Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C

Chapter 5
Memory Access

Z. Gu

Fall 2025

Acknowledgement: Lecture slides based on Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, University of Maine https://web.eece.maine.edu/~zhu/book/

Overview

- How data is organized in memory?
  - Big-Endian vs Little-Endian

- How data is addressed?
  - Register offset
    - `LDR r1, [r0, r3]          ; offset = r3`
    - `LDR r1, [r0, r3, LSL #2] ; offset = r3 * 4`
  - Immediate offset
    - Pre-index: `LDR r1, [r0, #4]`
    - Post-index: `LDR r1, [r0], #4`
    - Pre-index with update: `LDR r1, [r0, #4]!`

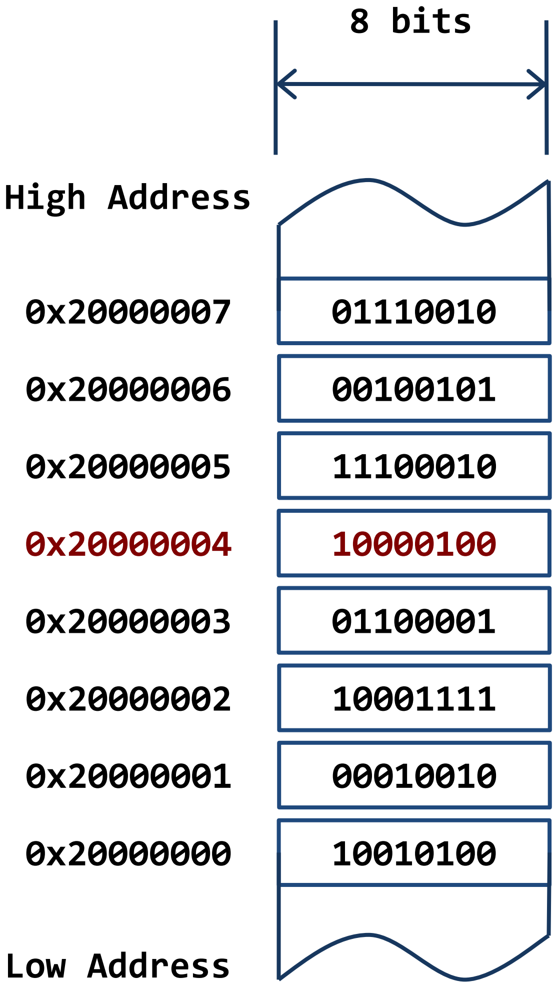
Logic View of Memory

- By grouping bits together we can store more values
  - 8 bits = 1 **byte**
  - 16 bits = 2 bytes = 1 **halfword**
  - 32 bits = 4 bytes = 1 **word**
- From software perspective, memory is an addressable array of bytes.
  - The byte stored at the memory address 0x20000004 is 0b10000100

0b10000100 → 0x84 → 132

Binary Hexadecimal Decimal

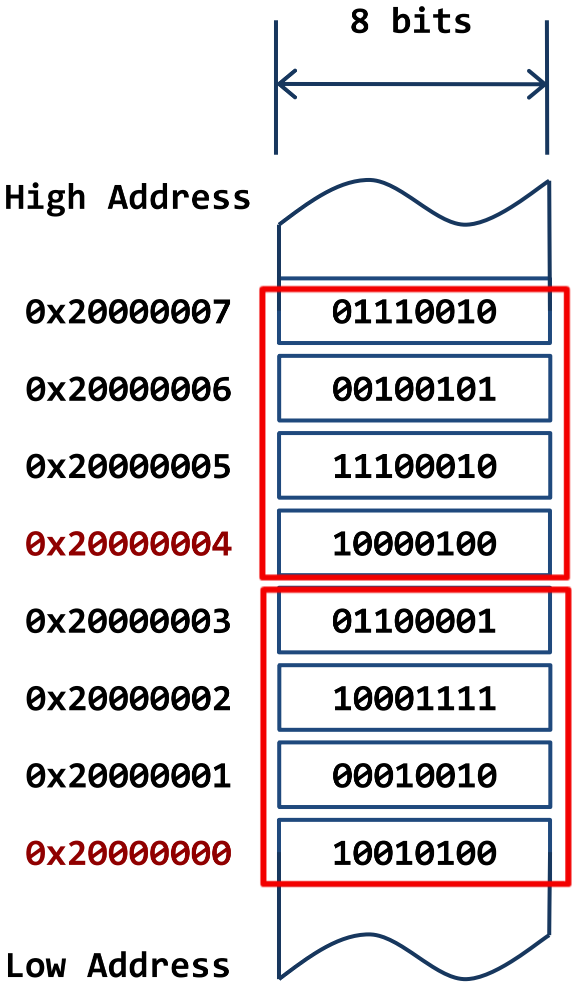
Computer memory is *byte-addressable*!



0x20000007	01110010
0x20000006	00100101
0x20000005	11100010
0x20000004	10000100
0x20000003	01100001
0x20000002	10001111
0x20000001	00010010
0x20000000	10010100

Logic View of Memory

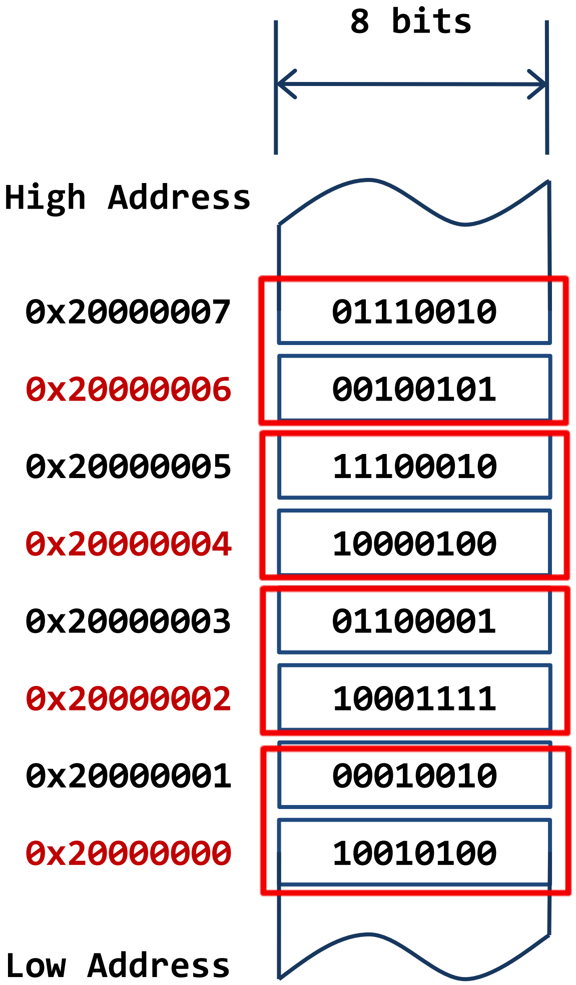
- When we refer to memory locations by address, we can only do so in units of bytes, halfwords or words
- Words
  - 32 bits = 4 bytes = 1 word = 2 halfwords
  - A word can only be stored at an address that's divisible by 4 (Word-address mod 4 = 0, binary address ends with 00)
    - Memory address of a word is the lowest address of all four bytes in that word.
    - Two words at addresses: `0x20000000` and `0x20000004`
  - A halfword can only be stored at an address that's divisible by 2 (Halfword-address mod 2 = 0, binary address ends with 0)
    - Memory address of a halfword is the lowest address of all 2 bytes in that word.



0x20000007	01110010
0x20000006	00100101
0x20000005	11100010
0x20000004	10000100
0x20000003	01100001
0x20000002	10001111
0x20000001	00010010
0x20000000	10010100

Logic View of Memory

- Halfwords
  - 16 bits = 2 bytes = 1 halfword
  - The right diagram has four halfwords at addresses of:
    - 0x20000000
    - 0x20000002
    - 0x20000004
    - 0x20000006

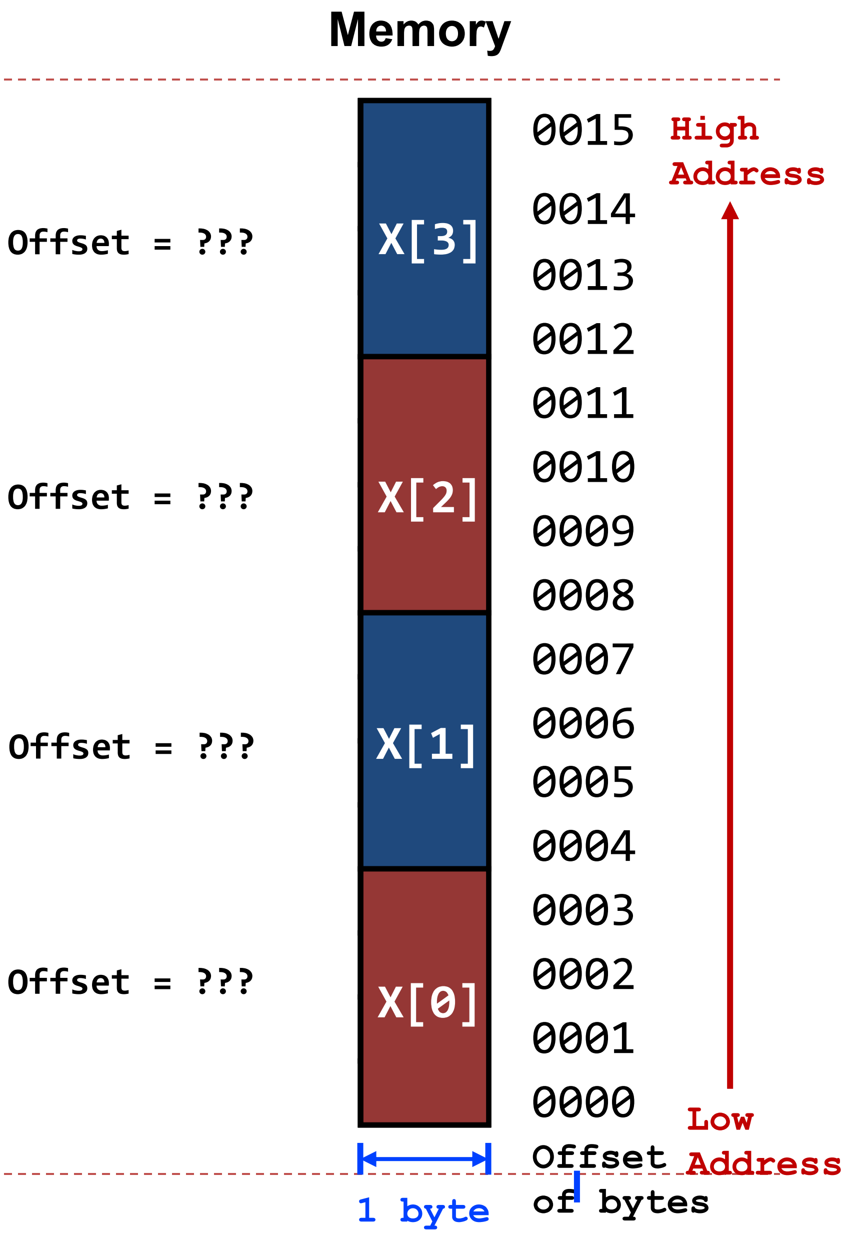


0x20000007	01110010
0x20000006	00100101
0x20000005	11100010
0x20000004	10000100
0x20000003	01100001
0x20000002	10001111
0x20000001	00010010
0x20000000	10010100

Quiz

```
uint32_t X[4];
```

What are their memory address offsets?



Memory

Offset = ???

Offset = ???

Offset = ???

Offset = ???

X[3]

X[2]

X[1]

X[0]

0015
0014
0013
0012
0011
0010
0009
0008
0007
0006
0005
0004
0003
0002
0001
0000

High Address

Low Address

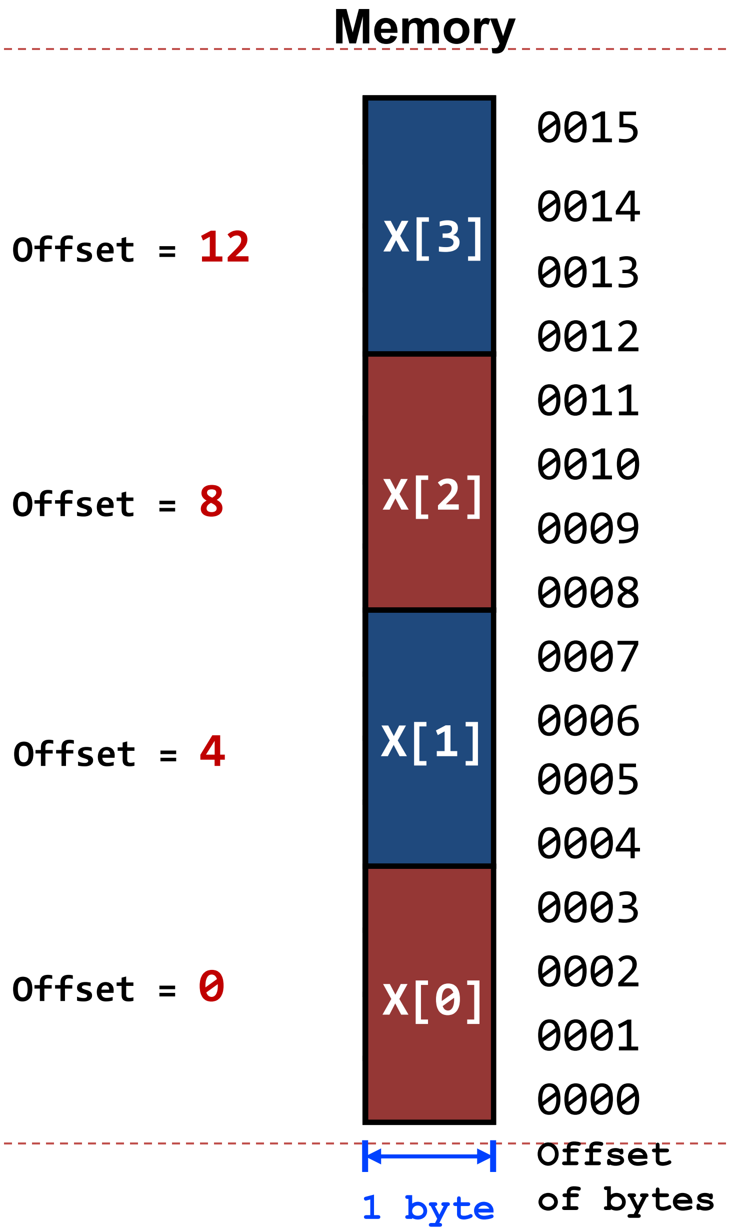
1 byte

Offset of bytes

Quiz ANS

`uint32_t X[4];`

What are their memory address offsets?



Memory

Offset = 12

Offset = 8

Offset = 4

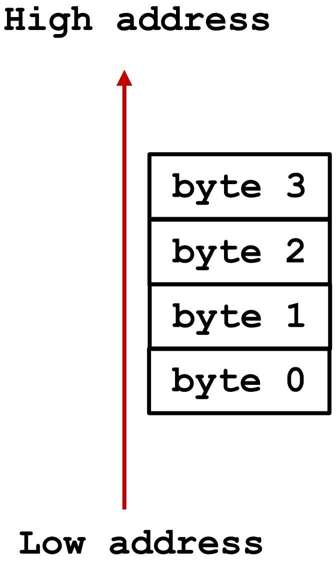
Offset = 0

If the array starts at address pAddr = 0000,

- Memory address of X[0] is pAddr = 0000
- Memory address of X[1] is pAddr + 4 = 0004
- Memory address of X[2] is pAddr + 8 = 0008
- Memory address of X[3] is pAddr + 12 = 0012

Sequential words are at addresses incrementing by 4, since each array element of type uint32_t is 4 bytes (32 bits)

Endianess



High address

byte 3

byte 2

byte 1

byte 0

Low address

MSB

Little-Endian

LSB

LSB is at lower address

LSB

Big-Endian

MSB

MSB is at lower address

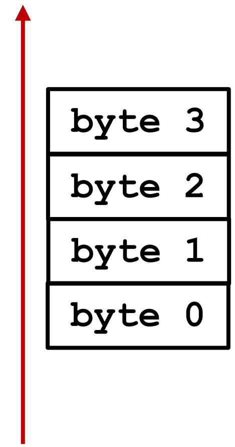
Gulliver's Travels (by Jonathan Swift, published in 1726):

- Two religious sects of Lilliputians
- The Little-Endians crack open their eggs from the little end
- The Big-Endians break their on the big end

Endianess

Endian: byte order, not bit order!

High address



byte 3

byte 2

byte 1

byte 0

Low address

MSB

Little-Endian

LSB

LSB is at lower address

LSB

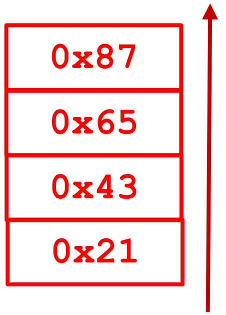
Big-Endian

MSB

MSB is at lower address

Little-Endian

High address



0x87

0x65

0x43

0x21

Low address

`uint32_t a = 0x87654321`

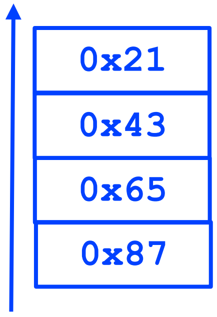
Reading from the top

byte 3	byte 2	byte 1	byte 0
0x87	0x65	0x43	0x21
byte 0	byte 1	byte 2	byte 3

Reading from the bottom

Big-Endian

High address



0x21

0x43

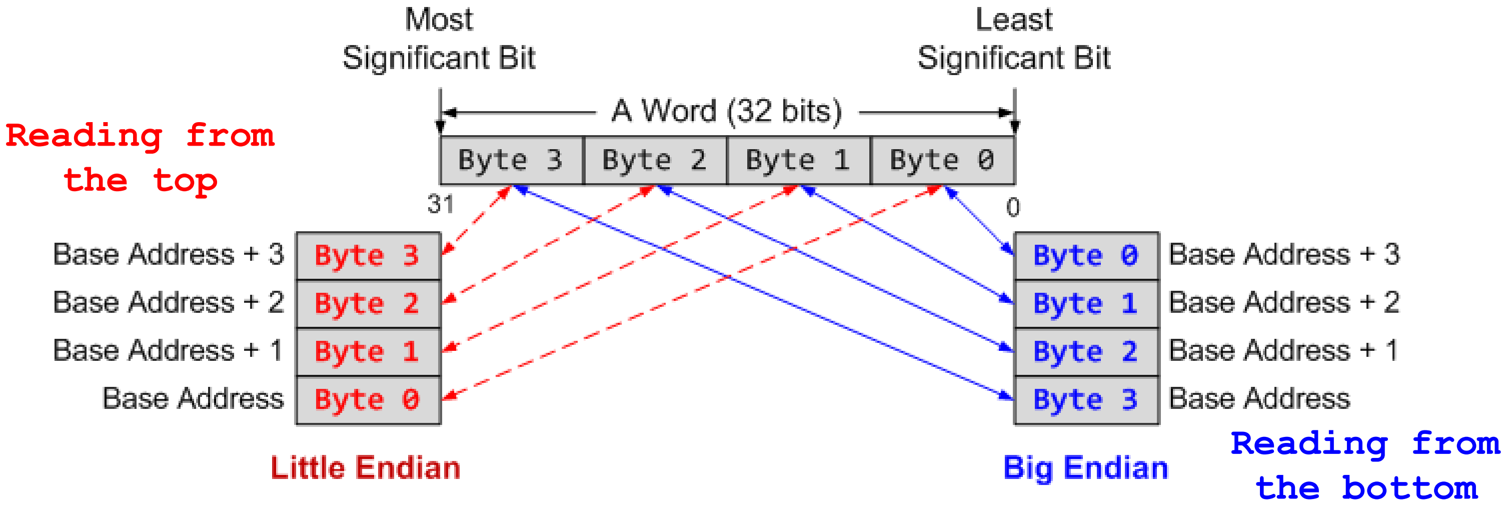
0x65

0x87

Low address

Endianess

- Little-Endian
  - Least significant byte (LSB) is stored at lowest (least) address of a word
- Big-Endian
  - Most significant byte (MSB) is stored at lowest (least) address of a word
- Regardless of endianness, the address of a word is defined as the lowest address of all bytes it occupies.
- ARM is *Little-Endian by default*.
  - It can be made Big-Endian by configuration.



Endianness Example

- **Little-Endian**
  - **LSB is at lower address**

```
                              Memory     Value
                              Offset  (LSB) (MSB)
                              ======  ===========
uint8_t a  = 1;               0x0000  01 02 FF 00
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;      0x0004  78 56 34 12
```

- **Big-Endian**
  - **MSB is at lower address**

```
                              Memory     Value
                              Offset  (LSB) (MSB)
                              ======  ===========
uint8_t a  = 1;               0x0000  01 02 00 FF
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;      0x0004  12 34 56 78
```

- **For uint8_t a and b, each with size of 1 Byte: No difference**
- **Little-endian:**
  - **For uint16_t c with size of 2 Bytes: LSB FF is at lower address and MSB 00 is at higher address**
  - **For uint32_t d with size of 4 Bytes: LSB 78 is at lower address and MSB 12 is at higher address.**
- **Big-endian:**
  - **For uint16_t c with size of 2 Bytes: LSB FF is at higher address and MSB 00 is at lower address**
  - **For uint32_t d with size of 4 Bytes: LSB 78 is at higher address and MSB 12 is at lower address.**

Example

If Big-Endian is used, the word stored at address 0x20008000 is

If Little-Endian is used, the word stored at address 0x20008000 is

Memory Address	Memory Data
0x20008003	0xA7
0x20008002	0x90
0x20008001	0x8C
0x20008000	0xEE

Example

If Big-Endian is used, the word stored at address 0x20008000 is

0xEE8C90A7

If Little-Endian is used, the word stored at address 0x20008000 is

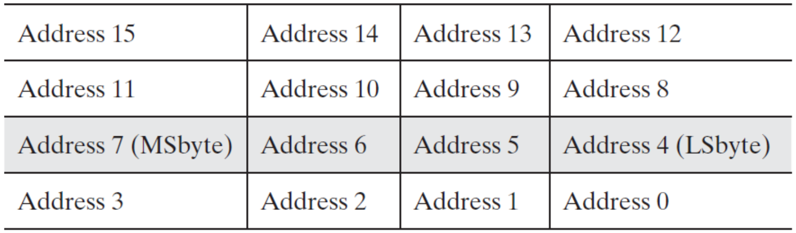
0xA7908CEE

Endianness specifies byte order, not bit order in a byte!

Memory Address	Memory Data
0x20008003	0xA7
0x20008002	0x90
0x20008001	0x8C
0x20008000	0xEE

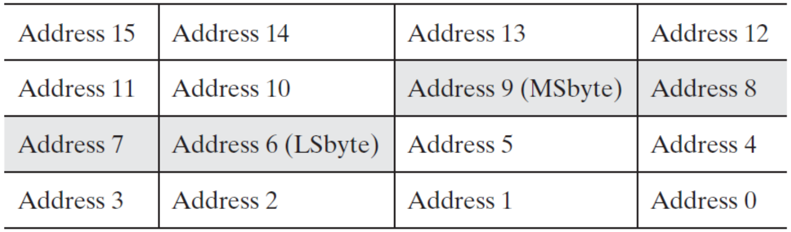
Data Alignment

- Assume a byte-addressable memory with a data bus that is 32 bits (4 bytes) wide
- Consider 16 bytes of memory (addresses 0 to 15) arranged as four 32-bit words (4 bytes each)



Address 15	Address 14	Address 13	Address 12
Address 11	Address 10	Address 9	Address 8
Address 7 (MSbyte)	Address 6	Address 5	Address 4 (LSbyte)
Address 3	Address 2	Address 1	Address 0

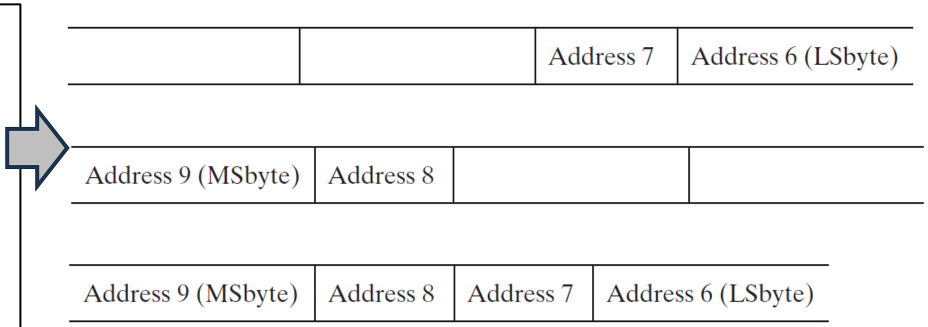
Well-aligned: each word begins on a mod-4 address, which can be read in a single memory cycle



Address 15	Address 14	Address 13	Address 12
Address 11	Address 10	Address 9 (MSbyte)	Address 8
Address 7	Address 6 (LSbyte)	Address 5	Address 4
Address 3	Address 2	Address 1	Address 0

Ill-aligned: a word begins on address 6, not a mod-4 address, which can be read in 2 memory cycles

The first read cycle would retrieve 4 bytes from addresses 4 through 7; of these, the bytes from addresses 4 and 5 are discarded, and those from addresses 6 and 7 are moved to the far right;
The second read cycle retrieves 4 bytes from addresses 8 through 11; the bytes from addresses 10 and 11 are discarded, and those from addresses 8 and 9 are moved to the far left;
Finally, the two halves are combined to form the desired 32-bit operand.



		Address 7	Address 6 (LSbyte)

Address 9 (MSbyte)	Address 8		

Address 9 (MSbyte)	Address 8	Address 7	Address 6 (LSbyte)

Load-Modify-Store

C statement

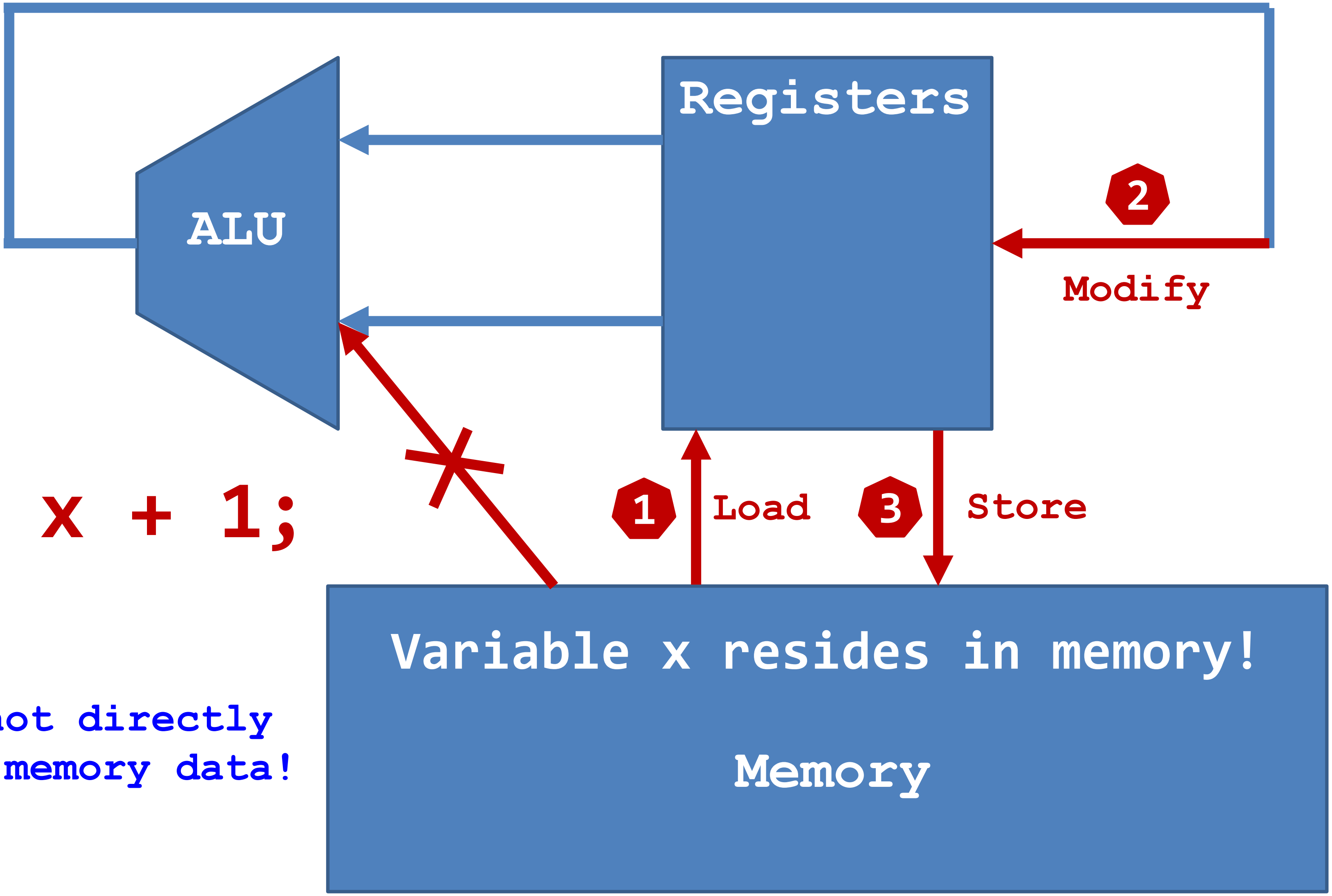
```c
x = x + 1;
```

Assume variable X resides in memory and is a 32-bit integer

```
; Assume the memory address of x is stored in r1


LDR r0, [r1]        ; load value of x from memory
ADD r0, r0, #1      ; x = x + 1
STR r0, [r1]        ; store x into memory
```

3 Steps: Load, Modify, Store



Variable x resides in memory!

```
x = x + 1;
```

ALU cannot directly operate memory data!

Load Instructions

- **LDR rt, [rs]**
  - **Read from memory**
  - Mnemonic: **L**oa**D** to **R**egister (**LDR**)
  - rs specifies the memory address
  - rt holds the 32-bit value fetched from memory

  - For Example:

```
; Assume r0 = 0x08200004
; Load a word:
LDR r1, [r0]            ; r1 = Memory.word[0x08200004]
```

Store Instructions

- **STR rt, [rs]**
  - **Write into memory**
  - Mnemonic: **ST**ore from **R**egister (**STR**)
  - rs specifies memory address
  - Save the content of rt into memory
  - For Example:

```
; Assume r0 = 0x08200004
; Store a word
STR r1, [r0]     ; Memory.word[0x08200004] = r1
```

Load/Store a Byte, Halfword, Word

```
LDRxxx R0, [R1]
; Load data from memory into a 32-bit register
```

LDR	Load Word	uint32_t/int32_t	unsigned or signed int
LDRB	Load Byte	uint8_t	unsigned char
LDRH	Load Halfword	uint16_t	unsigned short int
LDRSB	Load Signed Byte	int8_t	signed char
LDRSH	Load Signed Halfword	int16_t	signed short int

```
STRxxx R0, [R1]
; Store data extracted from a 32-bit register into memory
```

STR	Store Word	uint32_t/int32_t	unsigned or signed int
STRB	Store Lower Byte	uint8_t/int8_t	unsigned or signed char
STRH	Store Lower Halfword	uint16_t/int16_t	unsigned or signed short

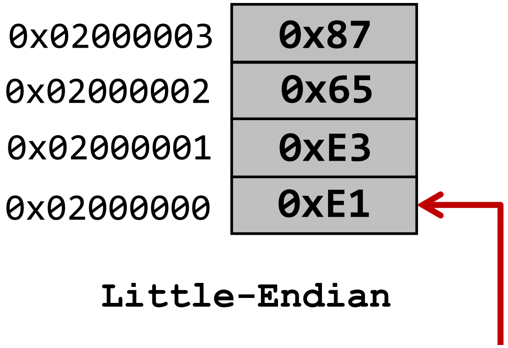
Load a Byte, Half-word, Word (Little-Endian)

Load a Byte

```
LDRB r1, [r0]
```

0x00	0x00	0x00	0xE1

31 0



0x02000003	0x87
0x02000002	0x65
0x02000001	0xE3
0x02000000	0xE1

Little-Endian

Assume
r0 = 0x02000000

Load a Halfword

```
LDRH r1, [r0]
```

0x00	0x00	0xE3	0xE1

31 0

Load a Word

```
LDR r1, [r0]
```

0x87	0x65	0xE3	0xE1

31 0

LDRH "Load Register Halfword“: it loads a 16-bit halfword value from the memory address pointed to by register r0 into register r1.The loaded 16-bit value is zero-extended to fill the 32-bit register r1. This means the upper 16 bits of r1 will be set to zero regardless of the halfword data.
LDRB "Load Register Byte“: it loads 8-bit byte value from the memory address pointed to by register r0 into register r1, and zero-extends it.

Sign Extension (Little-Endian)

Load a Signed Byte

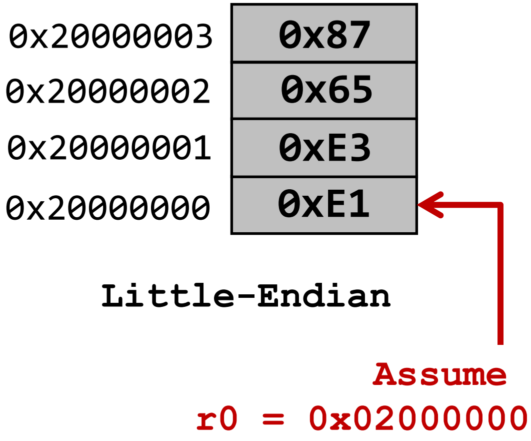
```
LDRSB r1, [r0]
```

0xFF	0xFF	0xFF	0xE1
31			0

Load a Signed Halfword

```
LDRSH r1, [r0]
```

0xFF	0xFF	0xE3	0xE1
31			0



0x20000003	0x87
0x20000002	0x65
0x20000001	0xE3
0x20000000	0xE1

Little-Endian

Assume
r0 = 0x02000000

LDRSH "Load Register Signed Halfword“
LDRSB "Load Register Signed Byte“
Similar to LDRH and LDRB, except each sign-extends the value to fill the 32-bit register, not zero-extend. Facilitate subsequent 32-bit signed arithmetic.

Address Modes: Offset in Register

- Address accessed by **LDR/STR** is specified by a base register plus an offset
- Offset can be hold in **a register**

```
LDR r0,[r1,r2]
```

- Base memory address hold in register r1
- Offset hold r2
- Target address = r1 + r2

```
LDR r0,[r1,r2,LSL #2]
```

- Base memory address hold in register r1
- Offset = r2, LSL #2
- Target address = r1 + r2 * 4

Address Modes: Immediate Offset

- Address accessed by **LDR/STR** is specified by a base register plus an offset
- Offset can be **an immediate value**

```
LDR r0,[r1,#8]
```

- `Base memory address hold in register r1`
- `Offset is an immediate value`
- `Target address = r1 + 8`

Three modes for immediate offset:

- Pre-index,
- Post-index,
- Pre-index with Update

Addressing Mode:
Pre-index *vs* Post-index

- Pre-index

```
LDR r1, [r0, Offset]
```

- Post-index

```
LDR r1, [r0], Offset
```

- Pre-index with Update

```
LDR r1, [r0, Offset]!
```

The table assumes r0 = 0x100, offset = 4 bytes (#4)

Mode	Address used for Load	Base register update	Example (r0=0x100)

Pre-index
LDR r1, [r0, #4]
	r0 + offset (0x104)	No	r1 = data[0x104];
			r0 = 0x100

Post-index
LDR r1, [r0], #4
	r0 (0x100)	Yes, after load	r1 = data[0x100];
			r0 = 0x104

Pre-index w/ Update
LDR r1, [r0, #4]!
	r0 + offset (0x104)	Yes, before load	r1 = data[0x104];
			r0 = 0x104

Pre-index

Pre-Index: LDR r1, [r0, #4]

Offset: range is -255 to +255

Assume: r0 = 0x20008000

Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

- Calculates address by adding the offset (here, #4) to the base register (r0) before the load. Loads data from the resulting address r0+4 into r1. The base register (r0) is not updated.
- Example: instruction accesses memory at r0 + 4 = 0x20008004, but r0 remains to be 0x20008000 after execution.

Pre-index

Pre-Index: LDR r1, [r0, #4]

Offset: range is -255 to +255

Assume: r0 = 0x20008000

Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

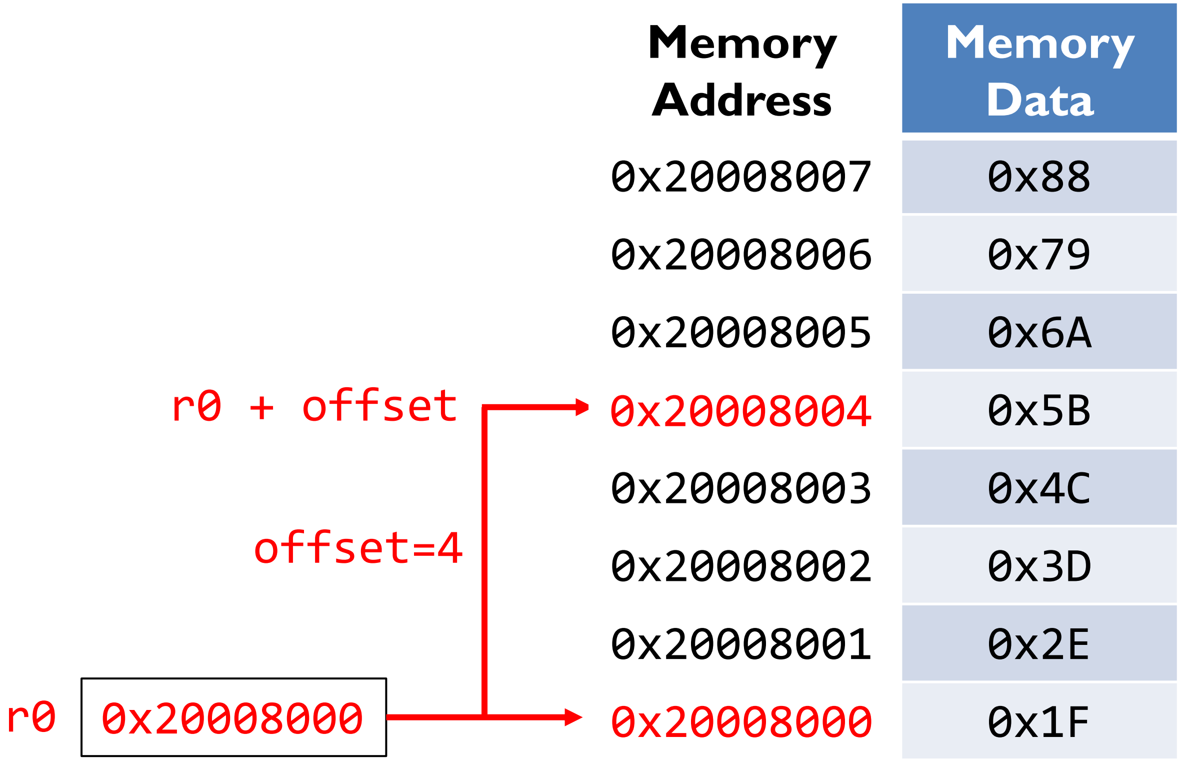
r0 0x20008000 → 0x20008000

Pre-index

Pre-Index: LDR r1, [r0, #4]

Offset: range is -255 to +255

Assume: r0 = 0x20008000



Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

r0 + offset

offset=4

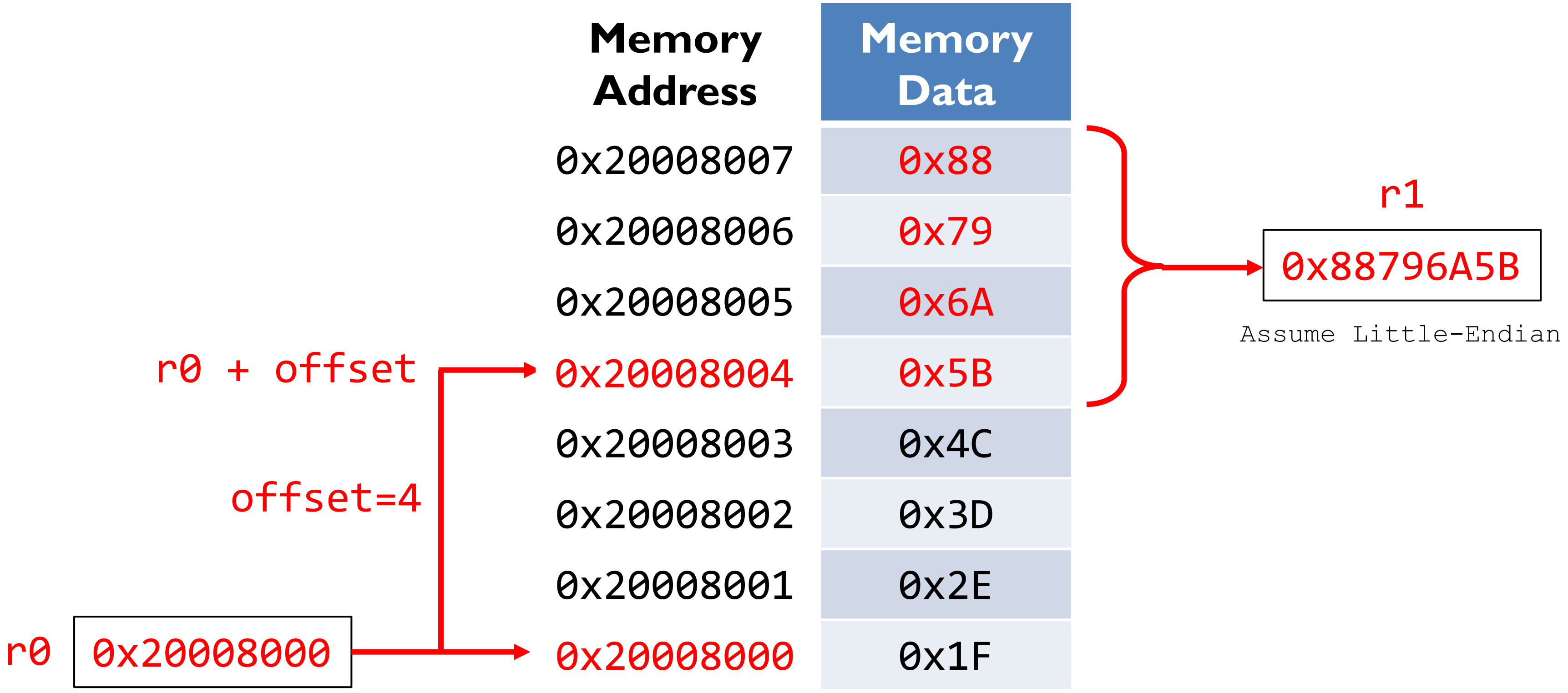
r0 0x20008000

Pre-index

Pre-Index: LDR r1, [r0, #4]

Assume: r0 = 0x20008000

Offset: range is -255 to +255



Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

r0 + offset

offset=4

r0 0x20008000

r1 0x88796A5B

Assume Little-Endian

Accessing an Array

- C code

```
uint32_t array[10];
array[0] += 5;
array[1] += 5;
```

Assume the memory address of the array starts at 0x20008000.

- Pre-index

Assume r0 = 0x20008000.

```
LDR r1, [r0]      ; Read array[0]
ADD r1, r1, #5
STR r1, [r0]      ; Write to array[0]

LDR r1, [r0, #4] ; Read array[1]
ADD r1, r1, #5
STR r1, [r0, #4] ; Write to array[1]
```

Post-index

Post-Index: LDR r1, [r0], #4

Offset: range is -255 to +255

Assume: r0 = 0x20008000

Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

- Loads data from the address currently in r0 into r1. After the load, updates the base register (r0) by adding the offset (#4).
- Example: instruction accesses memory at r0 = 0x20008000, then increments r0 by the offset of 4 to r0 + 4 = 0x20008004 after execution.

Post-index

Post-Index: LDR r1, [r0], #4

Assume: r0 = 0x20008000

Offset: range is -255 to +255

Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

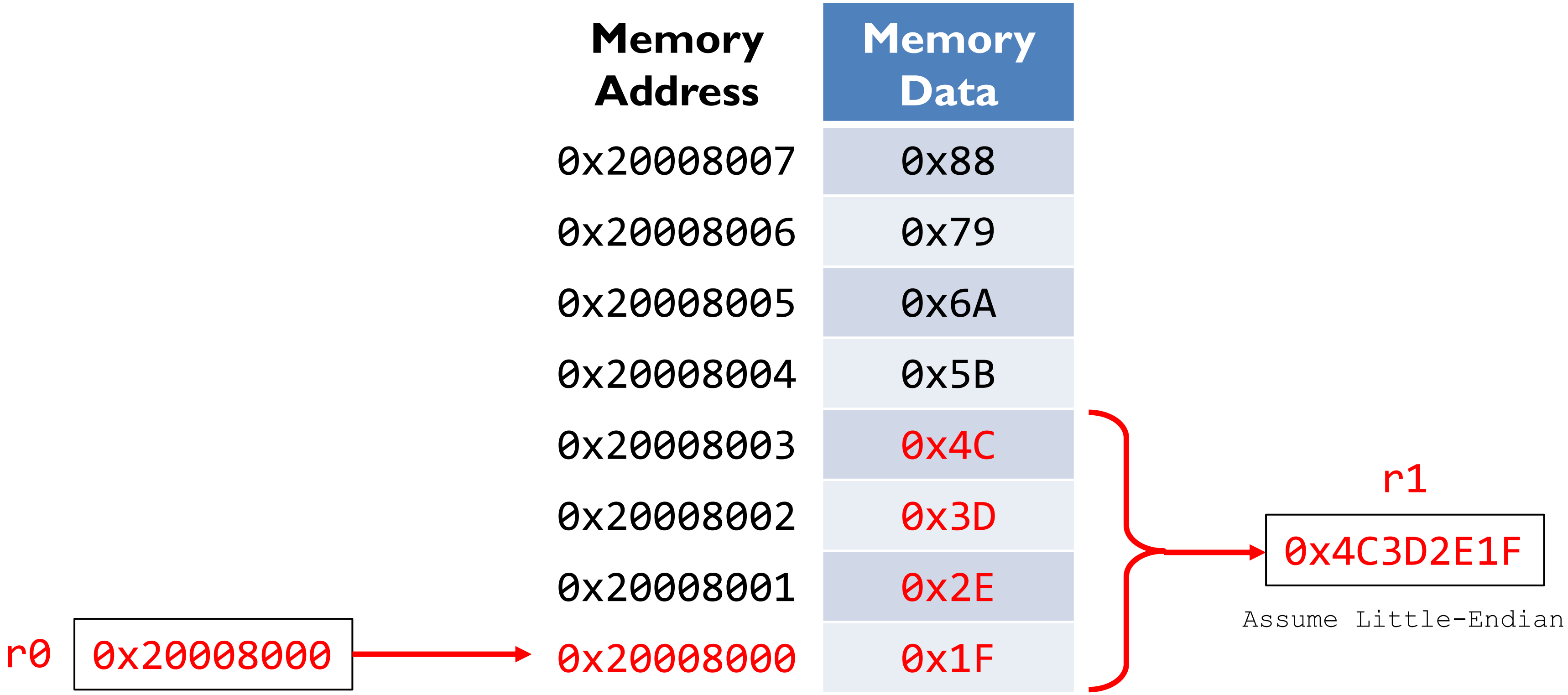
r0 0x20008000 → 0x20008000

Pre-index

Pre-Index: LDR r1, [r0, #4]

Offset: range is -255 to +255

Assume: r0 = 0x20008000



Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

r0 0x20008000

r1 0x4C3D2E1F

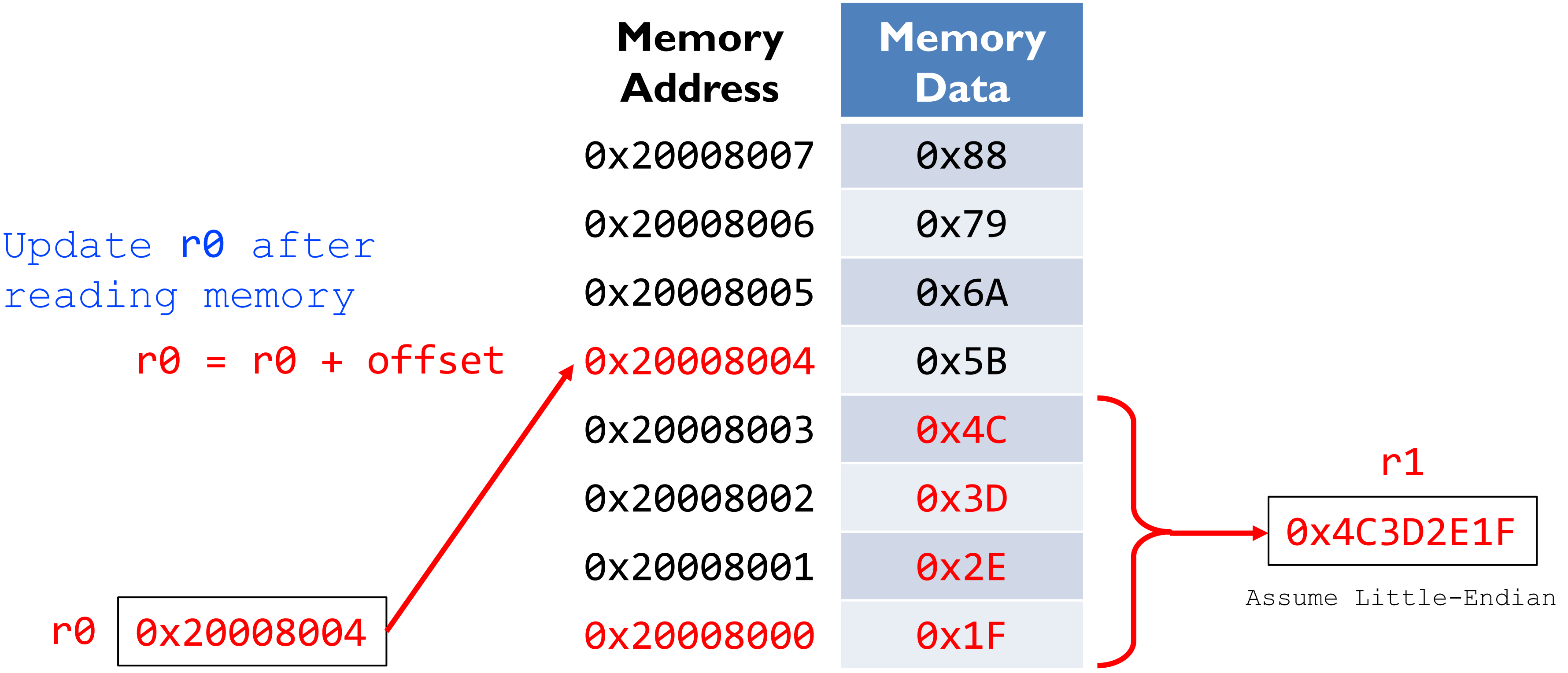
Assume Little-Endian

Pre-index

Pre-Index: LDR r1, [r0, #4]

Offset: range is -255 to +255

Assume: r0 = 0x20008000



Update r0 after reading memory

r0 = r0 + offset

Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

r0 0x20008004

r1 0x4C3D2E1F

Assume Little-Endian

Pre-index with Update

Pre-Index with Update: LDR r1, [r0, #4]!

Offset: range is -255 to +255

Assume: r0 = 0x20008000

Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

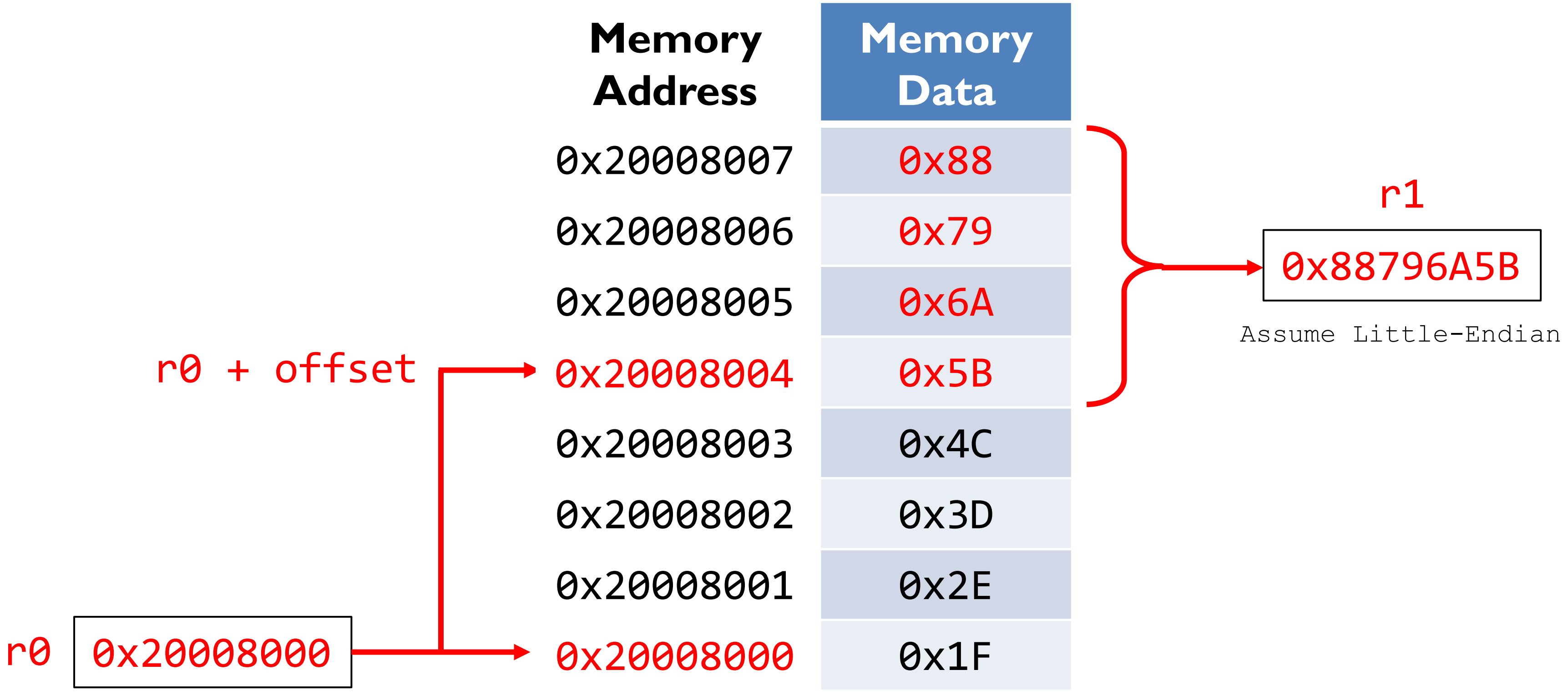
- First, adds the offset (#4) to the base register (r0), then loads from this updated address r0 + 4. Base register r0 is set to r0 + 4 afterwards.
- Example: instruction accesses memory at r0 + 4 = 0x20008004, and also sets r0 to 0x20008004 after execution.

Pre-index

Pre-Index with Update: LDR r1, [r0, #4]!

Assume: r0 = 0x20008000

Offset: range is -255 to +255



Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

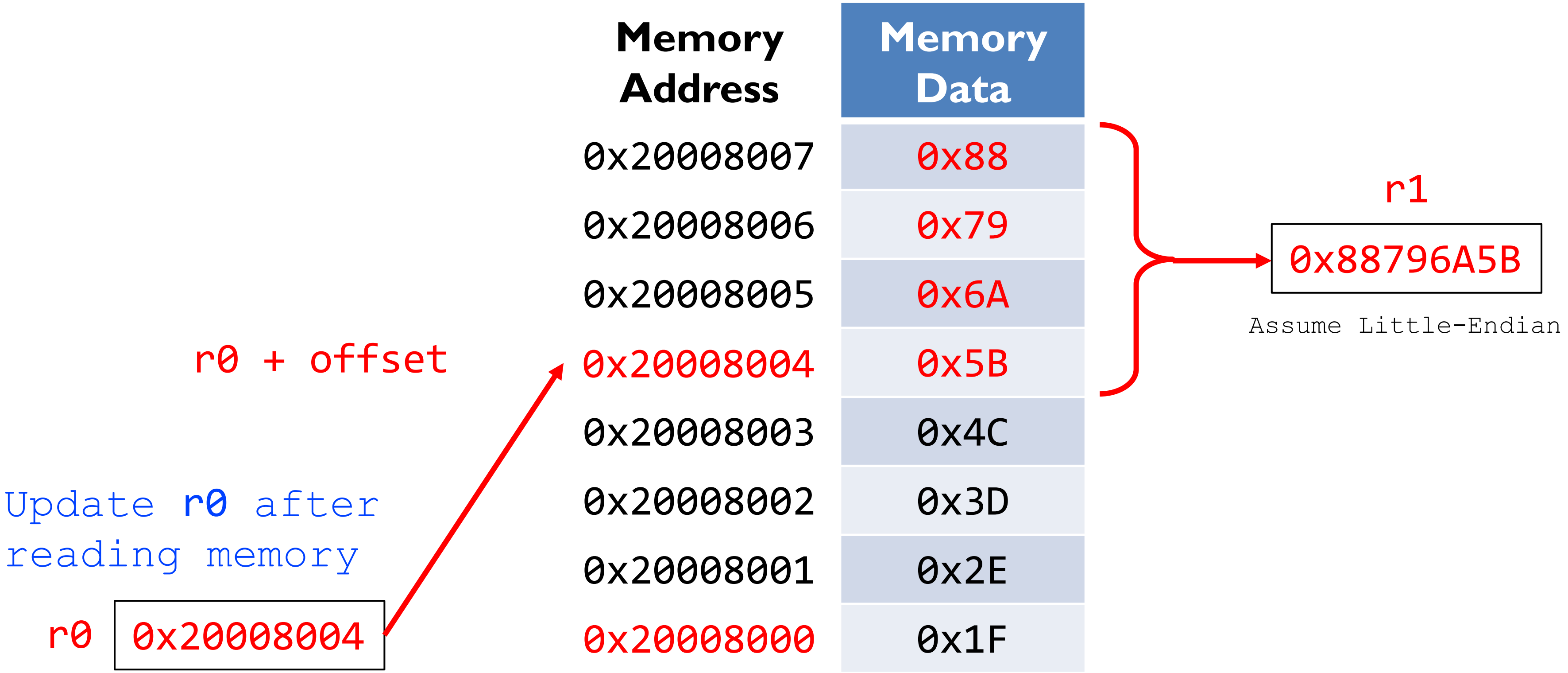
Assume Little-Endian

Pre-index

Pre-Index with Update: `LDR r1, [r0, #4]!`

Assume: r0 = 0x20008000

Offset: range is -255 to +255



Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

r0 + offset

r1 0x88796A5B

Assume Little-Endian

Update **r0** after reading memory

r0 0x20008004

Summary of Pre-index and Post-index

Index Format	Example	Equivalent
Pre-index	LDR r1, [r0, #4]	r1 ← memory[r0 + 4], r0 is unchanged

Pre-index with update	LDR r1, [r0, #4]!	r1 ← memory[r0 + 4]
		r0 ← r0 + 4

Post-index	LDR r1, [r0], #4	r1 ← memory[r0]
		r0 ← r0 + 4

In ARM Cortex-M/Thumb instruction set, for halfword and signed byte/halfword load/store instructions, the offset is an unsigned 8-bit immediate (0–255), and the U bit selects addition or subtraction, yielding an effective signed range of [−255, +255] around the base register.

In ARM (A32) instruction set, for word and unsigned byte LDR/STR, the immediate is typically a 12-bit unsigned value (0–4095, with an effective signed range of [−4095, +4095]

Example (Little-Endian ordering)

```
LDRH r1, [r0]
; r0 = 0x20008000
```

r1 before load

0x12345678

r1 after load

Memory Address	Memory Data
0x20008003	0x89
0x20008002	0xAB
0x20008001	0xCD
0x20008000	0xEF

Example ANS (Little-Endian ordering)

```
LDRH r1, [r0]
; r0 = 0x20008000
```

r1 before load

```
0x12345678
```

r1 after load

```
0x0000CDEF
```

Memory Address	Memory Data
0x20008003	0x89
0x20008002	0xAB
0x20008001	0xCD
0x20008000	0xEF

Example (Endianness does not matter for single byte)

```
LDRSB r1, [r0]
; r0 = 0x20008000
```

r1 before load

0x12345678

r1 after load

Memory Address	Memory Data
0x20008003	0x89
0x20008002	0xAB
0x20008001	0xCD
0x20008000	0xEF

Example ANS (Endianness does not matter for single byte)

```
LDRSB r1, [r0]
; r0 = 0x20008000
```

r1 before load

0x12345678

r1 after load

0xFFFFFFEF

Memory Address	Memory Data
0x20008003	0x89
0x20008002	0xAB
0x20008001	0xCD
0x20008000	0xEF

Example (Little-Endian ordering)

```
STR r1, [r0, #4]
; r0 = 0x20008000,  r1=0x76543210
```

r0 before the store

0x20008000

r0 after the store

	Memory Address	Memory Data
	0x20008007	0x00
	0x20008006	0x00
	0x20008005	0x00
	0x20008004	0x00
	0x20008003	0x00
	0x20008002	0x00
	0x20008001	0x00
r0 →	0x20008000	0x00

Example ANS (Little-Endian ordering)

```
STR r1, [r0, #4]
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

0x20008000

r0 after store

0x20008000

	Memory Address	Memory Data
	0x20008007	0x76
	0x20008006	0x54
	0x20008005	0x32
	0x20008004	0x10
	0x20008003	0x00
	0x20008002	0x00
	0x20008001	0x00
r0 ⇒	0x20008000	0x00

Example (Little-Endian ordering)

```
STR r1, [r0], #4
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

0x20008000

r0 after store

	Memory Address	Memory Data
	0x20008007	0x00
	0x20008006	0x00
	0x20008005	0x00
	0x20008004	0x00
	0x20008003	0x00
	0x20008002	0x00
	0x20008001	0x00
r0 ➡	0x20008000	0x00

Example ANS (Little-Endian ordering)

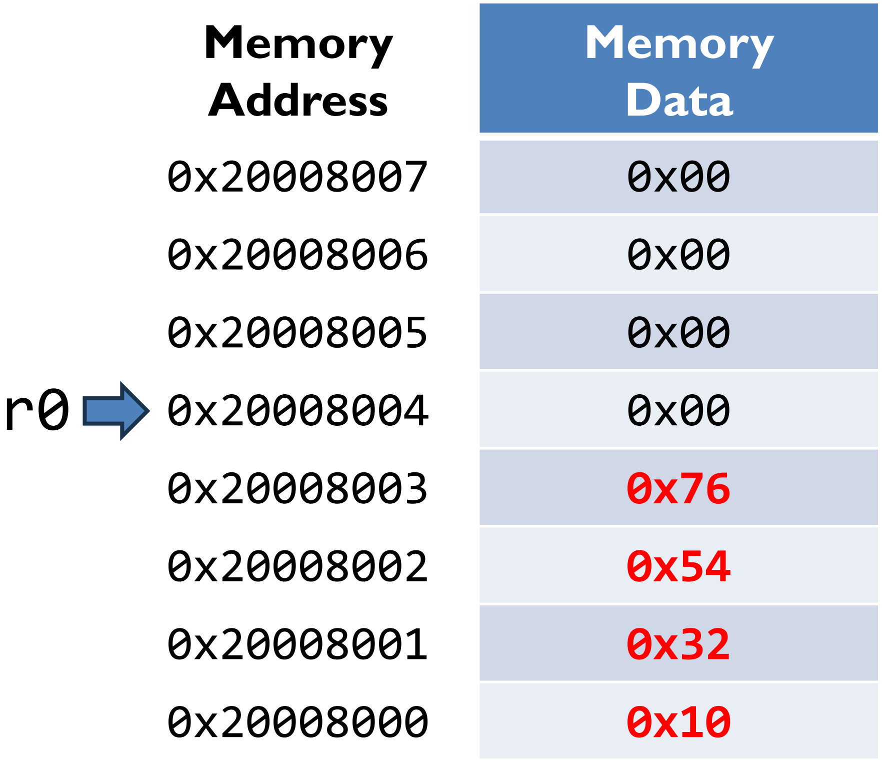
```
STR r1, [r0], #4
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

0x20008000

r0 after store

0x20008004



	Memory Address	Memory Data
	0x20008007	0x00
	0x20008006	0x00
	0x20008005	0x00
r0 ➡	0x20008004	0x00
	0x20008003	0x76
	0x20008002	0x54
	0x20008001	0x32
	0x20008000	0x10

Example

```
STR r1, [r0, #4]!
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

0x20008000

r0 after store

	Memory Address	Memory Data
	0x20008007	0x00
	0x20008006	0x00
	0x20008005	0x00
	0x20008004	0x00
	0x20008003	0x00
	0x20008002	0x00
	0x20008001	0x00
r0 →	0x20008000	0x00

Example

```
STR r1, [r0, #4]!
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

0x20008000

r0 after store

0x20008004

	Memory Address	Memory Data
	0x20008007	0x76
	0x20008006	0x54
	0x20008005	0x32
r0 ⇨	0x20008004	0x10
	0x20008003	0x00
	0x20008002	0x00
	0x20008001	0x00
	0x20008000	0x00

Addressing Modes for Load/Store Multiple Registers

```
STMxx rn{!}, {register_list}
LDMxx rn{!}, {register_list}
```

- xx = IA, IB, DA, or DB

Addressing Modes	Description	Instructions
IA	Increment After	STMIA, LDMIA
IB	Increment Before	STMIB, LDMIB
DA	Decrement After	STMDA, LDMDA
DB	Decrement Before	STMDB, LDMDB

- **IA**: address is incremented by 4 after a word is loaded or stored.
- **IB**: address is incremented by 4 before a word is loaded or stored.
- **DA**: address is decremented by 4 after a word is loaded or stored.
- **DB**: address is decremented by 4 before a word is loaded or stored.

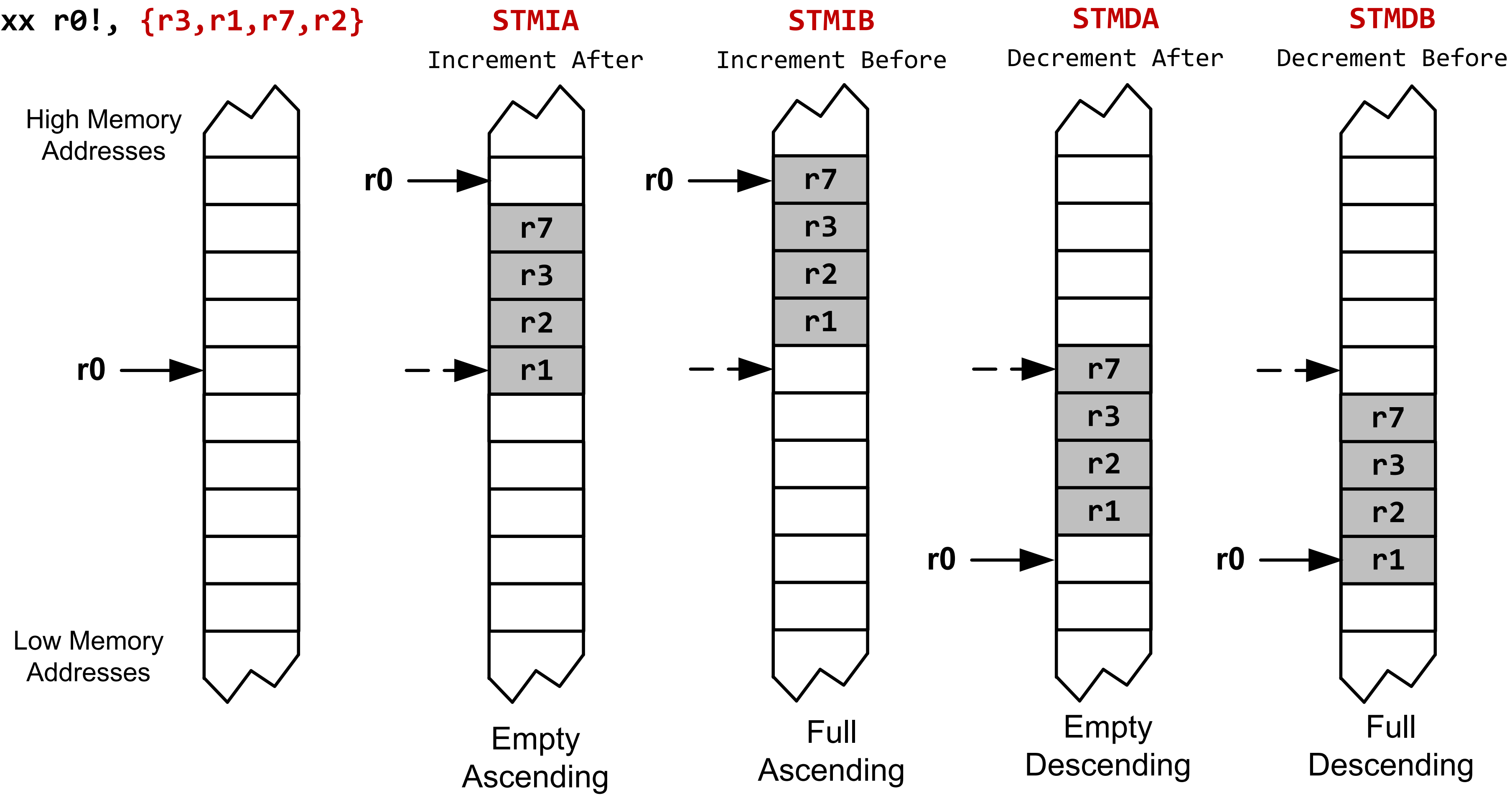
Load/Store Multiple Registers

- The following are synonyms.
  - `STM` = `STMIA` (Increment After) = `STMEA` (Empty Ascending)
  - `LDM` = `LDMIA` (Increment After) = `LDMFD` (Full Descending)

- The order in which registers are listed does not matter
  - For STM/LDM, the lowest-numbered register is stored/loaded at the lowest memory address.

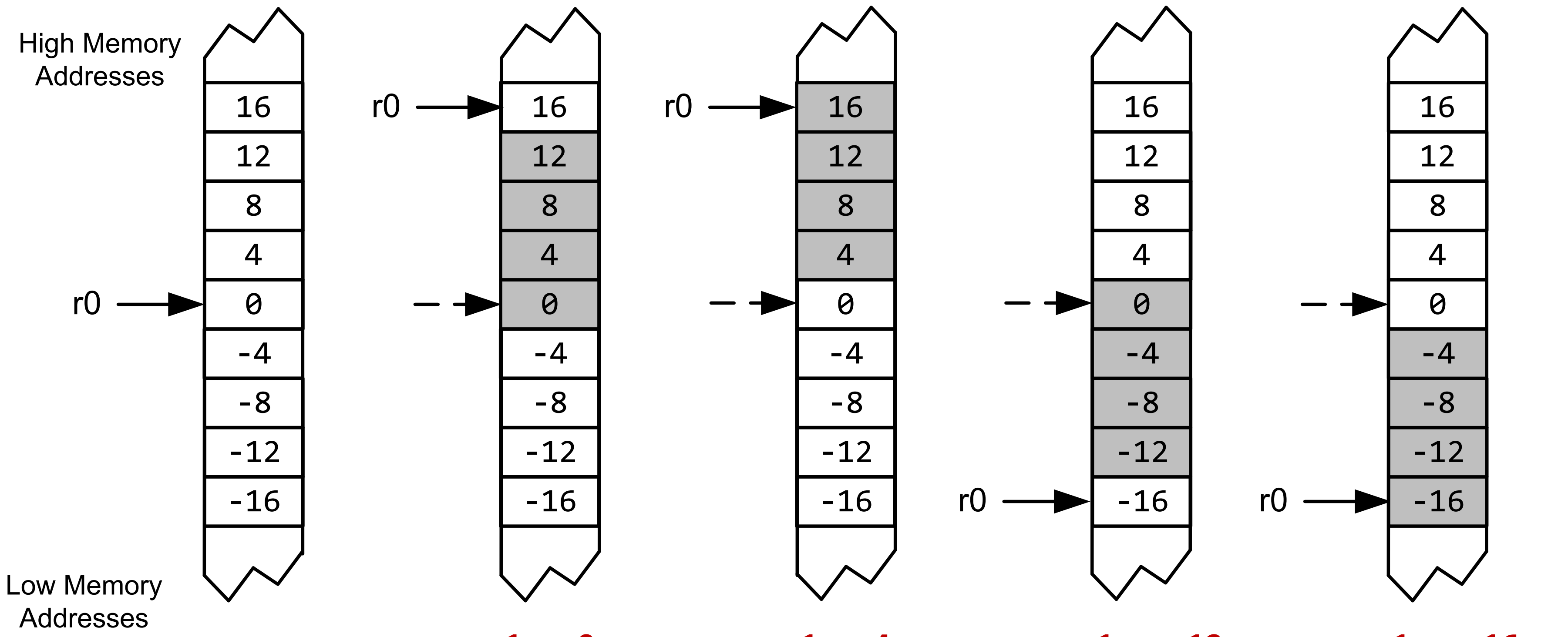
Store Multiple Registers

`STMxx r0!, {r3,r1,r7,r2}`



Load Multiple Registers

`LDMxx r0!, {r3,r1,r7,r2}`



LDMIA
Increment After

```
r1 = 0
r2 = 4
r3 = 8
r7 = 12
```

LDMIB
Increment Before

```
r1 = 4
r2 = 8
r3 = 12
r7 = 16
```

LDMDA
Decrement After

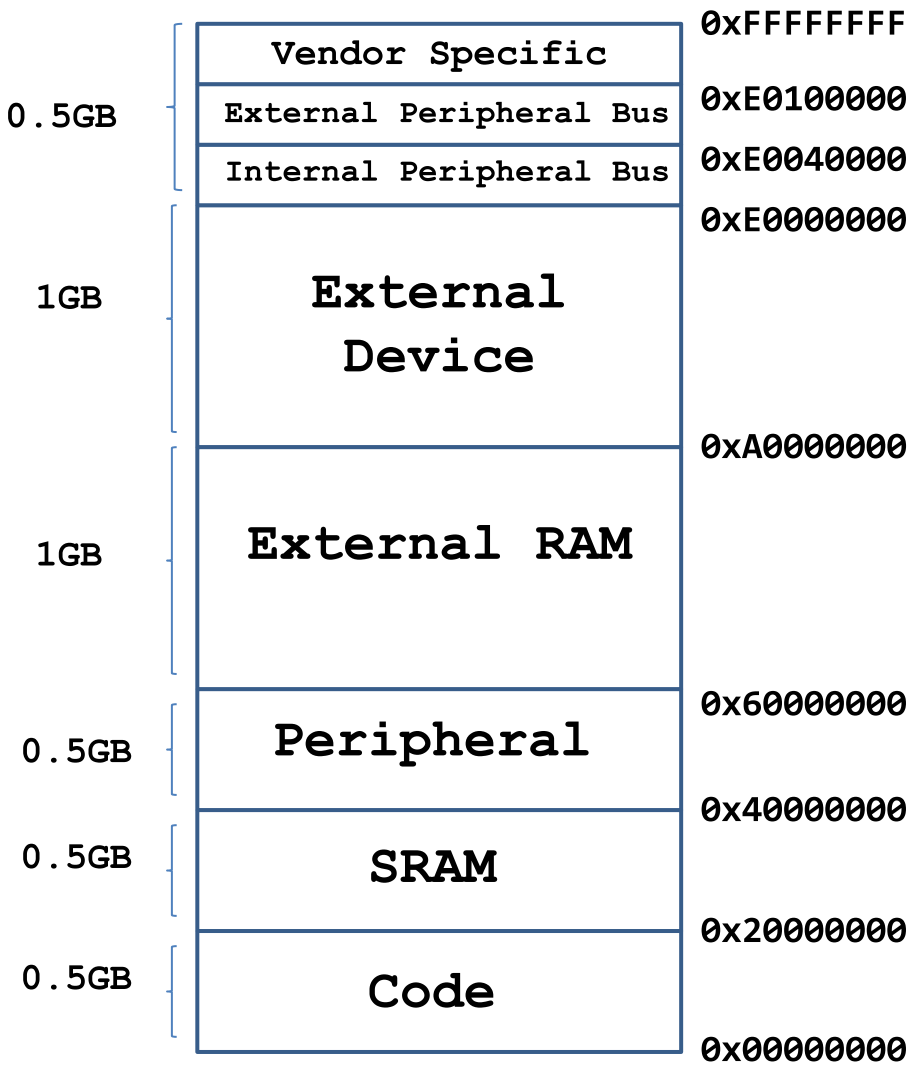
```
r1 = -12
r2 = -8
r3 = -4
r7 = -0
```

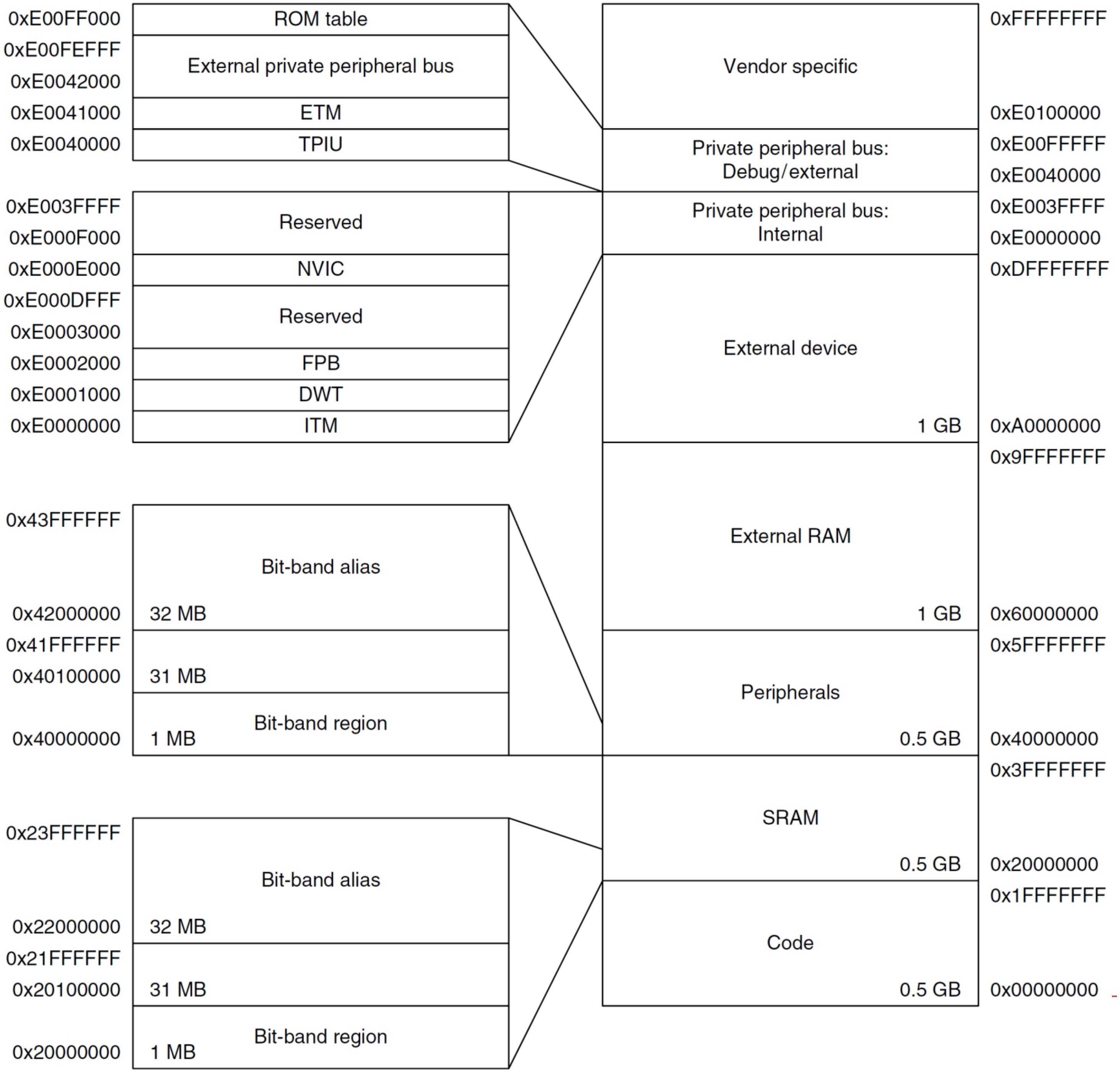
LDMDB
Decrement Before

```
r1 = -16
r2 = -12
r3 = -8
r7 = -4
```

Cortex-M3 & Cortex-M4 Memory Map

- 32-bit Memory Address
- 2^{32} bytes of memory space (4 GB)
- Harvard architecture: physically separated instruction memory and data memory





0xE00FF000 ROM table
0xE00FEFFF External private peripheral bus
0xE0042000
0xE0041000 ETM
0xE0040000 TPIU

0xE003FFFF Reserved
0xE000F000
0xE000E000 NVIC
0xE000DFFF Reserved
0xE0003000
0xE0002000 FPB
0xE0001000 DWT
0xE0000000 ITM

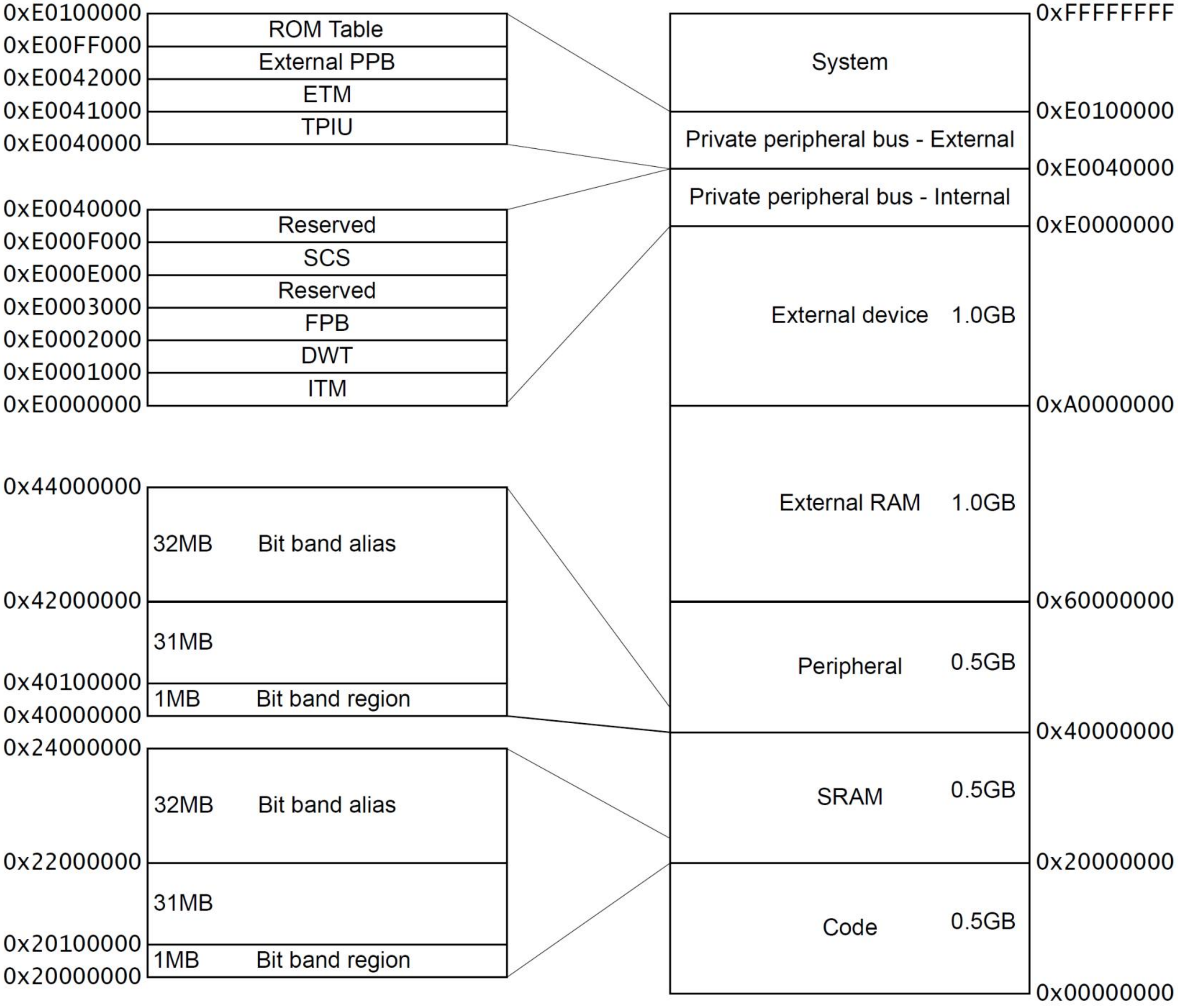
0x43FFFFFF Bit-band alias
0x42000000 32 MB
0x41FFFFFF
0x40100000 31 MB
Bit-band region
0x40000000 1 MB

0x23FFFFFF Bit-band alias
0x22000000 32 MB
0x21FFFFFF
0x20100000 31 MB
Bit-band region
0x20000000 1 MB

Vendor specific 0xFFFFFFFF
0xE0100000
Private peripheral bus: Debug/external 0xE00FFFFF
0xE0040000
Private peripheral bus: Internal 0xE003FFFF
0xE0000000
External device 0xDFFFFFFF
1 GB 0xA0000000
External RAM 0x9FFFFFFF
1 GB 0x60000000
Peripherals 0x5FFFFFFF
0.5 GB 0x40000000
SRAM 0x3FFFFFFF
0.5 GB 0x20000000
Code 0x1FFFFFFF
0.5 GB 0x00000000

Cortex-M3 Fixed Memory Map

Cortex-M4 Fixed Memory Map



Pseudo-instructions

- Pseudo instruction: available to use in an assembly program, but not directly supported by hardware.
- Pseudo → not real
- Compilers translate it to one or multiple actual machine instructions
- Pseudo instructions are provided for the convenience of programmers.

LDR Pseudo-instruction

LDR Rt, =expr

LDR Rt, =label

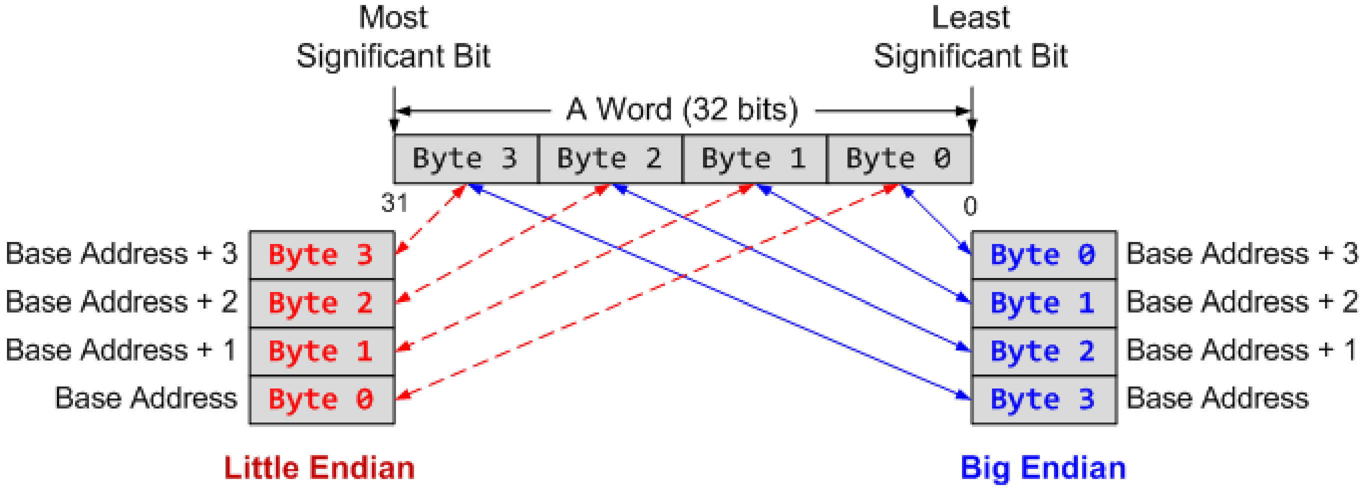
- If the value of expr can be loaded with **MOV**, **MVN** (16-bit instruction) or **MOVW** (32-bit instruction), the assembler uses that instruction.
  - MOV supports all 8-bit immediate numbers ranging in [0 – 255]. For numbers out of this range, some patterns can be encoded.
- If a valid MOV, MVN, MOVW instruction cannot be used (due to out of range), or if the label_expr syntax is used, the assembler places the constant in a literal pool and generates a PC-relative LDR instruction that reads the constant from the literal pool.

```
LDR r1,=0xFF0   ; loads 0xFF0 into R1
                ; =>  MOV r1,#0xFF0
LDR r2,=0xFFF   ; loads 0xFFF into R2
                ; =>  MOVW r2, #0xFFF
LDR r3,=array   ; loads the address of array into R3
                ; =>  LDR r3,[pc, offset_to_litpool]
                ;     ...
                ;     litpool DCD array
```

Software uses this pseudo instruction to set a register to some value without worrying about the size of the value.

Summary

- Memory address is always in terms of bytes.
- How data is organized in memory?



- How data is addressed?

Addressing Format	Example	Equivalent
Pre-index	`LDR r1, [r0, #4]`	r1 ← memory[r0 + 4], r0 is unchanged

Pre-index with update	`LDR r1, [r0, #4]!`	r1 ← memory[r0 + 4]
		r0 ← r0 + 4

Post-Index	`LDR r1, [r0], #4`	r1 ← memory[r0]
		r0 ← r0 + 4

References

- Lecture 22. Big-Endian and Little-Endian
  - https://www.youtube.com/watch?v=T1C9Kj_78ek&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=22
- Lecture 23. Load and Store Instructions
  - https://www.youtube.com/watch?v=CtfV3HsHwk4&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=23
- Lecture 24. Addressing mode: pre-index, post-index, and pre-index with update
  - https://www.youtube.com/watch?v=zgkxPdPkxa8&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=24
- ARM Instruction Set - Stack Instructions STMFD, STMFA , STMED, STMEA, Vishal Gaikwad
  - https://www.youtube.com/watch?v=H4xoaOINSJo