



Queen Mary
University of London

Science and Engineering

School of Electronic Engineering and Computer Science
QMUL-BUPT Joint Programme

EBU6475 Microprocessor System Design

EBU5476 Microprocessors for Embedded Computing

The ARM Cortex-M4 Processor Architecture
Part 2

arm

Last updated: March 5, 2022

University Program Education Kits

Outline

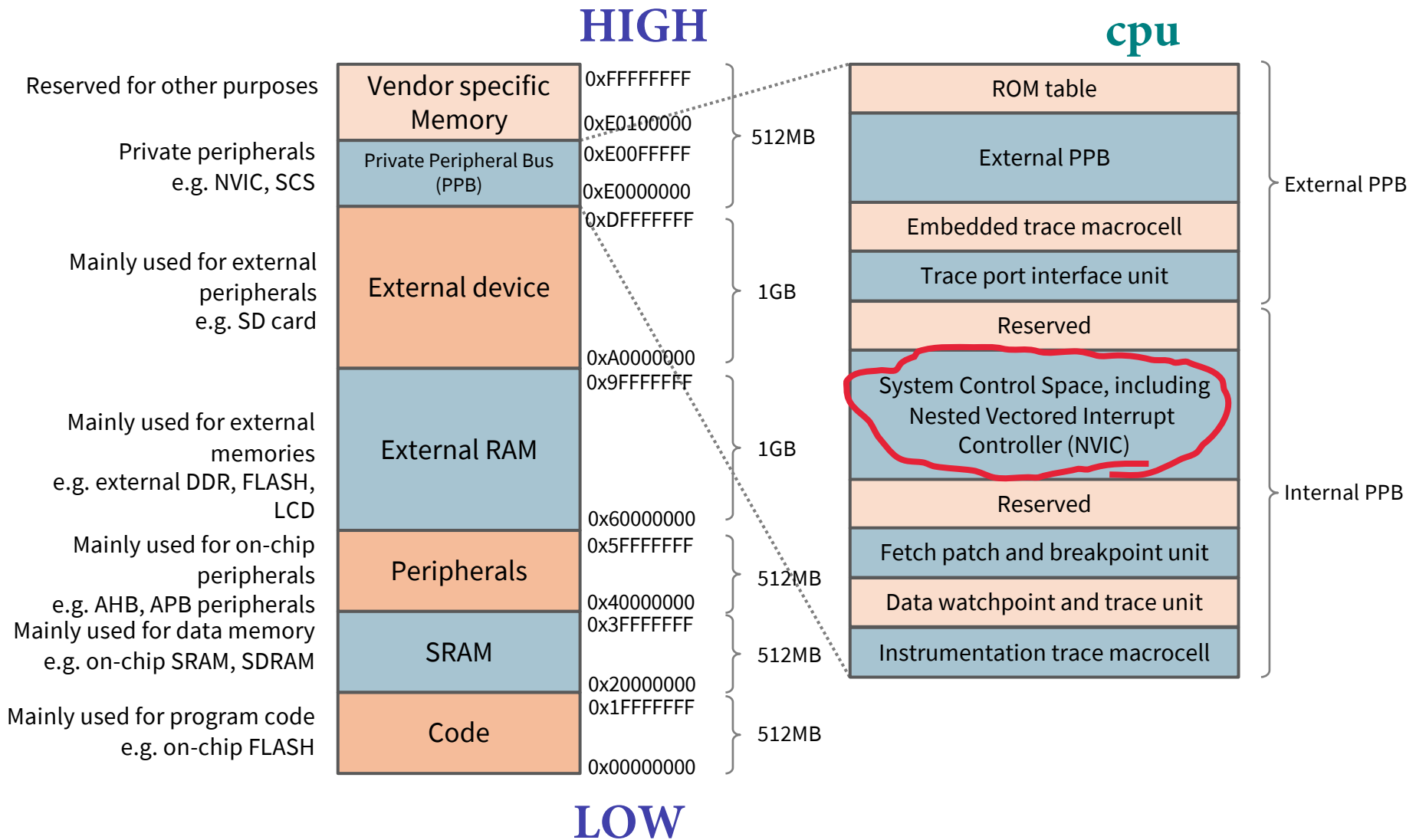
- Cortex-M4 Memory Map
 - Cortex-M4 Memory Map
 - Bit-band Operations
 - Cortex-M4 Program Image and Endianness
- ARM Cortex-M4 Processor Instruction Set
 - ARM and Thumb Instruction Set
 - Cortex-M4 Instruction Set
 - Moving data: MOV, MRS, LDR, STR
 - Basic arithmetic instructions: ADD, SUB, etc.
 - Assembly programming: example

ARM Cortex-M4 Processor Memory Map

Cortex-M4 Memory Map

- The Cortex-M4 processor has 4 GB of memory address space
 - Support for bit-band operation (detailed later)
- The 4GB memory space is architecturally defined as a number of regions
 - Each region is given for recommended usage
 - Easy for software programmer to port between different devices
- Nevertheless, despite of the default memory map, the actual usage of the memory map can also be flexibly defined by the user, except some fixed memory addresses, such as internal private peripheral bus

Cortex-M4 Memory Map



Cortex-M4 Memory Map

- Code Region

- Primarily used to store program code
- Can also be used for data memory
- On-chip memory, such as on-chip FLASH

CSP part

Coke, Sprite, Pepsi (可乐, 雪碧, 百事)
越来越往上

- SRAM Region

- Primarily used to store data, such as heaps and stacks
- Can also be used for program code
- On-chip memory; despite its name “SRAM”, the actual device could be SRAM, SDRAM or other types

variables

C programs

- Peripheral Region

- Primarily used for peripherals, such as Advanced High-performance Bus (AHB) or Advanced Peripheral Bus (APB) peripherals
- On-chip peripherals

volatile memory 断电就没

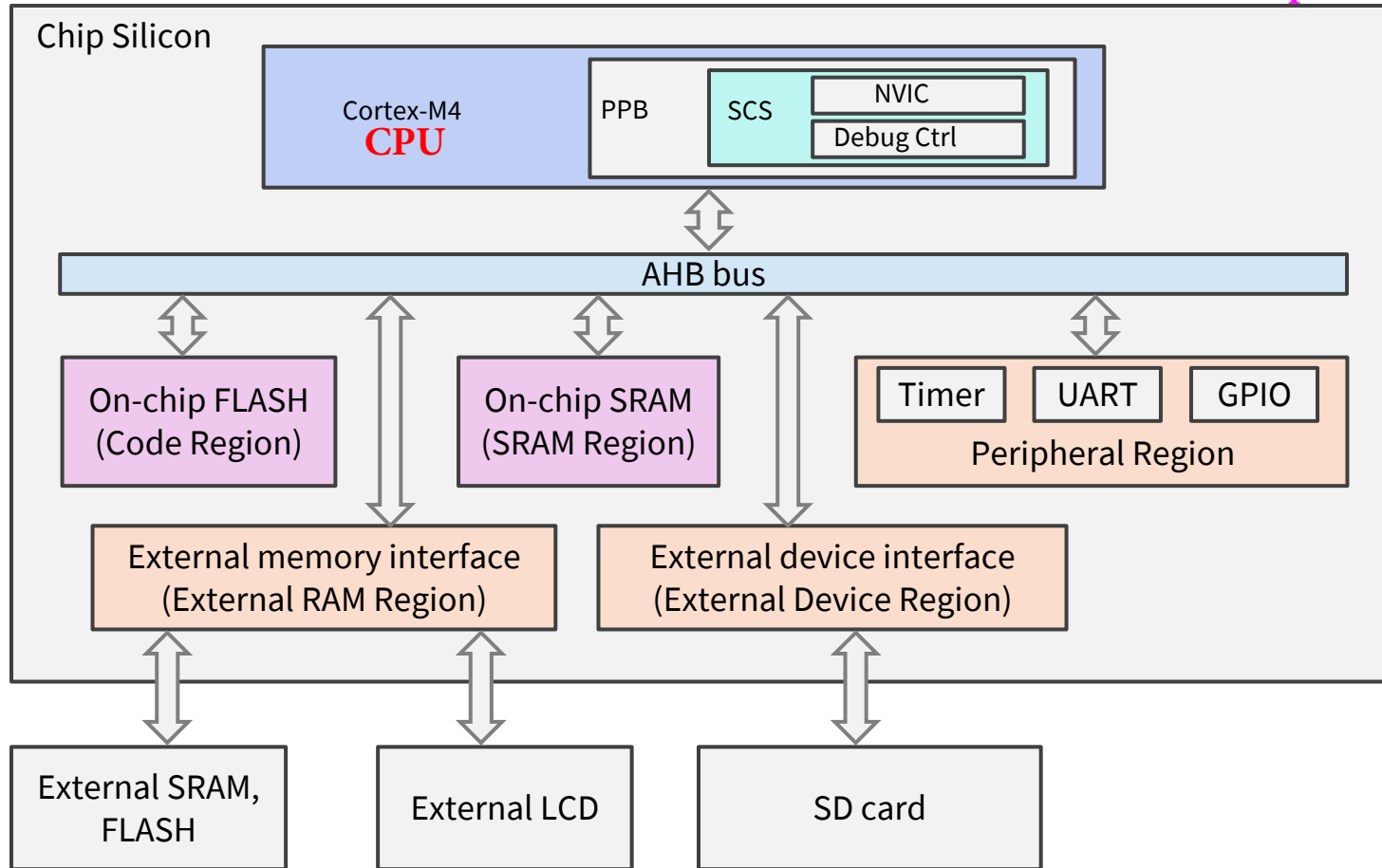
i/o devices, e.g. printer

Cortex-M4 Memory Map

- External RAM Region
 - Primarily used to store large data blocks, or memory caches
 - Off-chip memory, slower than on-chip SRAM region
 - External Device Region
 - Primarily used to map to external devices
 - Off-chip devices, such as SD card
 - Internal Private Peripheral Bus (PPB)
 - Used inside the processor core for internal control
 - Within PPB, a special range of memory is defined as System Control Space (SCS)
 - The Nested Vectored Interrupt Controller (NVIC) is part of SCS
- 都是external, off-chip
- 修改这部分, 可以控制CPU内部

Cortex-M4 Memory Map Example

part2-1 18:00
example: MP3 player



Bit-band Operations

- Bit-band operation allows a single load/store operation to access a single bit in the memory, for example, to change a single bit of one 32-bit data:
 - Normal operation without bit-band (read-modify-write)
 - load • Read the value of 32-bit data
 - modify • Modify a single bit of the 32-bit value (keep other bits unchanged)
 - store • Write the value back to the address
 - Bit-band operation
Directly write a single bit (0 or 1) to the “bit-band alias address” of the data
- Bit-band alias address
 - Each bit-band alias address is mapped to a real data address
 - When writing to the bit-band alias address, only a single bit of the data will be changed

Bit-band Operation Example

第4位

- For example, in order to set bit[3] in word data in address 0x20000000:

;Read-Modify-Write Operation

```
LDR    R1, =0x20000000    ;Setup address
LDR    R0, [R1]            ;Read
ORR.W  R0, #0x8            ;Modify bit
STR    R0, [R1]            ;Write back
```

掩码

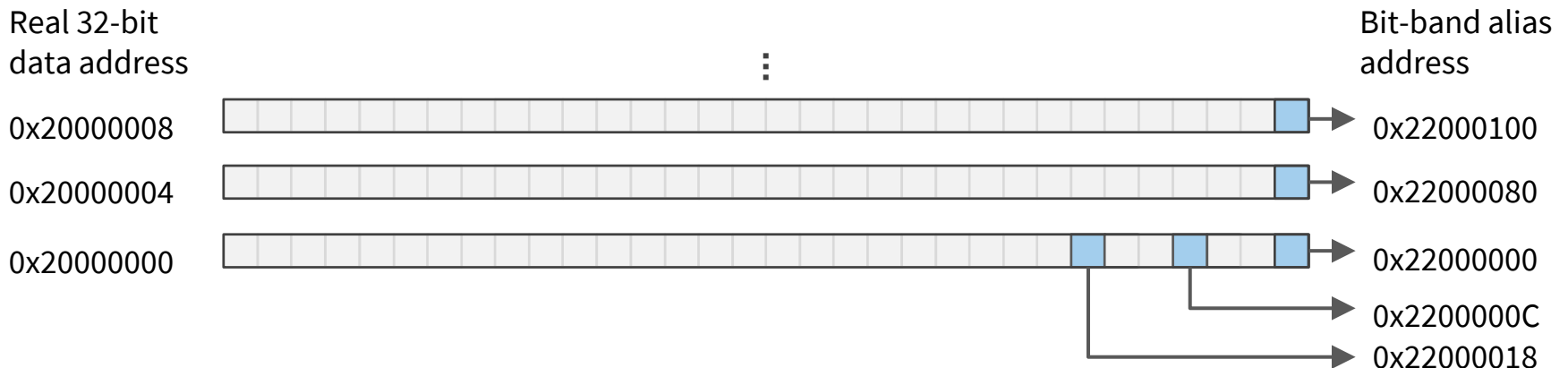
;Bit-band Operation

```
LDR    R1, =0x2200000C    ;Setup address
MOV     R0, #1             ;Load data
STR     R0, [R1]           ;Write
```

- Read-Modify-Write operation
 - Read the real data address (0x20000000)
 - Modify the desired bit (retain other bits unchanged)
 - Write the modified data back
- Bit-band operation
 - Directly set the bit by writing '1' to address 0x2200000C, which is the alias address of the fourth bit of the 32-bit data at 0x20000000
 - In effect, this single instruction is mapped to 2 bus transfers: read data from 0x20000000 to the buffer, and then write to 0x20000000 from the buffer with bit [3] set

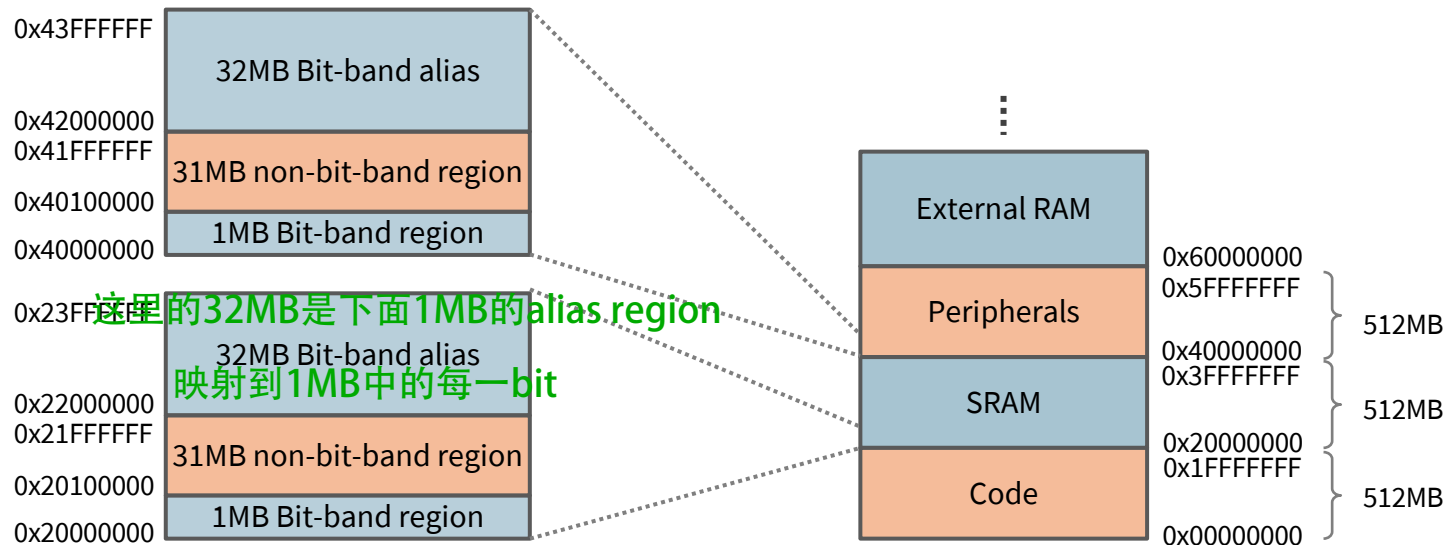
Bit-band Alias Address

- Each bit of the 32-bit data is one-to-one mapped to the bit-band alias address
 - For example, the fourth bit (bit [3]) of the data at 0x20000000 is mapped to the bit-band alias address at 0x2200000C
 - Hence, to set bit [3] of the data at 0x20000000, we only need to write '1' to address 0x2200000C
 - In Cortex-M4, there are two pre-defined bit-band alias regions: one for SRAM region, and one for peripherals region



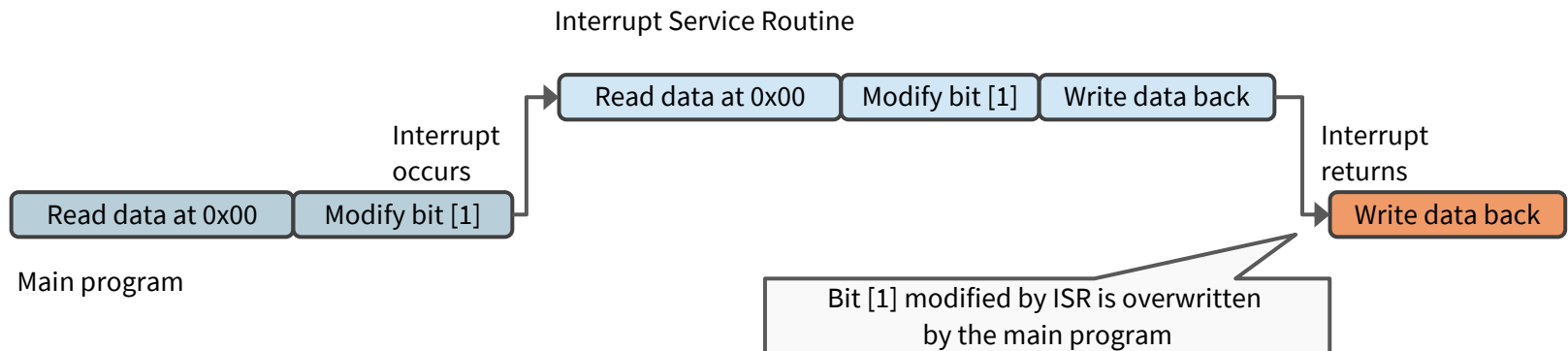
Bit-band Alias Address

- SRAM region
 - 32MB memory space (0x22000000 – 0x23FFFFFF) is used as the bit-band alias region for 1MB data (0x20000000 – 0x200FFFFFF)
- Peripherals region
 - 32MB memory space (0x42000000 – 0x43FFFFFF) is used as the bit-band alias region for 1MB data (0x40000000 – 0x400FFFFFF)



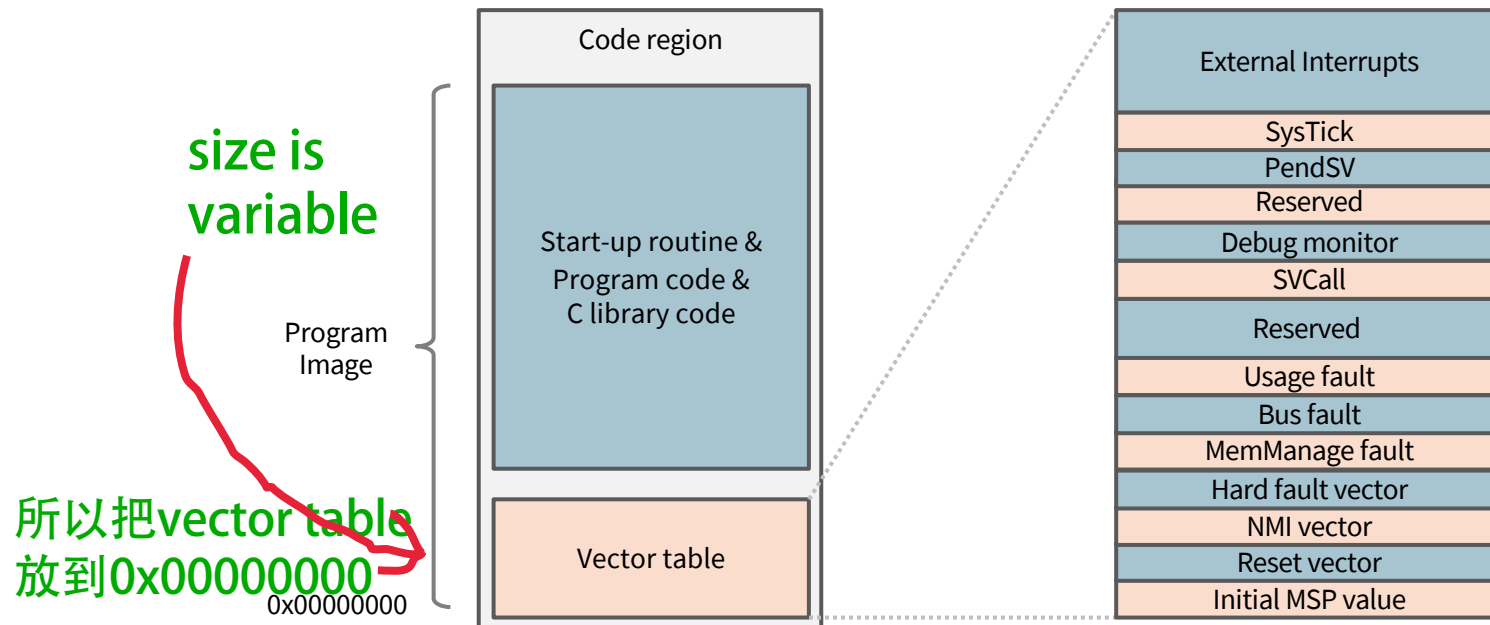
Benefits of Bit-Band Operations

- Faster bit operations **supported by hardware**
- Fewer instructions **smaller code**
- Atomic operation, avoid hazards
 - For example, if an interrupt is triggered and served during the Read-Modify-Write operations, and the interrupt service routine modifies the same data, a data conflict will occur



Cortex-M4 Program Image

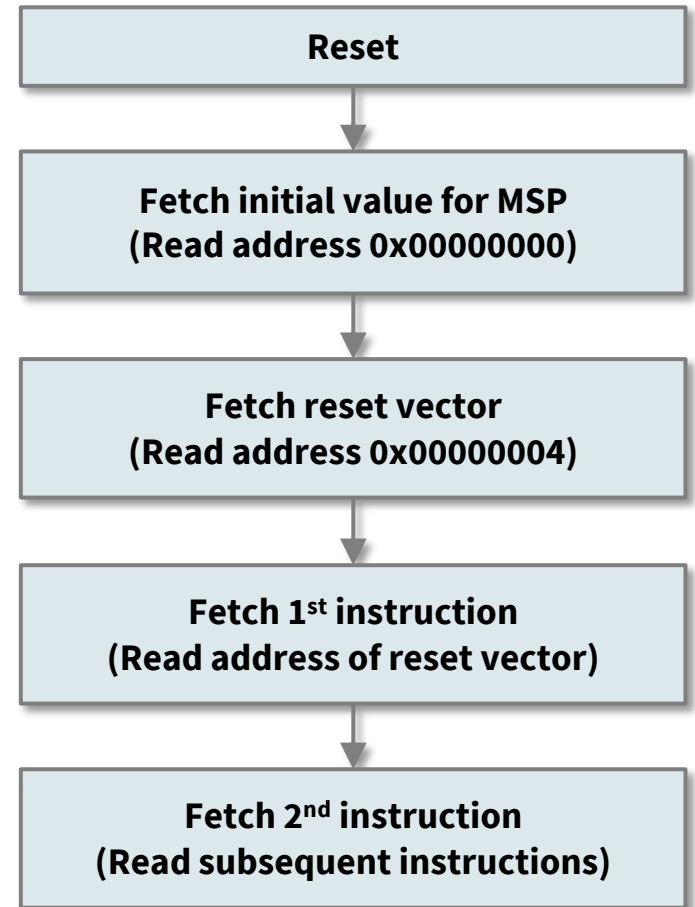
- The program image in Cortex-M4 contains
 - Vector table – includes the starting addresses of exceptions (vectors) and the value of the main stack point (MSP);
 - C start-up routine;
 - Program code – application code and data;
 - C library code – program codes for C library functions.



Cortex-M4 Program Image

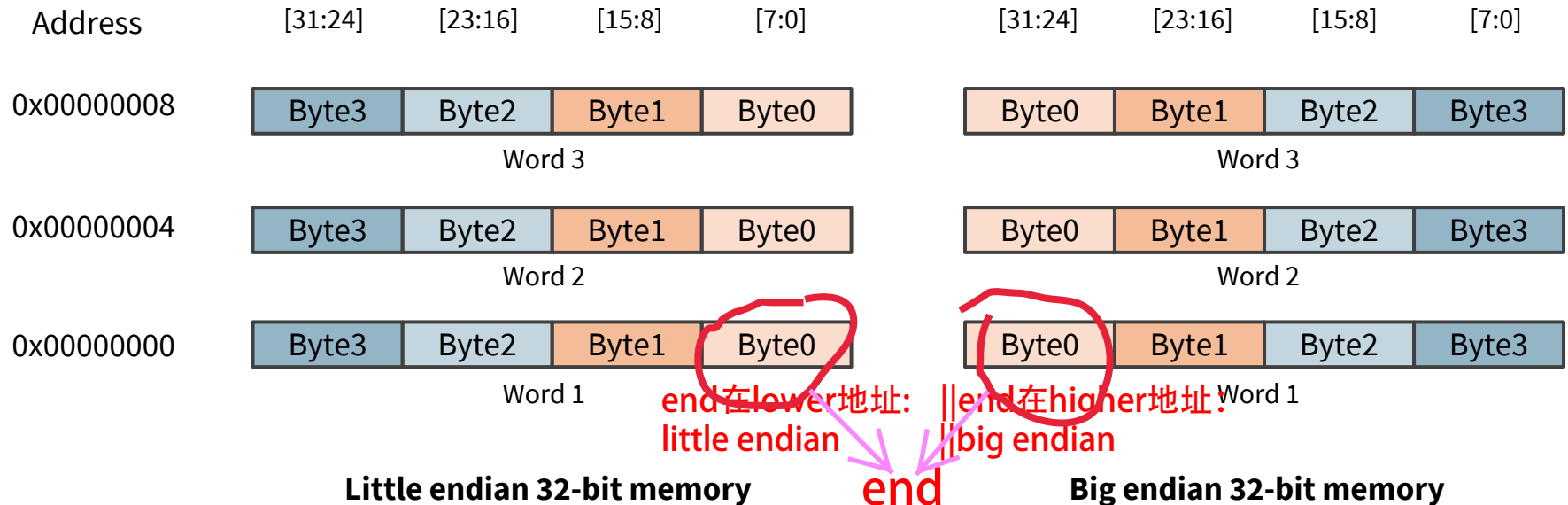
- After Reset, the processor:
 - First reads the initial MSP value;
 - Then reads the reset vector;
 - Branches to the start of the programme execution address (reset handler);
 - Subsequently executes program instructions

main stack pointer (MSP)



Cortex-M4 Endianness

- Endian refers to the order of bytes stored in memory
 - Big endian: lowest byte of a word-size data is stored in bit 0 to bit 7
 - Big endian: lowest byte of a word-size data is stored in bit 24 to bit 31
- Cortex-M4 supports both little endian and big endian
- However, endianness only exists in the hardware level



ARM Cortex-M4 Processor Instruction Set

Reference:

Chapter 5.1-5.6, Definitive Guide to ARM Cortex-M3 Processors

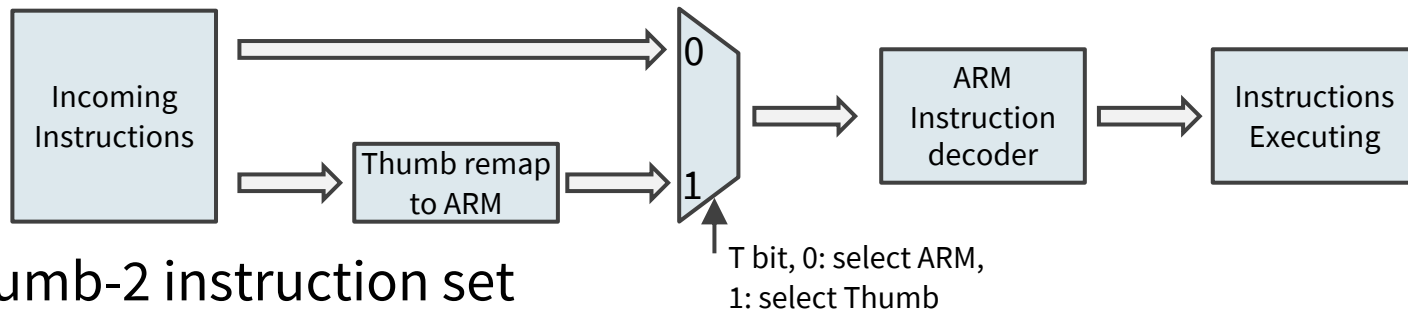
ARM and Thumb® Instruction Set

Acorn RISC Machine: ARM, Reduced Instruction Set Computer: RISC

- Early ARM instruction set **1983**
 - 32-bit instruction set, called the ARM instructions
 - Powerful and good performance
 - Larger program memory compared to 8-bit and 16-bit processors **programs became larger, because every instruction is 32 bits.**
 - Larger power consumption
- Thumb-1 instruction set
 - 16-bit instruction set, first used in ARM7TDMI processor in 1995
 - Provides a subset of the ARM instructions, giving better code density compared to 32-bit RISC architecture
 - Code size is reduced by ~30%, but performance is also reduced by ~20% **性价比不行**

ARM and Thumb® Instruction Set

- Mix of ARM and Thumb-1 Instruction sets
 - Benefit from both 32-bit ARM (high performance) and 16-bit Thumb-1 (high code density)
 - A multiplexer is used to switch between two states: ARM state (32-bit) and Thumb state (16-bit), which requires a switching overhead



- Thumb-2 instruction set
 - Consists of both 32-bit Thumb instructions and original 16-bit Thumb-1 instruction sets
 - Compared to 32-bit ARM instructions set, code size is reduced by ~26%, while keeping a similar performance
 - Capable of handling all processing requirements in one operation state

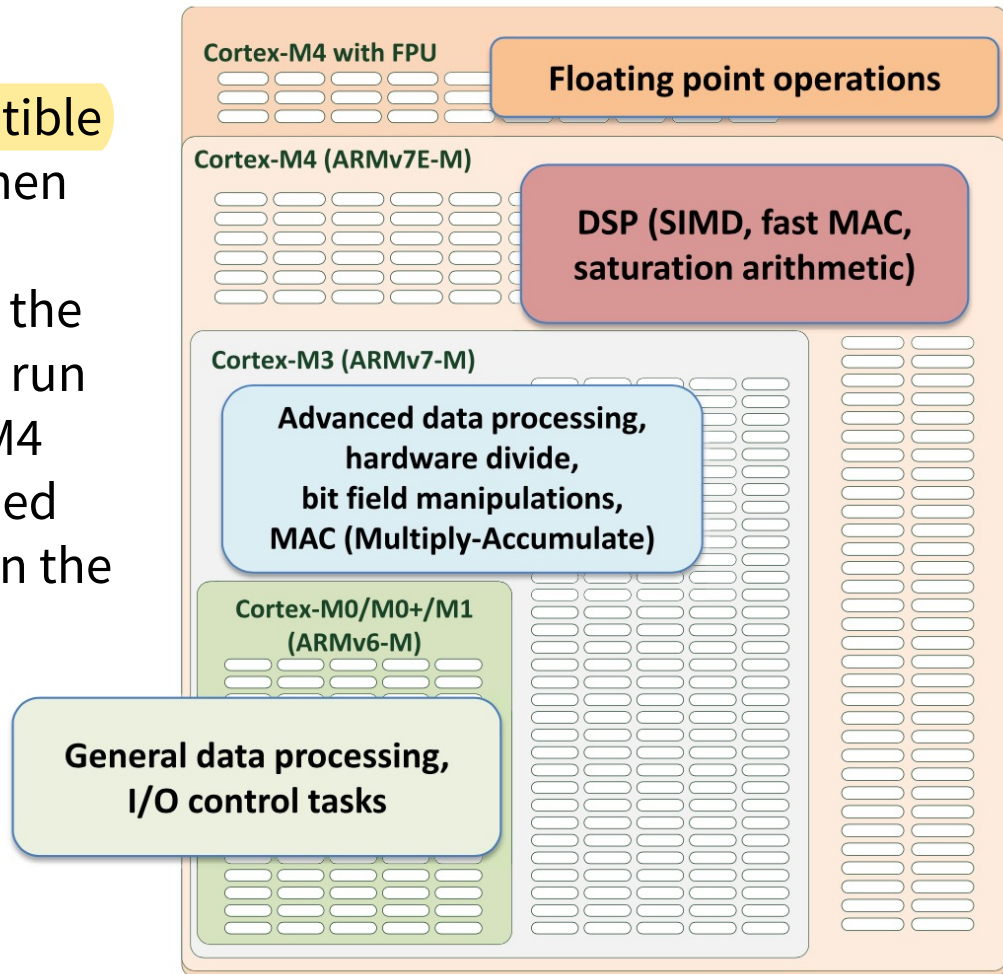
Cortex-M4 Instruction Set

- Cortex-M4 processor
 - ARMv7-M architecture
 - Supports 32-bit Thumb-2 instructions
 - Possible to handle all processing requirements in one operation state (Thumb state)
 - Compared with traditional ARM processors (use state switching), advantages include:
 - No state switching overhead – both execution time and instruction space are saved
 - No need to separate ARM code and Thumb code source files, which makes the development and maintenance of software easier
 - Easier to get optimised efficiency and performance

M4 Instructions: Simplified View

The design of the Cortex-M processors is **upward compatible** from Cortex-M0, to M3 and then to M4.

Therefore code compiled for the Cortex-M0/M1 processor can run on the Cortex-M3 or Cortex-M4 processors, and code compiled for Cortex-M3 can also run on the Cortex-M4 processor.



The Assembly Language Syntax

It is useful to have a general overview of what instructions are available, and of assembly language syntax.

Such knowledge in this area can be very useful for debugging.

label

mnemonic operand1, operand2, ... ; Comments

↑
optional,
reference to an
address location

for, while 循环中

↑
dest. of op. /
dest. of loading /
src. of storing

↑
operands: src. of op.
constants (immediate)
addresses to load/store

mnemonic: a short "nickname" for an instruction that is easy for the human user to remember

Instructions and Operands

- Some mnemonics can be used with different types of operands.
 - This can result in different instruction encodings.
- The no. of operands in an instruction depends on what type of instruction it is, and the syntax for the operands can also be different in each case.

```
MOVS R0, #0x12      ; Set R0 = 0x12 (hexadecimal)
MOVS R1, #'A'        ; Set R1 = ASCII char A
```

Immediate data (prefix with #): a simple way to get data is to make the data part of the instruction.

immediately available, constant

Cortex-M4 Instruction Set - Suffix

- Some instructions can be followed by suffixes to update processor flags or execute the instruction on a certain condition

Suffix	Description	Example	Example explanation
S	Update APSR (flags)	ADDS R1, #0x21	Add 0x21 to R1 and update APSR
EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE	Condition execution e.g. EQ= equal, NE= not equal, LT= less than	BNE label	Branch to the label if not equal

Example:

```
MOVS R0, R1 ; move R1 into R0, update APSR  
MOV R0, R1 ; move R1 into R0, not update APSR
```


Moving data within the processor

- The most basic operation in a microprocessor is to move data around inside the processor.
 - Move data from one register to another
 - Move data between a register and a special register (e.g., CONTROL, PRIMASK, FAULTMASK, BASEPRI)
 - Move an immediate constant into a register

; Copy value from R0 to R4

MOV R4, R0

; Copy value of PRIMASK (special register) to R7

MRS R7, PRIMASK

; Copy value of R2 into CONTROL (special register)

MSR CONTROL, R2

; Set R3 value to 0x34 with APSR update

MOVS R3, #0x34

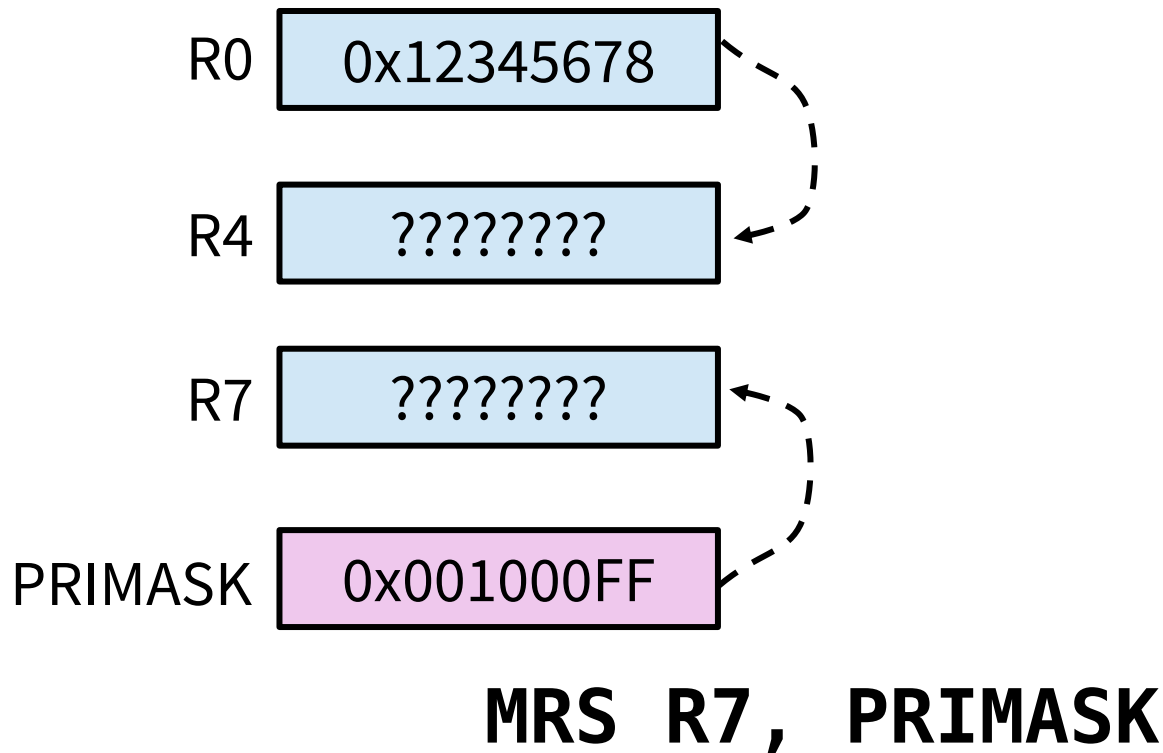
MRS:move register special

MSR:move special register

Reference: Chapter 5.6, Definitive Guide to ARM Cortex-M3 Processors

Moving data within the processor

MOV R4, R0



Memory Access Instructions

- The default data size for memory access is 32 bits. ^{one word}
 - If you wish to transfer 8 bits (1 byte), suffix by B. ^{one Byte}
 - If you wish to transfer 16 bits (2 bytes), suffix by H. ^{Half word}
 - Careful with sign extensions, suffix by S if the data is considered signed.

LDR. LD, load; R, register.
 STR. ST, store; R, register.

memory->register
 register->memory

; load/store 32 bits to/from R1 from/to memory addr R0

LDR R1, [R0]

STR R1, [R0]

; load/store 8 bits, unsigned (padded with 0's)

LDRB R1, [R0] 没正负号, 前面全填0

STRB R1, [R0]

; load/store 16 bits, signed (signed extended)

LDRSH R1, [R0] 有正负号, 用的最左边一位, 把其余空位填满

STRSH R1, [R0]

0xF314



0xFFFFF314

0xf = 1111B
negative

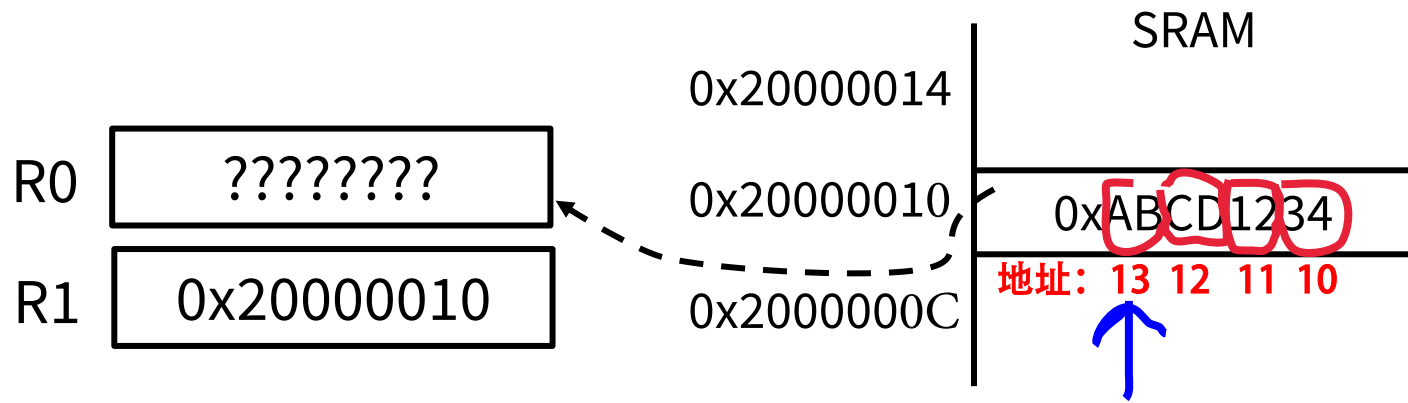
111111111111
用1填满, 也就成negative了

Immediate Offset (pre-index)

The memory address of the data transfer is the sum of a register value and an immediate constant value (offset). Sometimes this is referred to as “pre-index” addressing. For example:

```
LDRB R0, [R1, #0x3]
```

Read a byte value from address $R1+0x3$, and store the read data in R0.



The offset value can be positive or negative.

PC-related addressing (Literal)

- A memory access can generate the address value from the current PC value and an offset value.
- This is commonly needed for loading immediate values into a register, also known as **literal pool accesses**.

LDR R0, =0x12345678 ; Set R0 to 0x12345678

is a pseudo-instruction that is translated to

LDR R0, [PC, #offset] _____
... } **offset**
DCD 0x12345678 _____

The LDR instruction reads the memory at [PC + offset] and stores the value into R0. The assembler will calculate the offset for you so you don't have to worry about it.

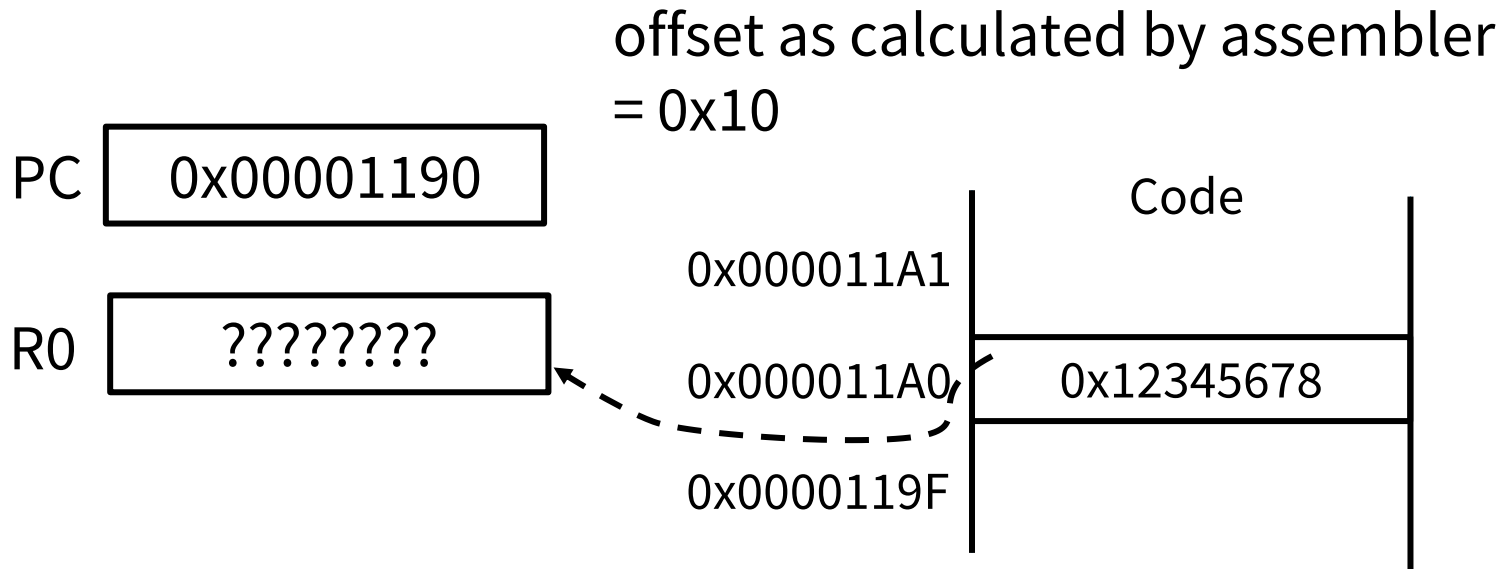
PC-related addressing (Literal)

```
LDR R0, [PC, #offset]
```

...

```
DCD 0x12345678
```

Example:



Data Insertion and Alignment

- Insert data inside programs
 - DCD: insert a word-size data declare constant data
 - DCB: insert a byte-size data declare constant Byte
 - ALIGN:
Used before inserting a word-size data
Uses a number to determine the alignment size

```
LDR R3, =MY_NUMBER    ; Get the mem loc of MY_NUMBER
LDR R4, [R3]           ; Read the value 0x12345678 into R4
LDR R0, =HELLO_TEXT    ; Get the starting addr of
                        ; HELLO_TEXT

...
ALIGN 4                LDRB R1, [R0]        ;R1='H'
                        LDRB R2, [R0, #1]    ;R2='e'
MY_NUMBER DCD 0x12345678
HELLO_TEXT DCB "Hello\n", 0 ; Null terminated string
```

Register Offset (pre-index)

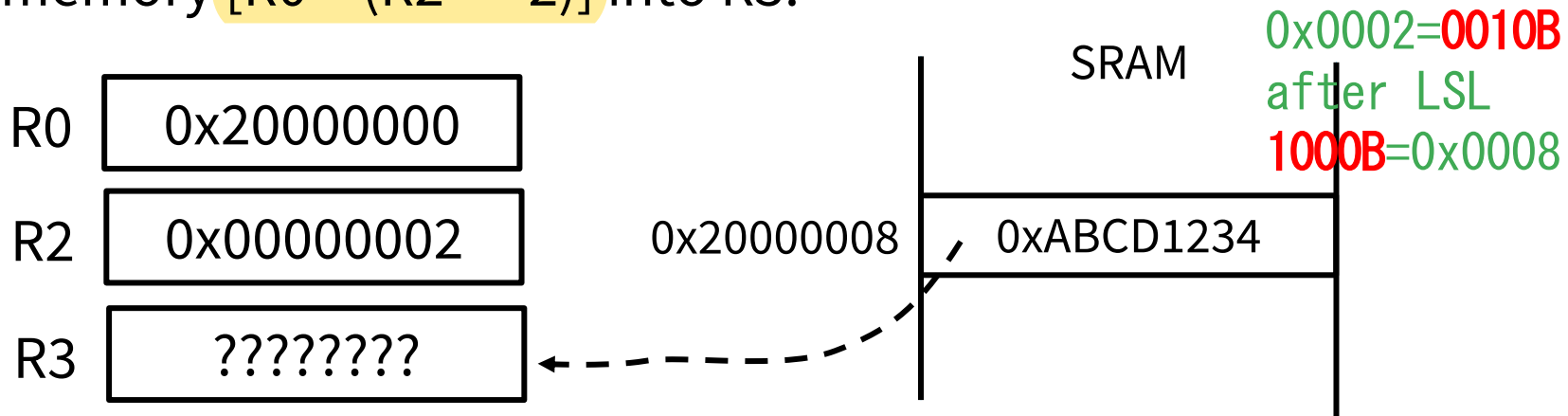
- The memory address of the data transfer is the sum of a register value and another register (offset).
- The index value can be shifted by a distance of 0 to 3 bits before being added to the base register (optional).

LDR R3, [R0, R2, LSL #2]

LSL: Logical Shift Left

Read memory $[R0 + (R2 \ll 2)]$ into R3.

R2 is the offset register



This address mode can be used with different sizes (B, H) and signs (S).

Post-index

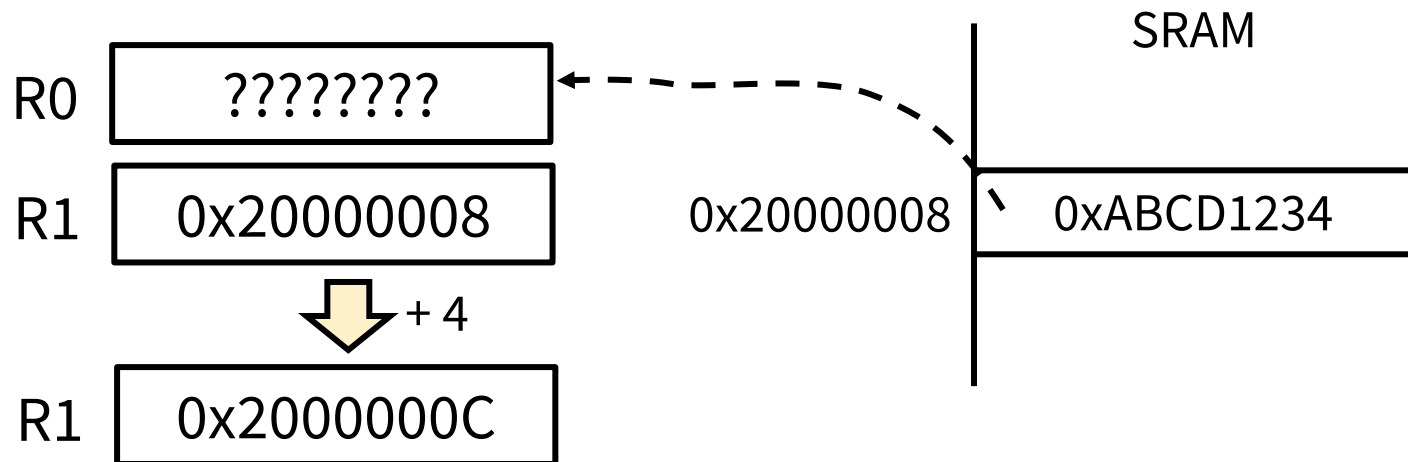
很像i++。在用完i后，再++。

LDR R0, [R1], #4。用完R1以后，再+4。

When the offset is provided as post-index (i.e. outside []), then it is not used in memory access, but used to update the address register.

LDR R0, [R1], #4

Read memory [R1] into R0 then update R1 to R1 + 4.



Arithmetic Operations

- Applications of MCUs often involve performing mathematical calculations on data in order to alter program flow and modify program actions.
- Many data processing instructions can have multiple instruction formats.

```
; R0 = R0 + R1
```

```
ADD R0, R0, R1
```

```
; R0 = R0 + 0x12 with APSR (flags) update
```

```
ADDS R0, R0, #0x12
```

```
; R0 = R1 + R2 + carry
```

```
ADC R0, R1, R2      add with carry
```

Arithmetic Operations (2)

In subtraction, carry serves as a borrow and is subtracted from the difference in SBC.

```
; R1 = R3 - R2  
SUB R1, R3, R2  
; R0 = R1 - 0x26 - not(carry)  
SBC R0, R1, #0x26
```

You should note that in MUL, the product is 32 bits.

```
; R0 = R1 * R2  
MUL R0, R1, R2  
; R0 = R1 / R2 (unsigned or signed)  
UDIV R0, R1, R2  
SDIV R0, R1, R2
```

Assembly Programming: Example

- Programming in assembly language is to write instructions to carry out meaningful tasks.
- It can be difficult when you get started since you don't know which instruction to use.
- Let's look at a simple example: $x = 2x - y + 3$

```
; x in R0; y in R1
LDR R0, =0x0A      ; test value x
LDR R1, =0x05      ; test value y
ADDS R0, R0, R0     ; 2x
SUBS R0, R0, R1     ; 2x - y
ADDS R0, R0, #3     ; 2x - y + 3
ADDS R1, R1, R1     ; 2y
SUBS R0, R0, R1     ; x-2y
SUBS R0, R0, #1     ; X-2Y-1
```

Can you try to change the program for $x = x - 2y - 1$?

Useful Resources

- Architecture Reference Manual:
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403c/index.html>
- Cortex-M4 Technical Reference Manual:
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439d/DDI0439D_cortex_m4_processor_r0p1_trm.pdf
- Cortex-M4 Devices Generic User Guide:
http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf
- VisUAL: A highly visual ARM emulator
 - <https://salmanarif.bitbucket.io/visual/index.html>