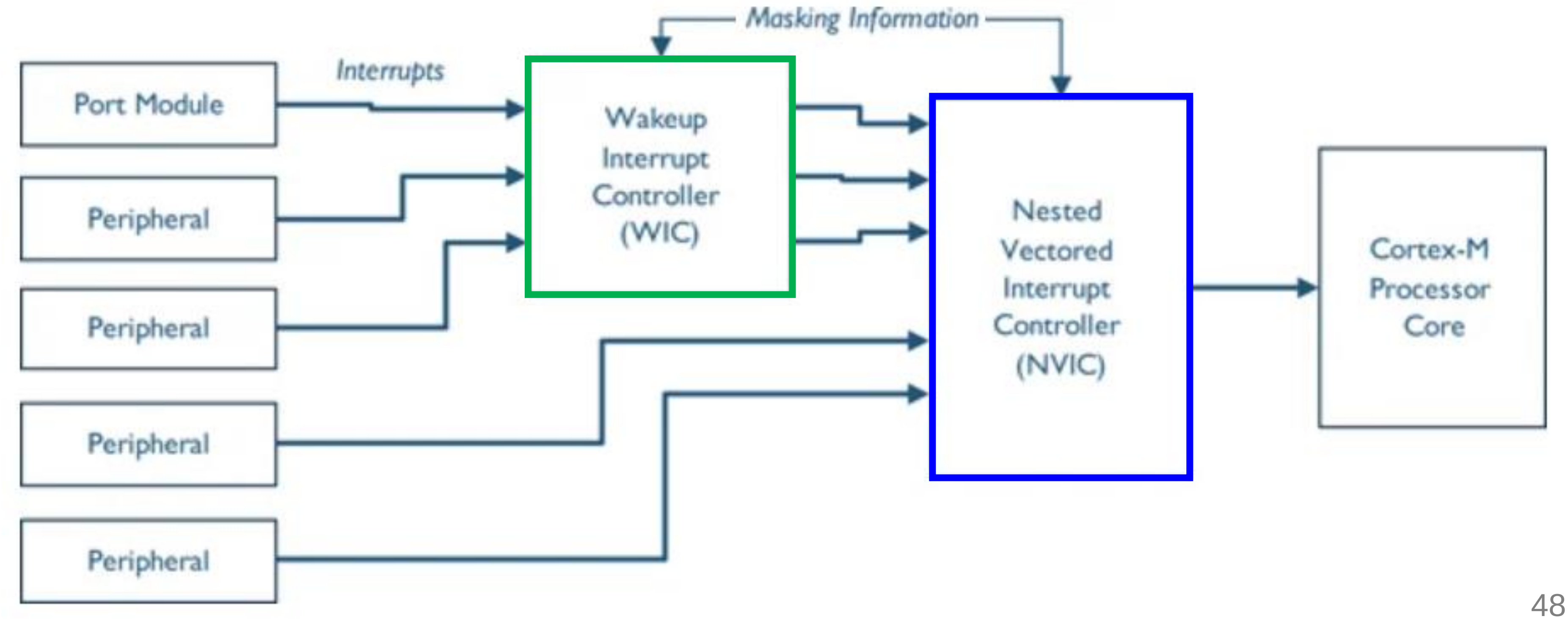
- Each possible exception (and interrupt) has a vector to specify its handler.
  - The starting address of handlers are stored in the vector table.
- In general, the vector table is fixed at address `0x00000000` on system reset.
  - A privileged software can relocate the vector table start address by modifying the **Vector Table Offset Register** (**VTOR**) register.



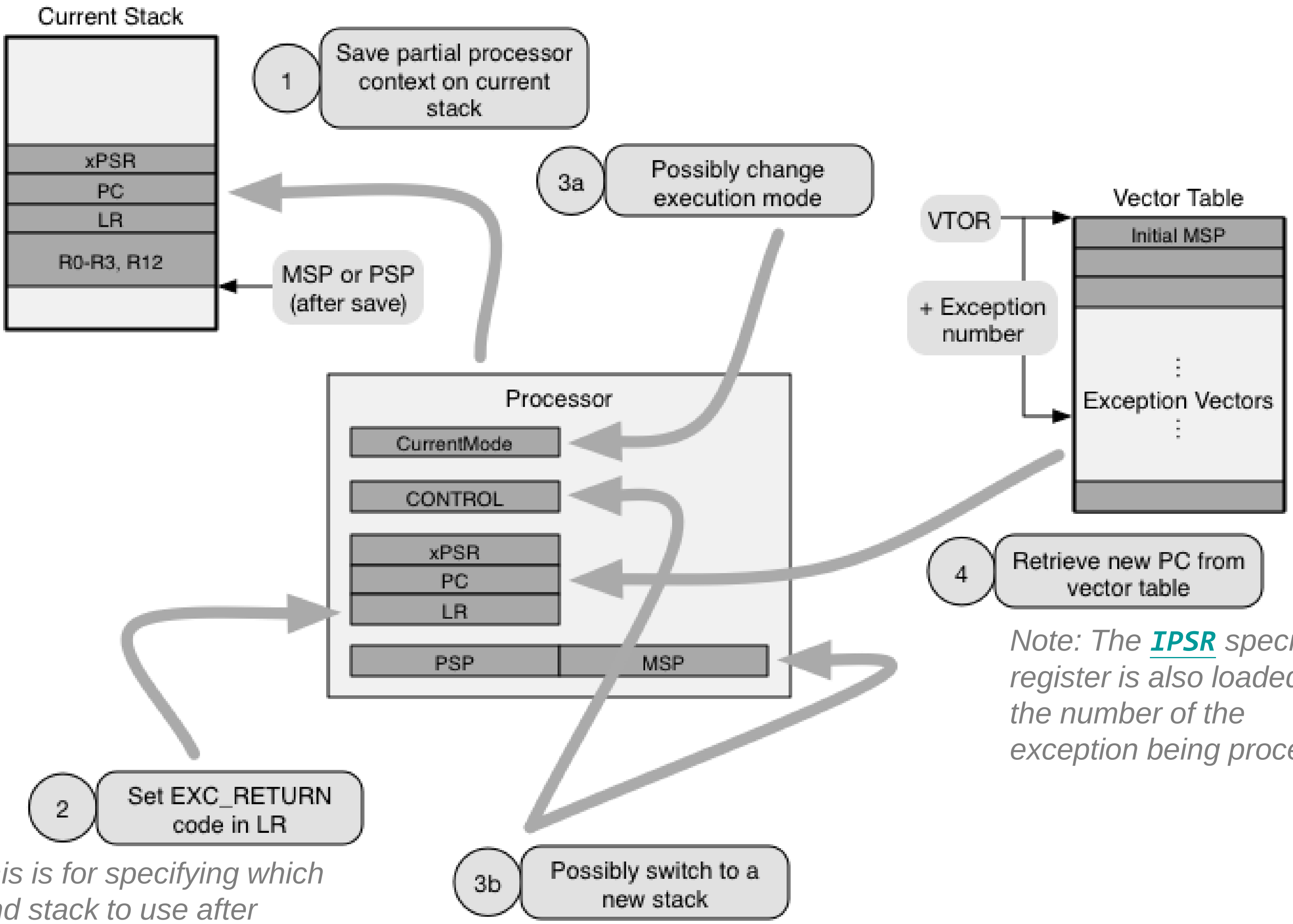
Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.		.	.
.		.	.
.		.	.
		0x004C	
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

③: How to "determine" the handler? (2/2)

- The hardware involved in recognizing interrupts:
  - **Peripherals** can generate interrupt requests.
  - The **NVIC** selects the interrupt requests with the highest priority and directs the **core** to execute the interrupt handler.
    - The (optional) **WIC** duplicates the interrupt masking, enabling the NVIC to be turned off to reduce power use.



Entering a Handler: More Details



Current Stack
xPSR
PC
LR
R0-R3, R12
1
Save partial processor context on current stack
MSP or PSP (after save)
3a
Possibly change execution mode
VTOR
+ Exception number
Vector Table
Initial MSP
Exception Vectors
Processor
CurrentMode
CONTROL
xPSR
PC
LR
PSP
MSP
4
Retrieve new PC from vector table
2
Set EXC_RETURN code in LR
3b
Possibly switch to a new stack

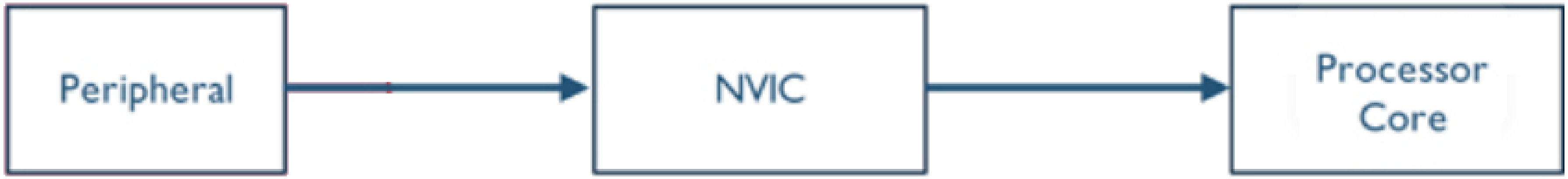
*Note: The **IPSR** special register is also loaded with the number of the exception being processed.*

Note: This is for specifying which mode and stack to use after completing the exception handling.

https://www.embeddedrelated.com/showarticle/912.php

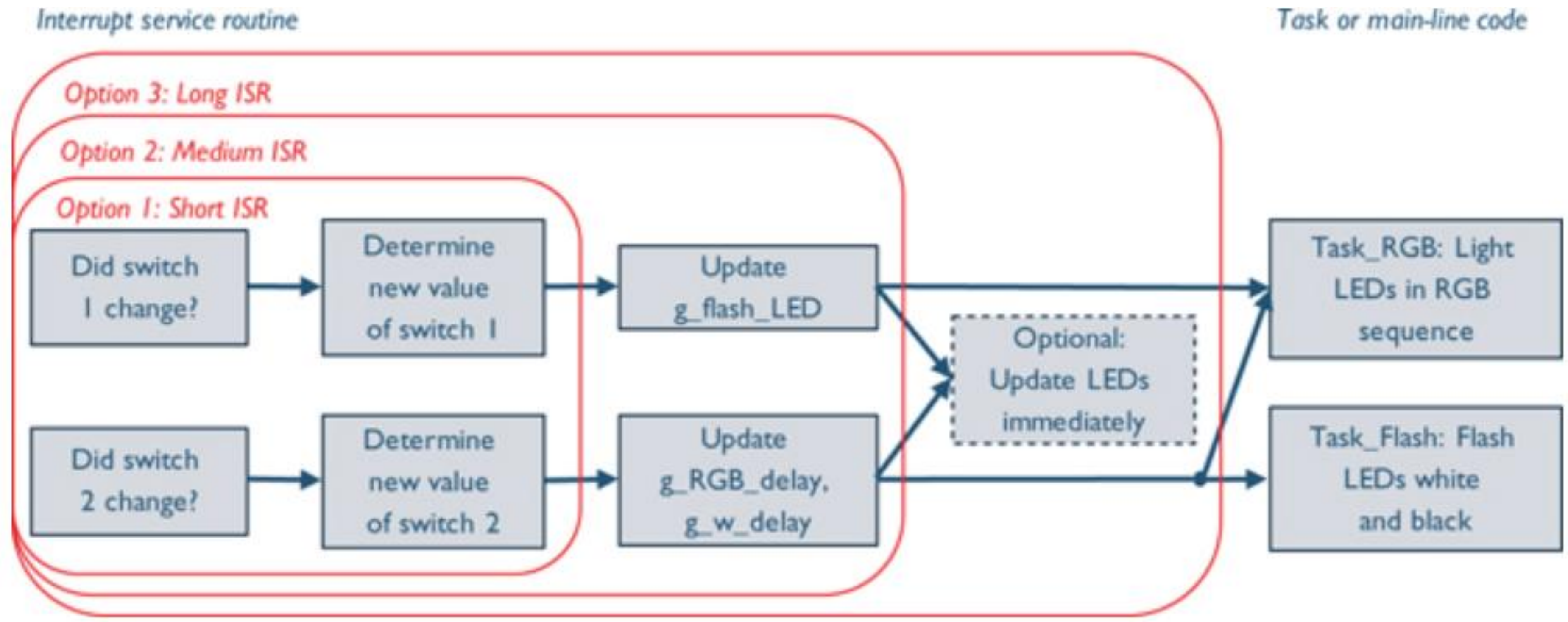
Interrupt Configuration

- Both the **peripheral** and **NVIC** need to be configured to use interrupts.
  - In Cortex-M4 processors, the NVIC supports up to 240 interrupt sources, each with up to 256 levels of priority.
  - Each interrupt source can be enabled, disabled, and prioritized (using the "reversed" priority numbering scheme).
    - That is, priority zero corresponds to the highest urgency interrupt.
  - The statuses of pending (i.e., requested but not yet serviced) interrupts can be read and modified.
- The **processor core** can also be configured to accept or ignore certain types of exceptions

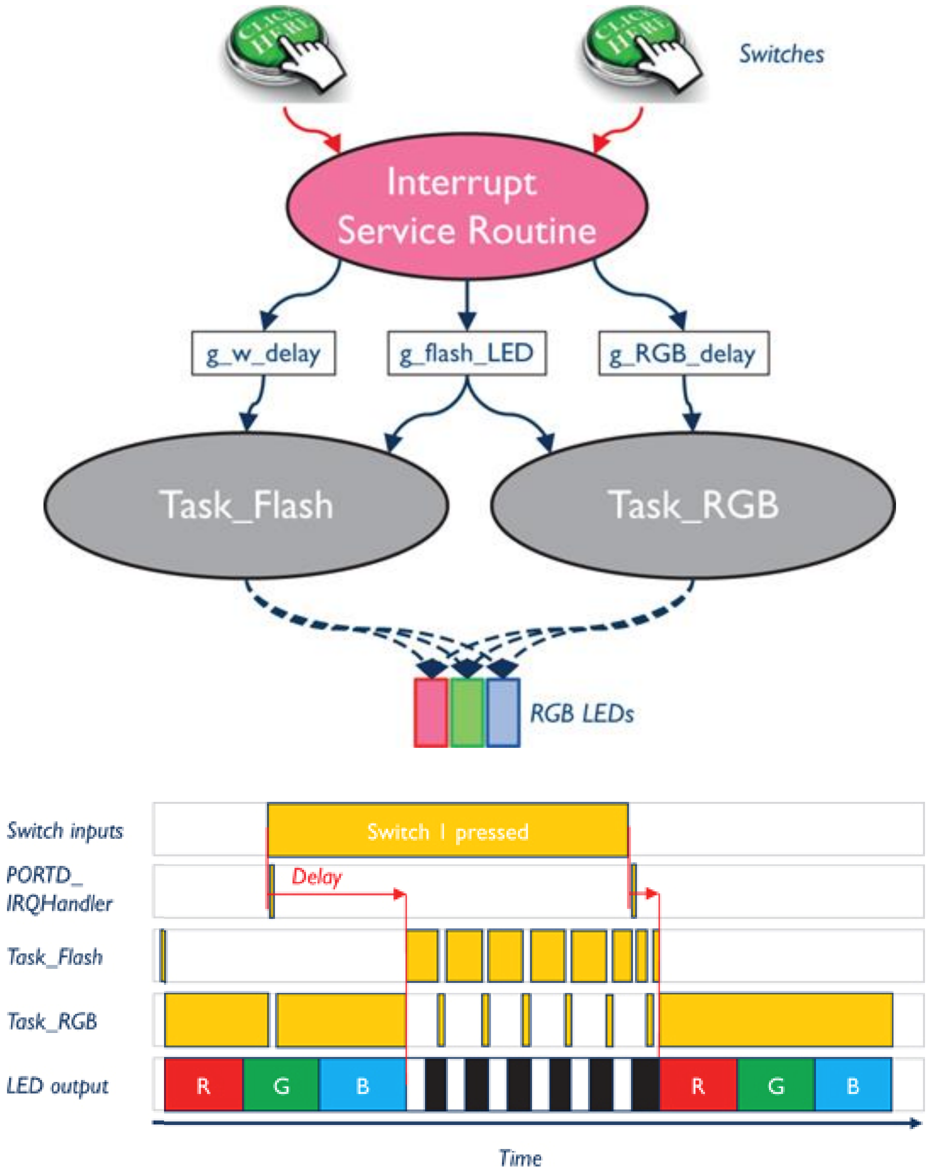


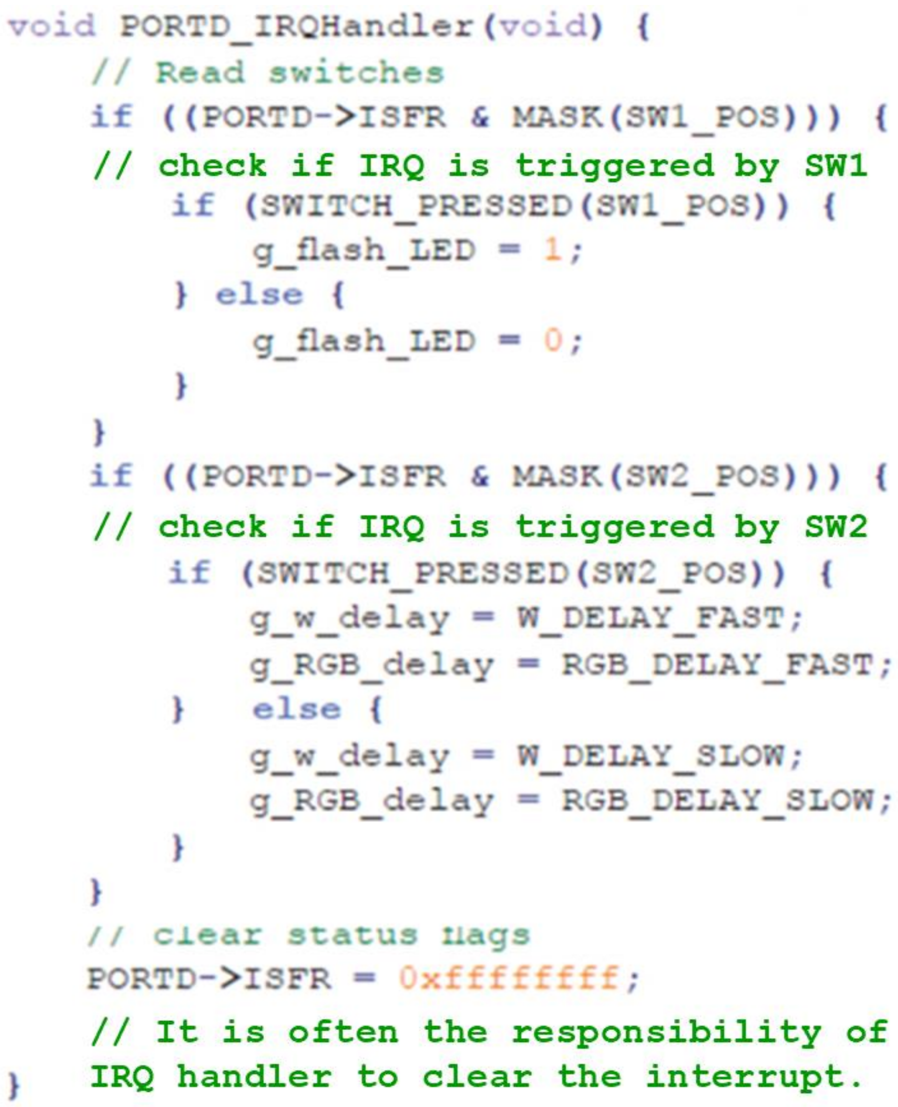
Software for Interrupts (1/3)

- The ISR should perform only quick, urgent work related to the interrupt; other work should be deferred to the main-line code when feasible.
  - This keeps each ISR quick and doesn't delay other ISRs.
- Re-examine the "Switch-triggered Program":


Interrupt service routine
Task or main-line code
Option 3: Long ISR
Option 2: Medium ISR
Option 1: Short ISR
Did switch 1 change?
Determine new value of switch 1
Update g_flash_LED
Did switch 2 change?
Determine new value of switch 2
Update g_RGB_delay, g_w_delay
Optional: Update LEDs immediately
Task_RGB: Light LEDs in RGB sequence
Task_Flash: Flash LEDs white and black

Software for Interrupts (2/3)





```
void PORTD_IRQHandler(void) {
    // Read switches
    if ((PORTD->ISFR & MASK(SW1_POS))) {
    // check if IRQ is triggered by SW1
        if (SWITCH_PRESSED(SW1_POS)) {
            g_flash_LED = 1;
        } else {
            g_flash_LED = 0;
        }
    }
    if ((PORTD->ISFR & MASK(SW2_POS))) {
    // check if IRQ is triggered by SW2
        if (SWITCH_PRESSED(SW2_POS)) {
            g_w_delay = W_DELAY_FAST;
            g_RGB_delay = RGB_DELAY_FAST;
        }   else {
            g_w_delay = W_DELAY_SLOW;
            g_RGB_delay = RGB_DELAY_SLOW;
        }
    }
    // clear status flags
    PORTD->ISFR = 0xffffffff;
    // It is often the responsibility of
    IRQ handler to clear the interrupt.
}
```

Software for Interrupts (3/3)

- **Option 1: Short ISR** (simply signals which switch changed)
  - A task in the main-line program needs to update the delay and flash mode variables based on this switch information.
    - This could be done by `Task_RGB` and `Task_Flash`, which would reduce the code's modularity; or the variables could be updated by a new task, which would add the overhead of creating and running another task.
- **Option 2: Medium ISR** (directly updates the delay and flash mode variables)
  - This is a quick, low overhead approach. (This is also our choice!)
- **Option 3: Long ISR** (directly updates the delay and flash mode variables, and immediately updates the LED).
  - This approach is much more responsive than the first two, but there is a problem upon resuming the previously executing task.
    - That task may light the LEDs with the wrong colors or the wrong delays.
    - We need a non-trivial way to disable or restart that task.
  - This explains why we don't opt for this option; instead, we further exploit **timer peripherals** to improve the responsiveness.

Summary

- **Processor Core**
  - A Simplified View
  - Sequential Execution
  - Architecture Profile
  - Programming Model: Load/Store
  - Registers
  - Memory
  - Instructions
    - Condition Code Flags
    - Branch & Subroutine
  - Exception/Interrupt Handling