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

Chapter 5
Memory Access

Zonghua 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/

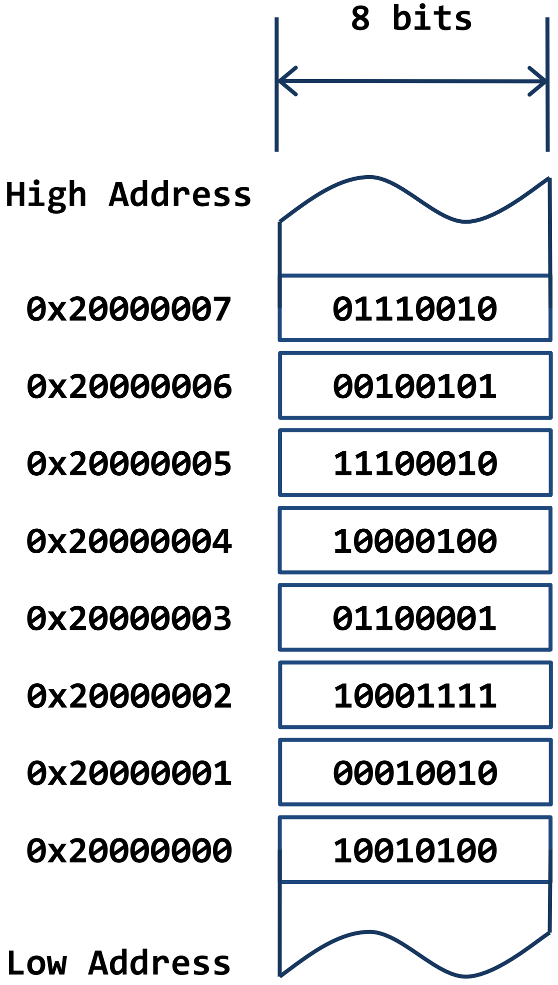
Overview

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

- How is data 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, we can store more values
  - 8 bits = **1 byte**
  - 16 bits = 2 bytes = **1 halfword**
  - 32 bits = 4 bytes = **1 word**



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

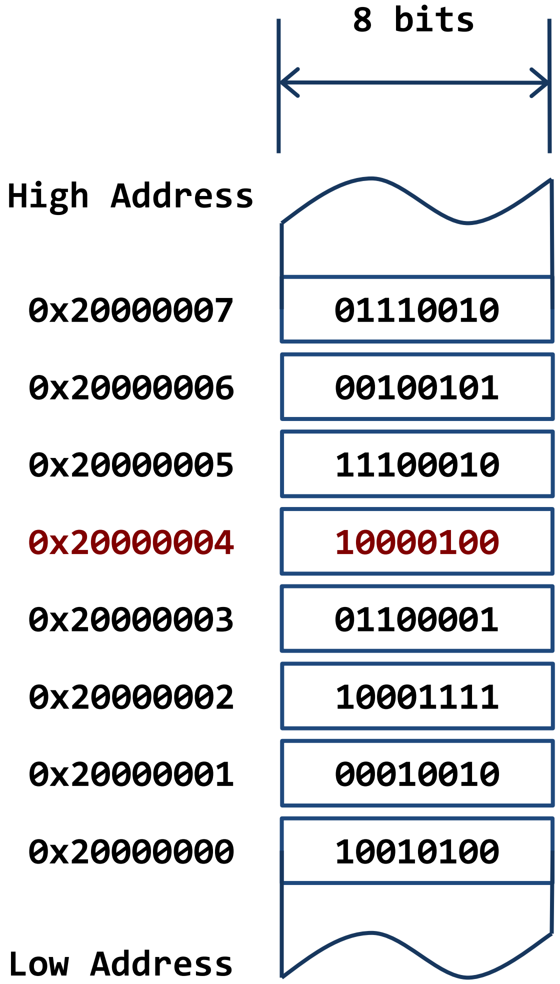
Logic View of Memory

- By grouping bits, we can store more values
  - 8 bits = 1 **byte**
  - 16 bits = 2 bytes = 1 **halfword**
  - 32 bits = 4 bytes = 1 **word**
- From the 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

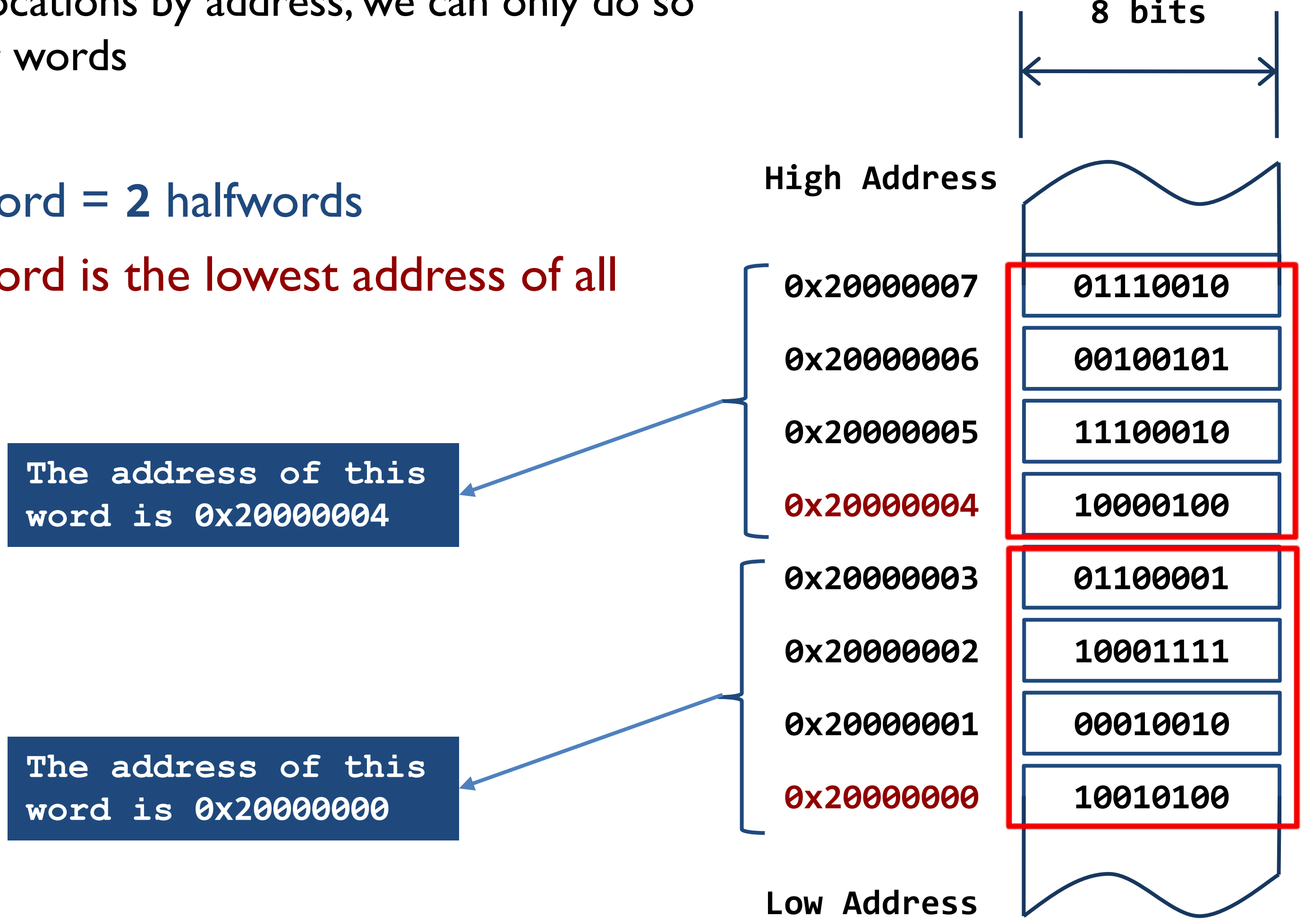
Computer memory is *byte-addressable*!



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
  - Memory address of a word is the lowest address of all four bytes in that word.



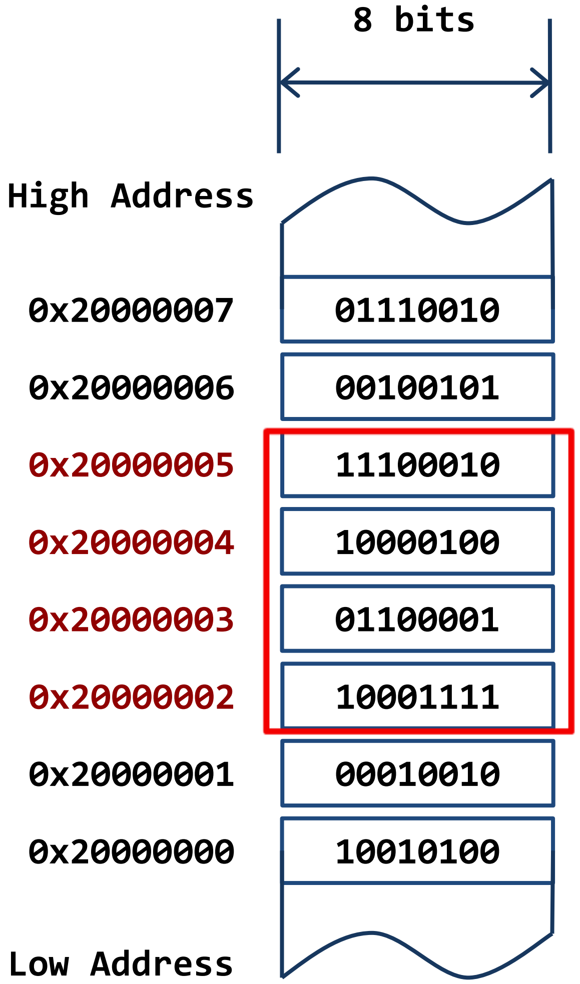
The address of this word is 0x20000004

The address of this word is 0x20000000

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

Logic View of Memory

- Can you store a word anywhere?



Logic View of Memory

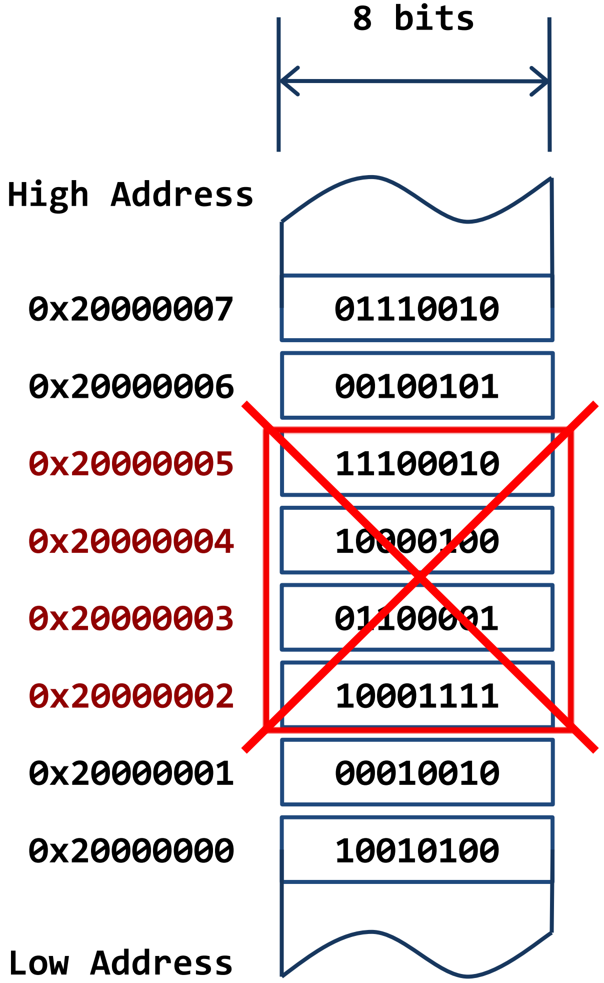
- Can we store a word anywhere in memory?

  NO on most computers!

- A word can only be stored at an address that's divisible by 4.

Word-address mod 4 = 0

We cannot store a word at address 0x20000002.



8 bits

High Address

0x20000007 01110010

0x20000006 00100101

0x20000005 11100010

0x20000004 10000100

0x20000003 01100001

0x20000002 10001111

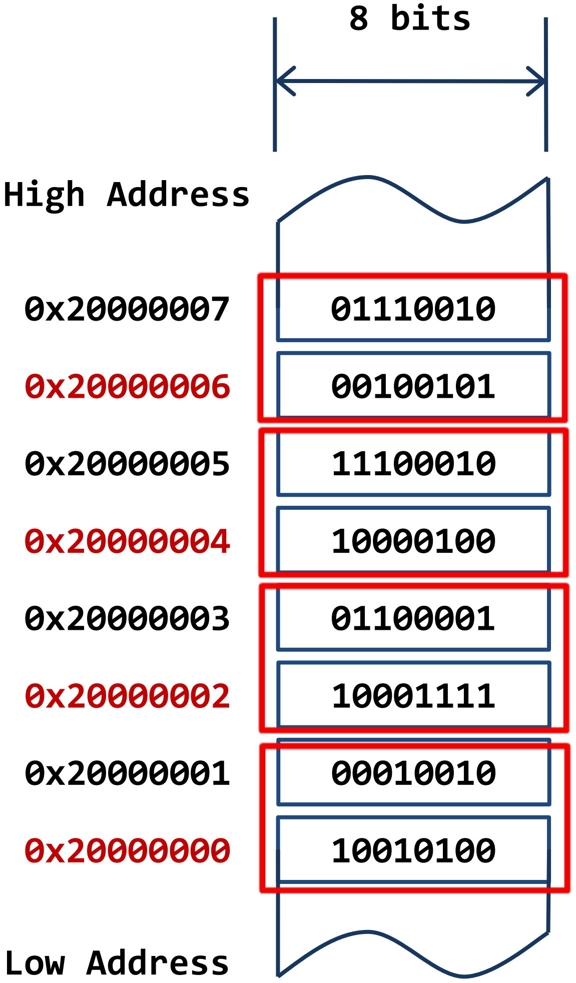
0x20000001 00010010

0x20000000 10010100

Low Address

Logic View of Memory

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



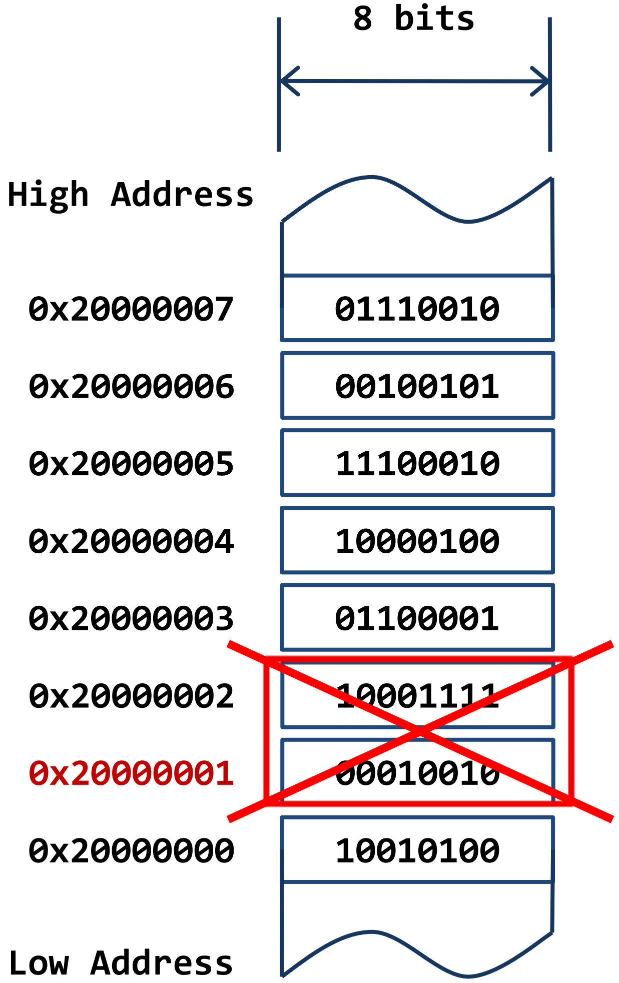
0x20000007	01110010
0x20000006	00100101
0x20000005	11100010
0x20000004	10000100
0x20000003	01100001
0x20000002	10001111
0x20000001	00010010
0x20000000	10010100

Logic View of Memory

- Can you store a halfword anywhere? **NO.**
- A halfword can only be stored at an address that's divisible by 2.
- Memory address of a halfword is the lowest address of its two bytes.

Halfword-address mod 2 = 0

We cannot store a halfword at address 0x20000001.

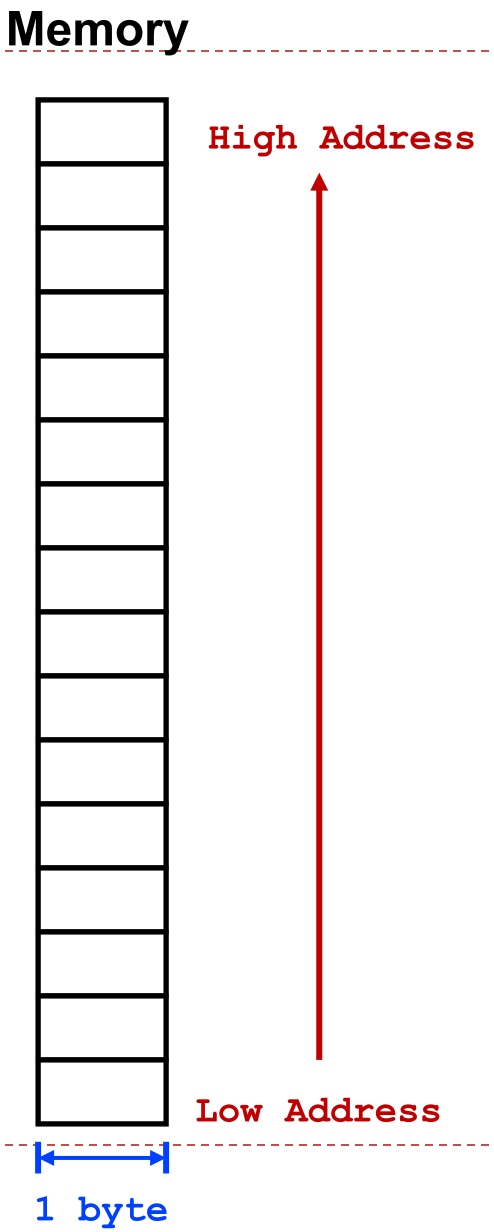


0x20000007	01110010
0x20000006	00100101
0x20000005	11100010
0x20000004	10000100
0x20000003	01100001
0x20000002	10001111
0x20000001	00010010
0x20000000	10010100

Quiz

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

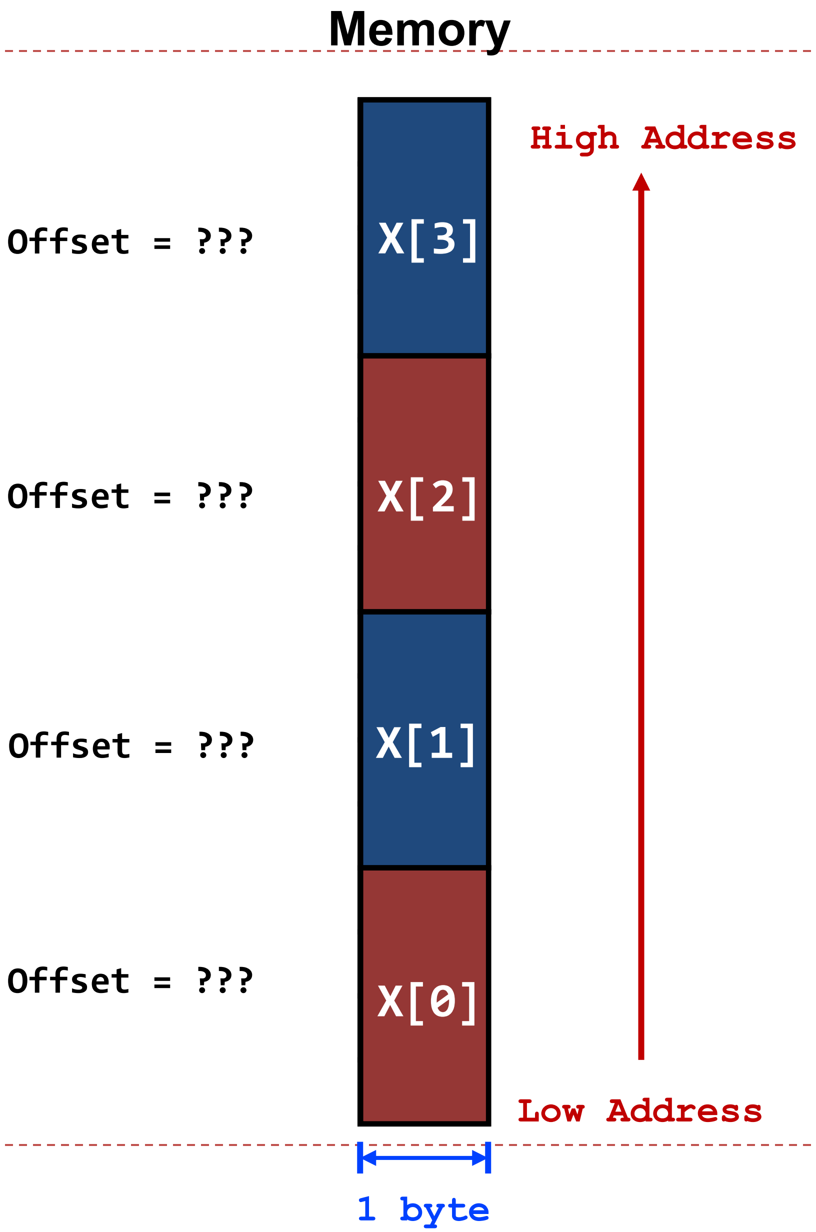
What are their memory address offsets?



Quiz

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

What are their memory address offsets?



Memory

Offset = ???

Offset = ???

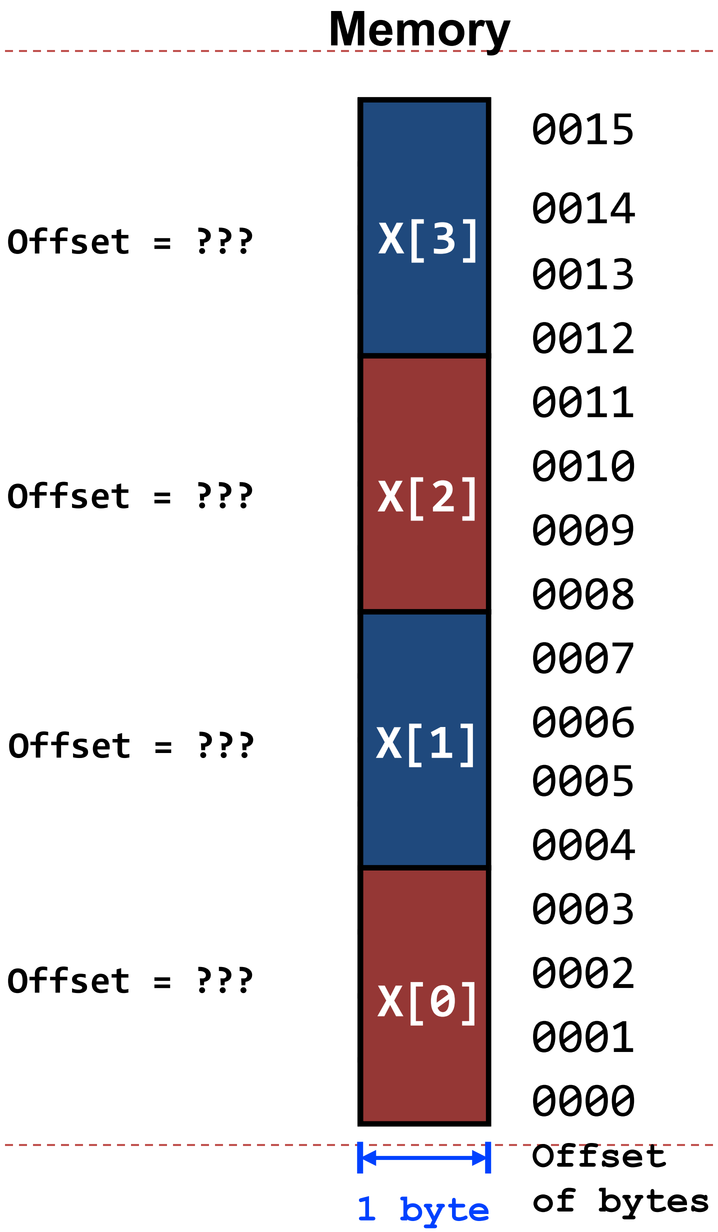
Offset = ???

Offset = ???

Quiz

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

What are their memory address offsets?



Quiz

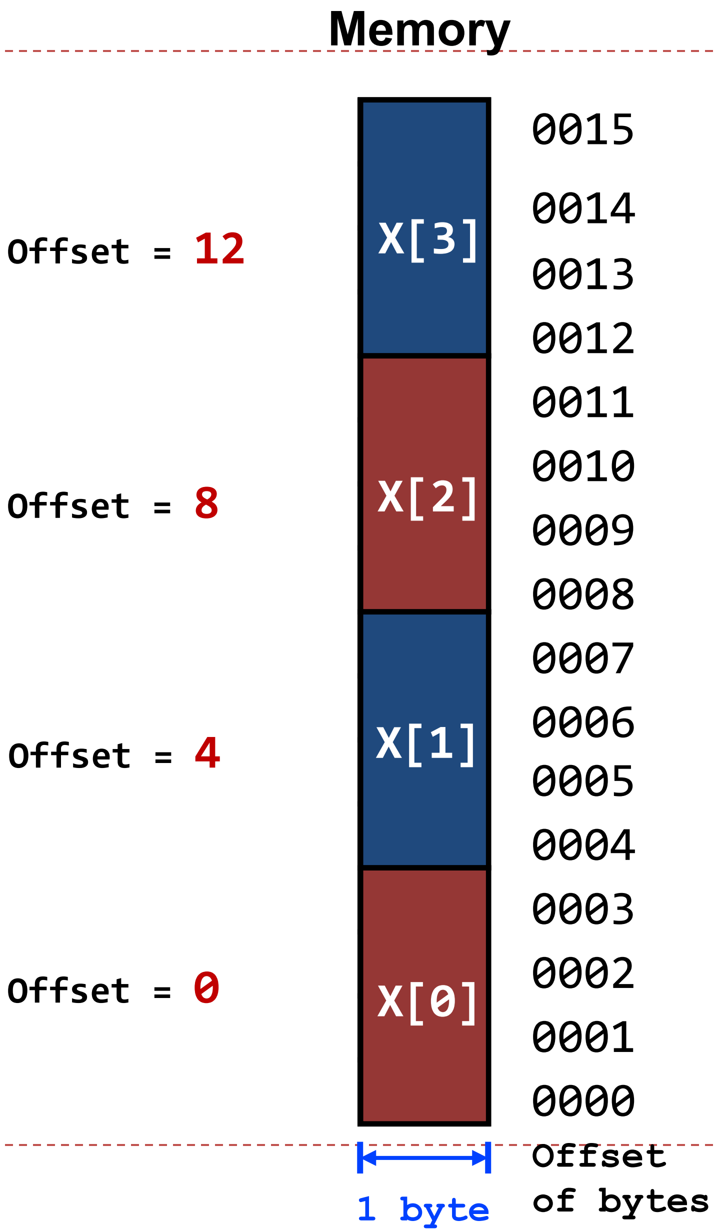
```
uint32_t X[4];
```

What are their memory address offsets?

If the array starts at address pAddr,

- Memory address of X[0] is pAddr
- Memory address of X[1] is pAddr + 4
- Memory address of X[2] is pAddr + 8
- Memory address of X[3] is pAddr + 12

Sequential words are at addresses incremented by 4, not by 1!



Memory

Offset = 12

Offset = 8

Offset = 4

Offset = 0

X[3]

X[2]

X[1]

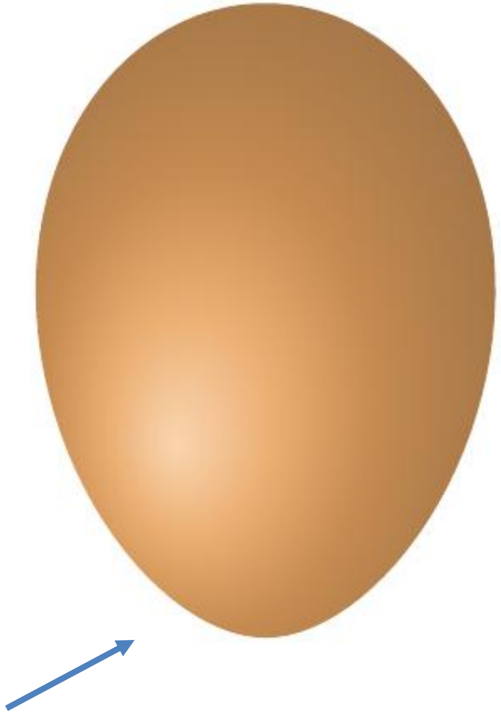
X[0]

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

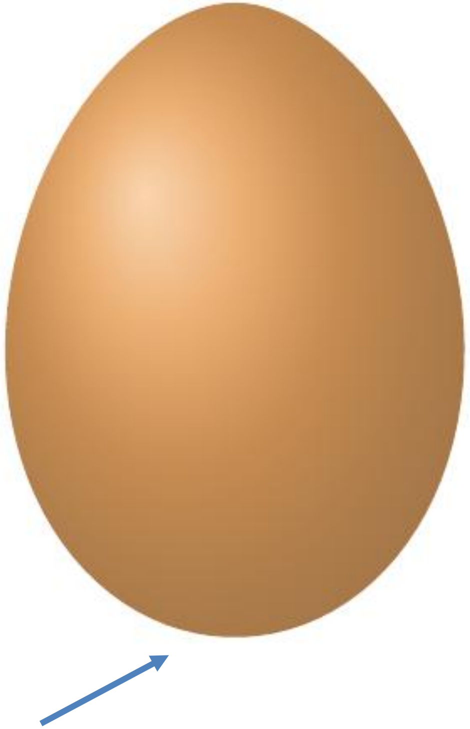
1 byte

Offset of bytes

Which end do you break to eat a boiled egg?

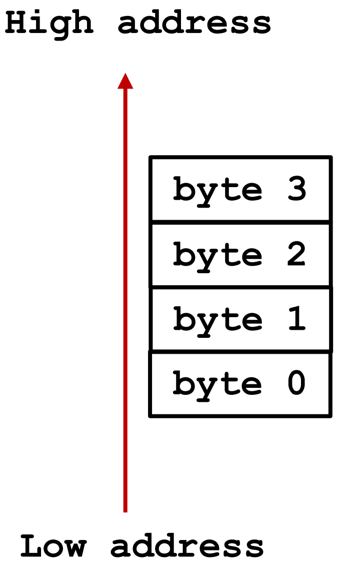


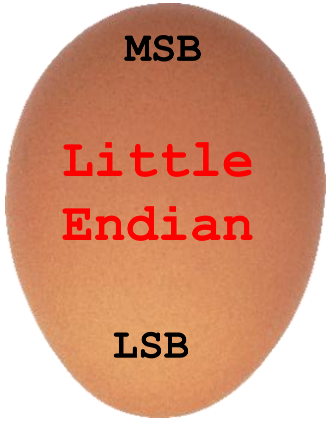
Little Endian



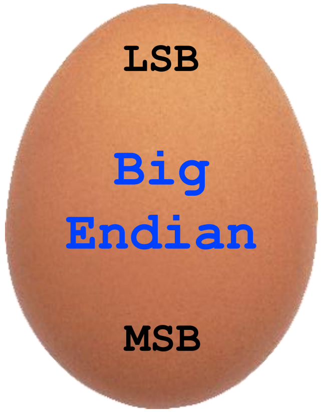
Big Endian

Endianess





LSB is at
least address



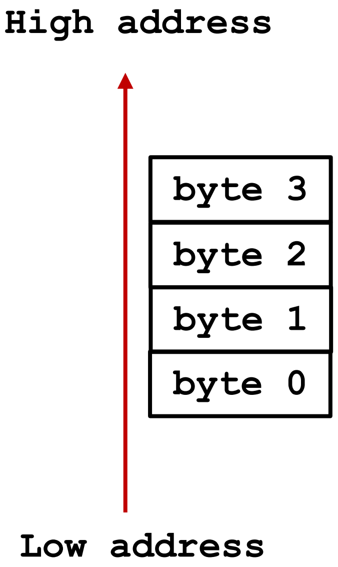
MSB is at
least address

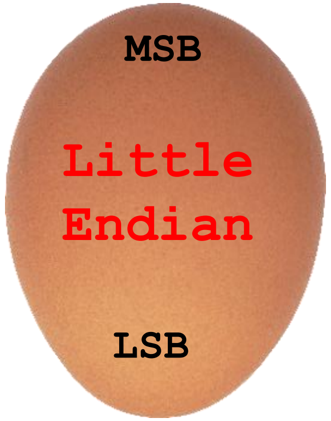
Gulliver's Travels (by Jonathan Swift, published in 1726):

- **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**

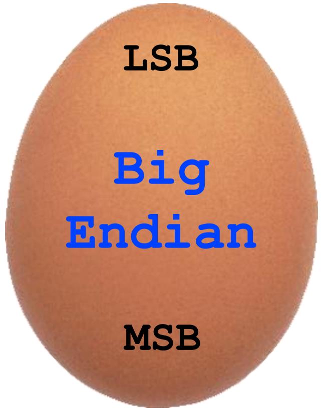
Endianess

Endian: byte order, not bit order!



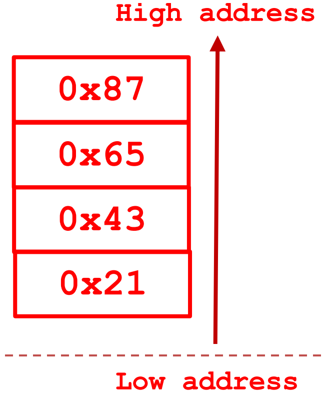


LSB is at
least address



MSB is at
least address

Little-Endian



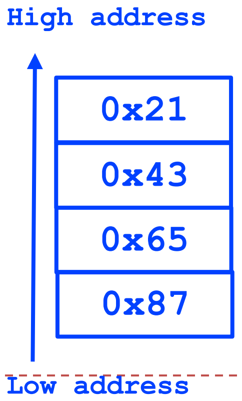
```
uint32_t a = 0x87654321
```

Reading from the top

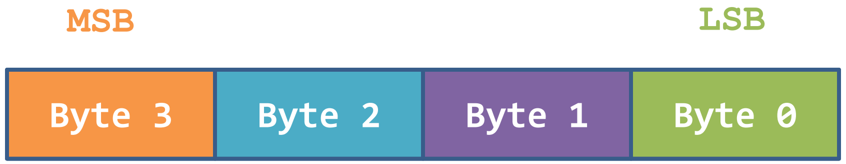
byte 3	byte 2	byte 1	byte 0
0x87	0x65	0x43	0x21
byte 0	byte 1	byte 2	byte 3

Reading from the bottom

Big-Endian



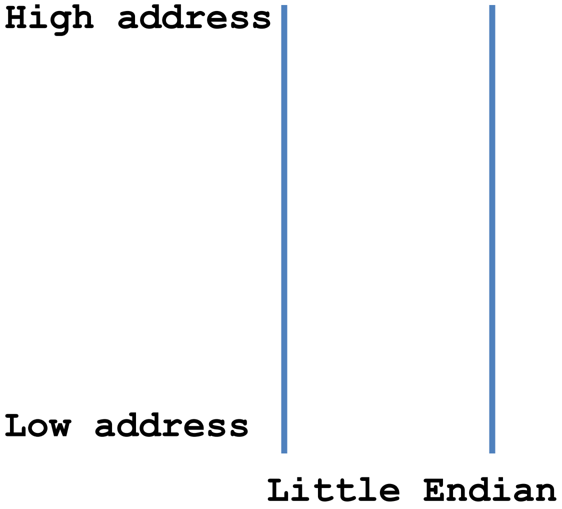
Little Endian *vs* Big Endian



MSB

LSB

Byte 3 Byte 2 Byte 1 Byte 0



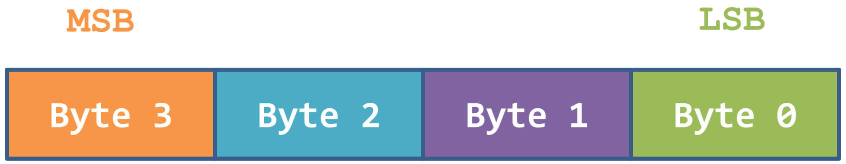
High address

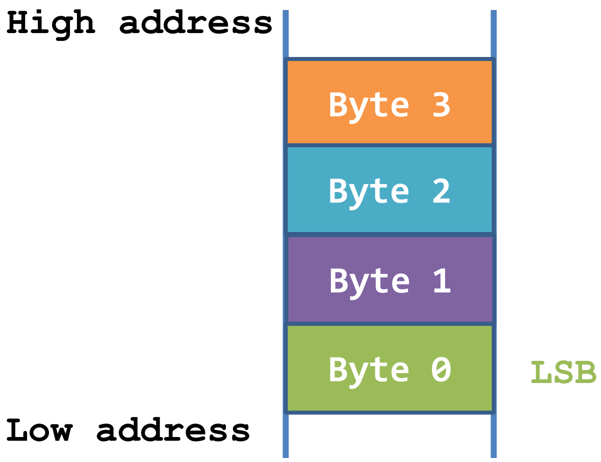
Low address

Little Endian

LSB is at least address!

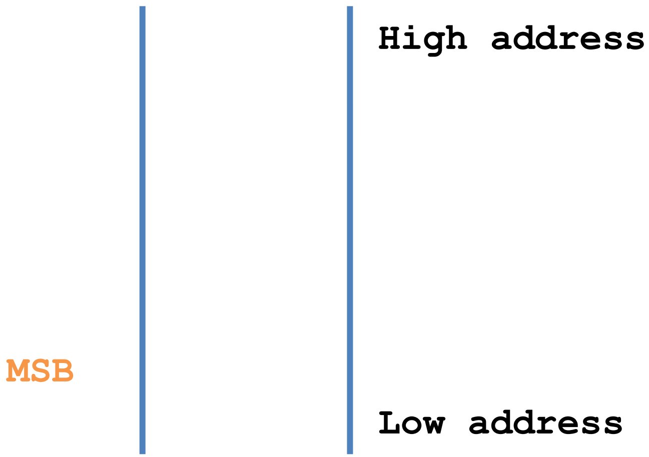
Little Endian *vs* Big Endian





Little Endian

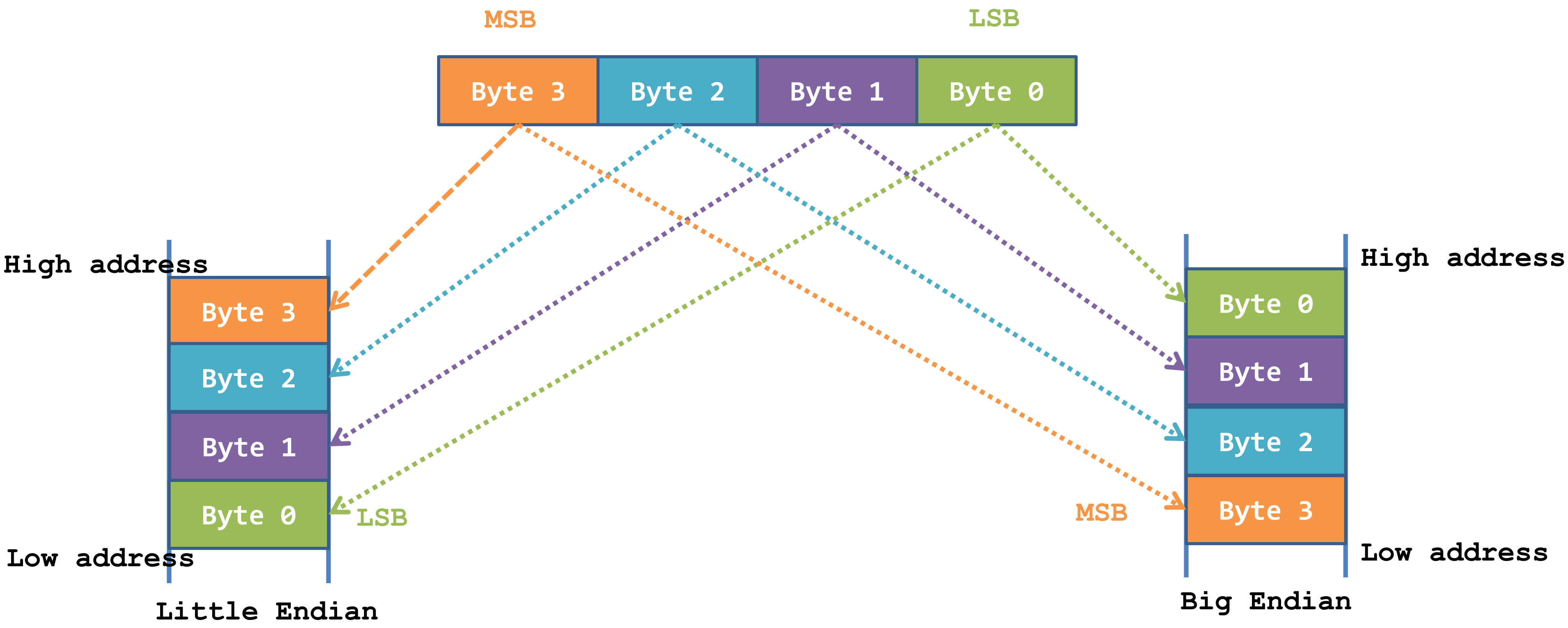
LSB is at least address!



Big Endian

MSB is at least address!

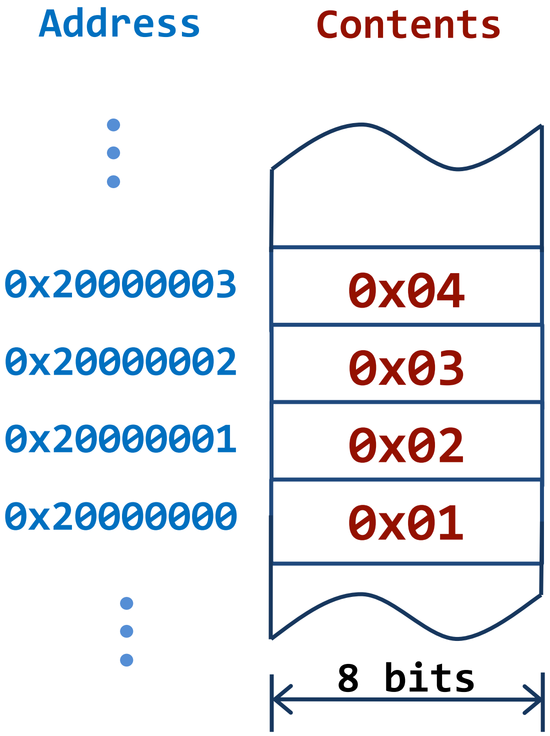
Little Endian *vs* Big Endian



LSB is at least address!

MSB is at least address!

Word stored at 0x20000000?



Address	Contents
0x20000003	0x04
0x20000002	0x03
0x20000001	0x02
0x20000000	0x01

- If Little Endian

  value = 0x04030201

- If Big Endian

  value = 0x01020304

Example

If big endianess is used

The word stored at address 0x20008000 is

0xEE8C90A7

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

Example

If little endianess is used

The word stored at address 0x20008000 is

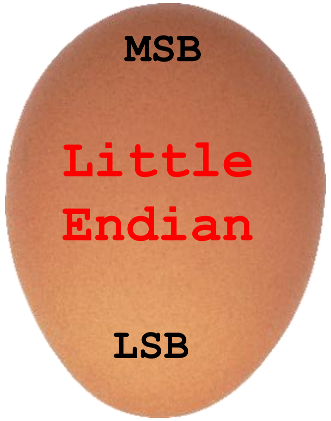
0xA7908CEE

Endian only specifies byte order, not bit order in a byte!

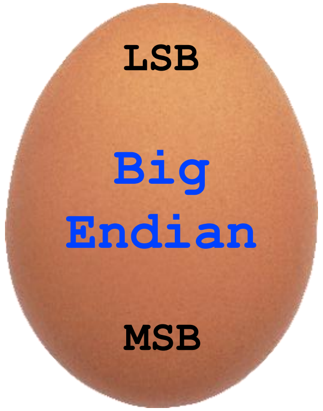
Memory Address	Memory Data
0x20008003	0xA7
0x20008002	0x90
0x20008001	0x8C
0x20008000	0xEE

Endian on Modern Architecture

- Intel x86 and AMD64/x86-64 use little endian.
- Atmel AVR32 and OpenRISC use big endian.
- Arm Cortex-M supports both Little Endian and Big Endian. However, endian maybe fixed for specific chips.
  - ST's L4 Series, TI's Tiva C, and NXP's K64 only supports only Little Endian.

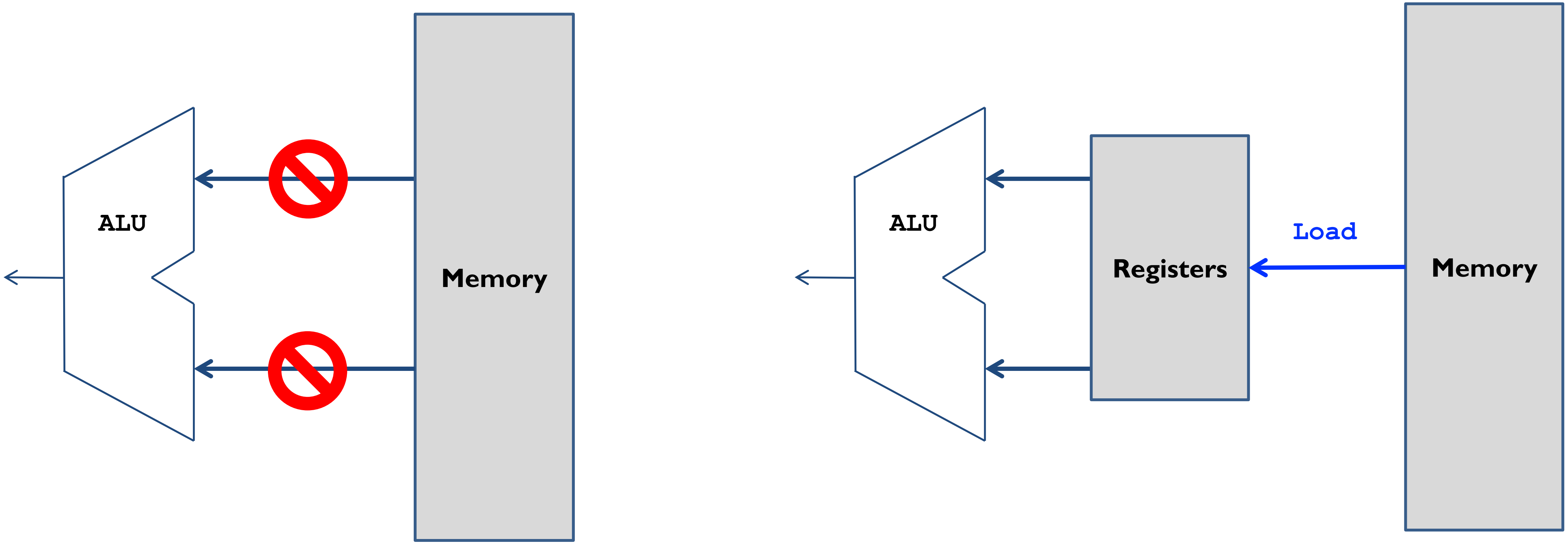


LSB is at least address

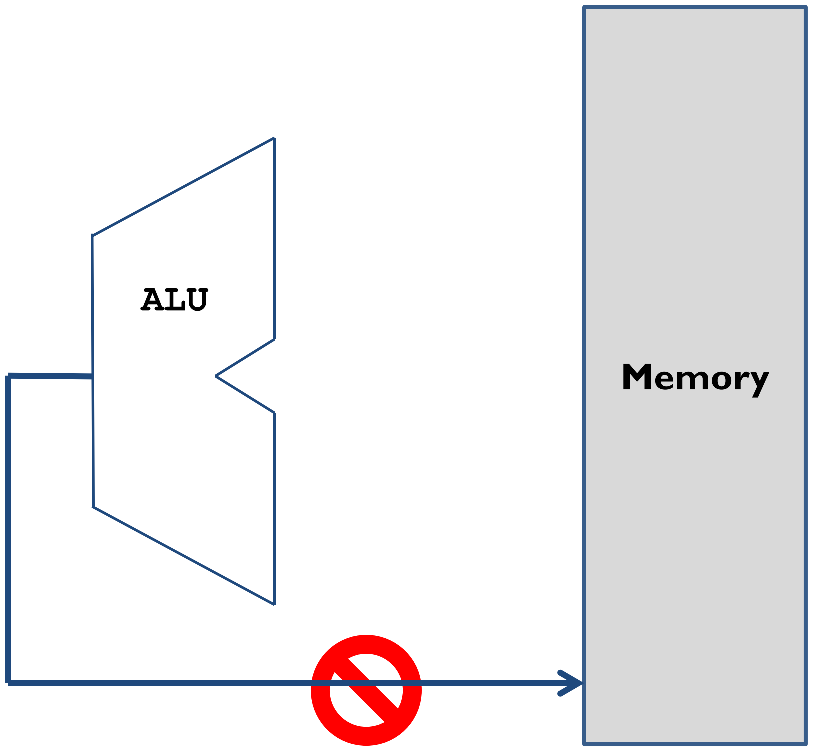


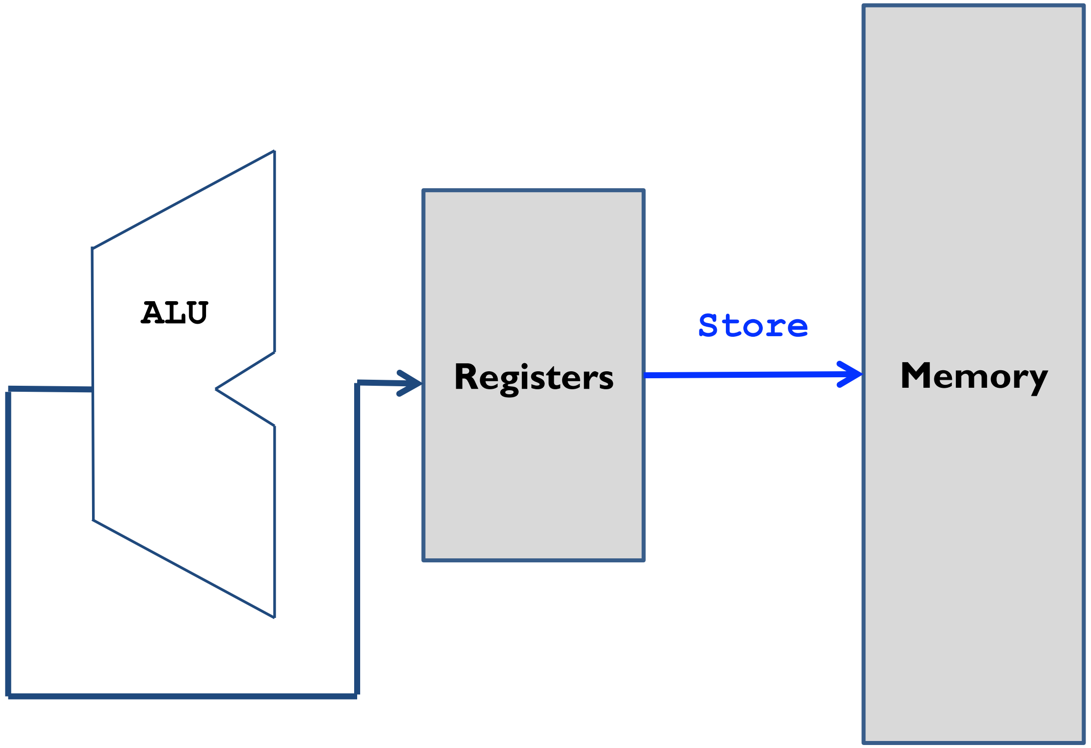
MSB is at least address

Loading Data from Memory



Storing Data to Memory





Load-Modify-Store

C statement

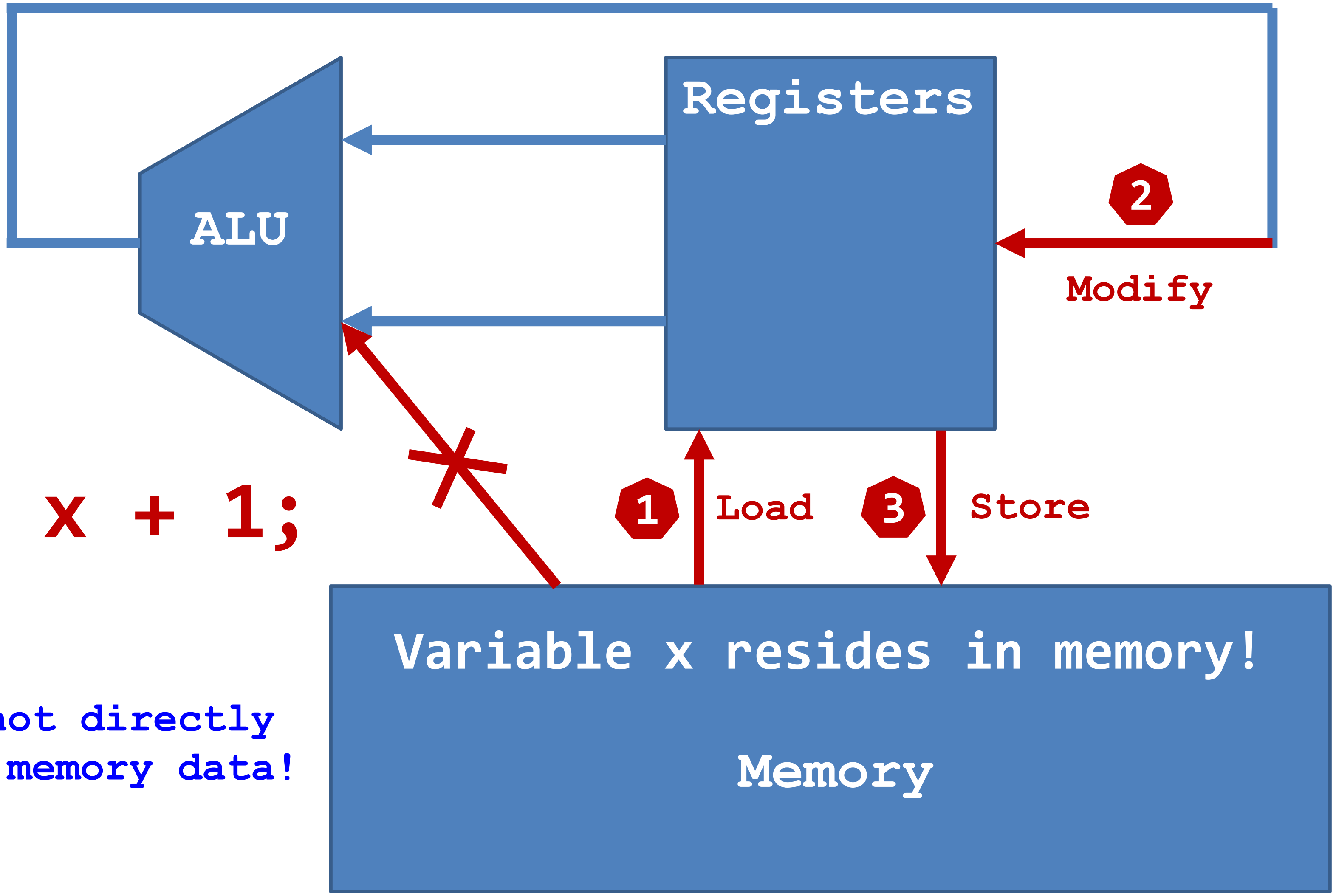
```
x = x + 1;
```

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



Registers

ALU

2 Modify

1 Load

3 Store

Variable x resides in memory!

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]
```

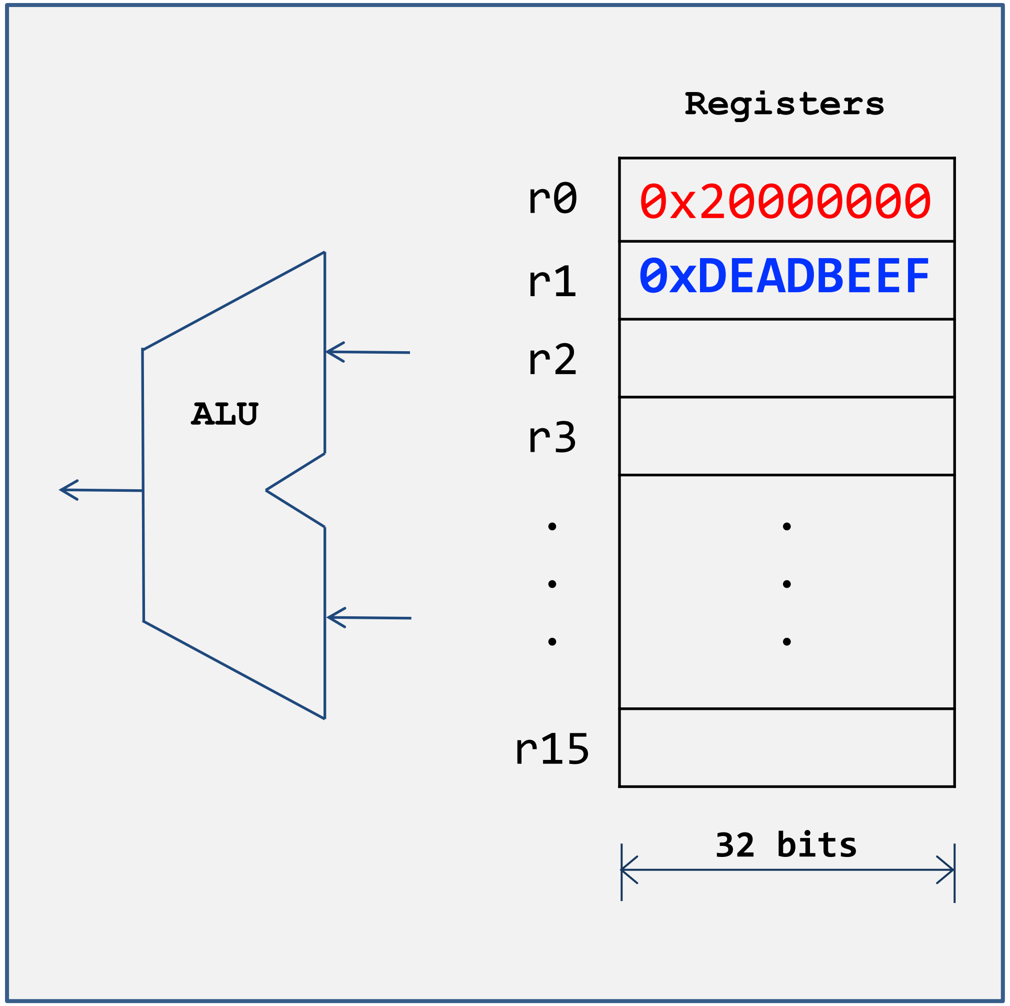
Store Instructions

- **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
```

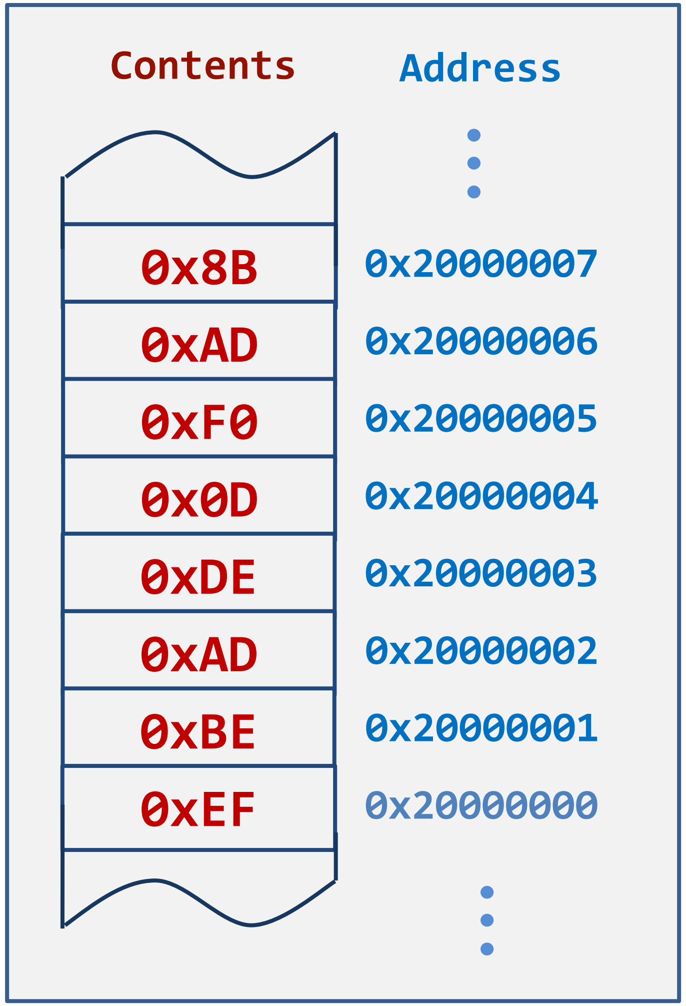
Loading Word from Memory

```
LDR r1, [r0] ; r1 = memory.word[r0]
             ; LDR stands for Load to Register
```



r0	0x20000000
r1	0xDEADBEEF
r2	
r3	
. . .	. . .
r15	

Processor Core

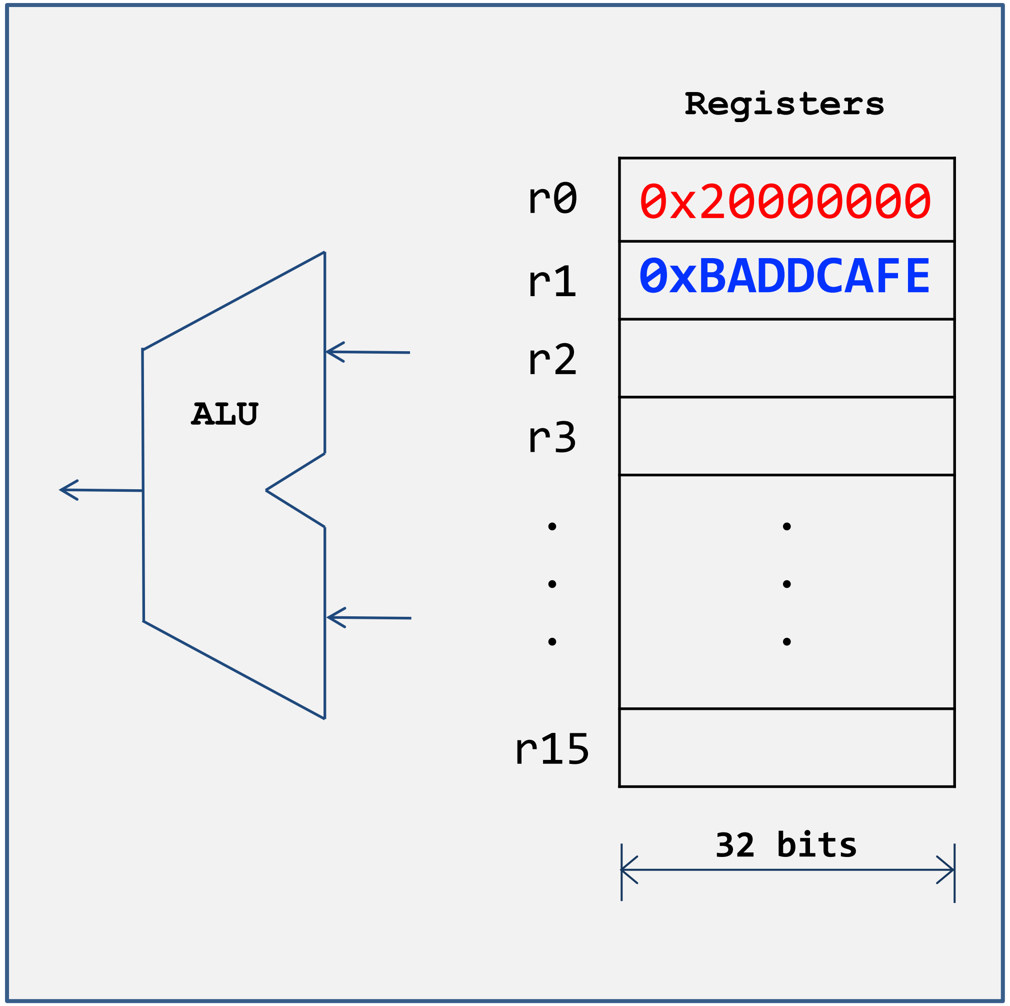


Contents	Address
	⋮
0x8B	0x20000007
0xAD	0x20000006
0xF0	0x20000005
0x0D	0x20000004
0xDE	0x20000003
0xAD	0x20000002
0xBE	0x20000001
0xEF	0x20000000
	⋮

Memory

Storing Word to Memory

```
STR r1, [r0] ; memory.word[r0] = r1
; STR stands for Store Register
```

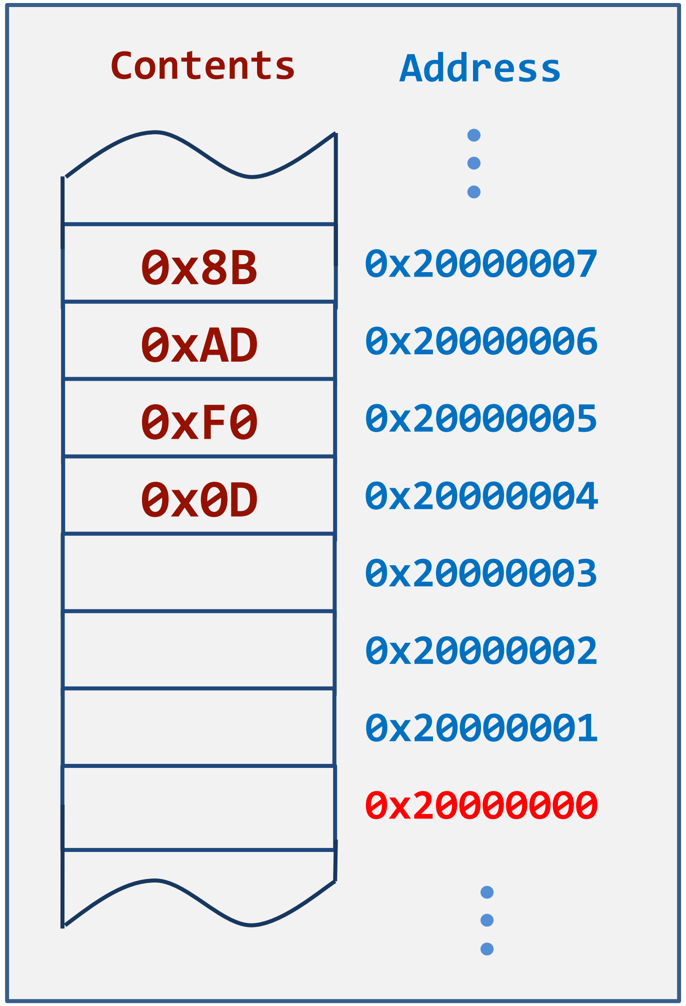


Registers

r0	0x20000000
r1	0xBADDCAFE
r2	
r3	
. . .	. . .
r15	

32 bits

Processor Core

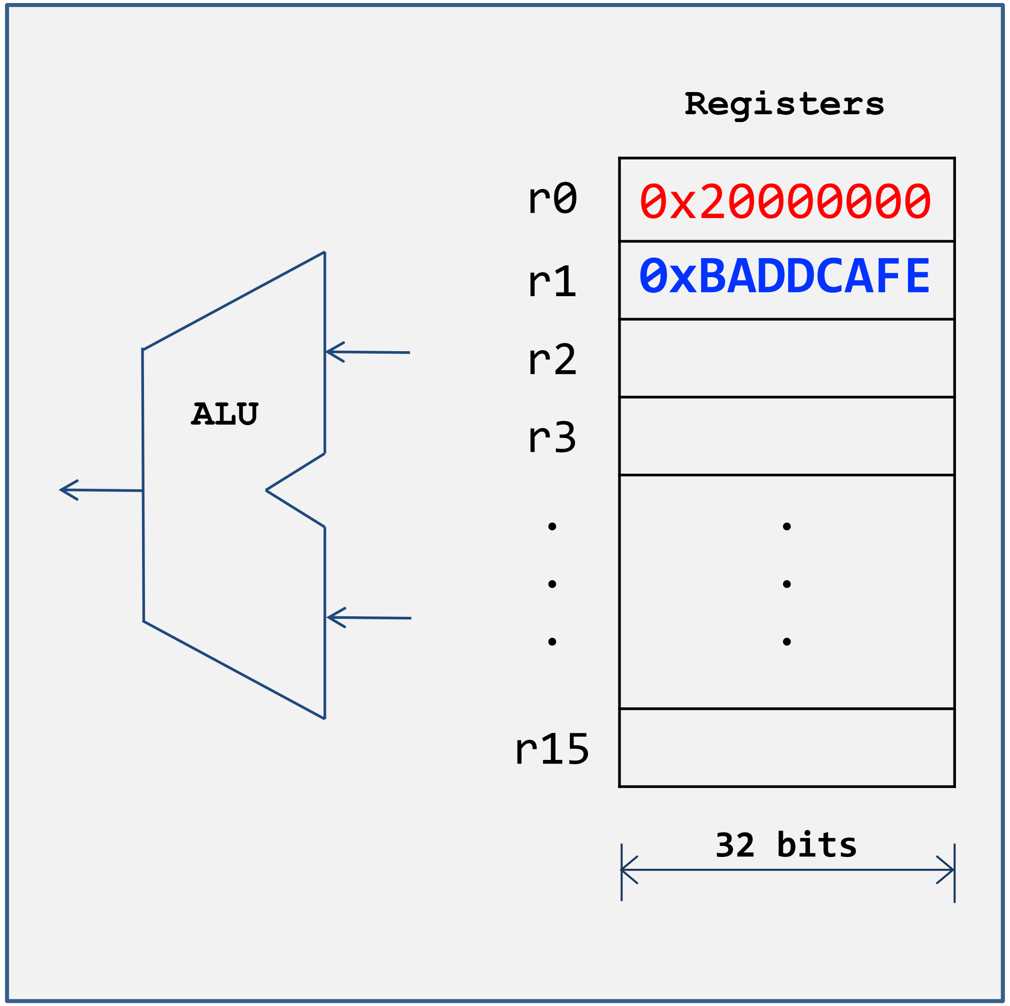


Contents	Address
	⋮
0x8B	0x20000007
0xAD	0x20000006
0xF0	0x20000005
0x0D	0x20000004
	0x20000003
	0x20000002
	0x20000001
	0x20000000
	⋮

Memory

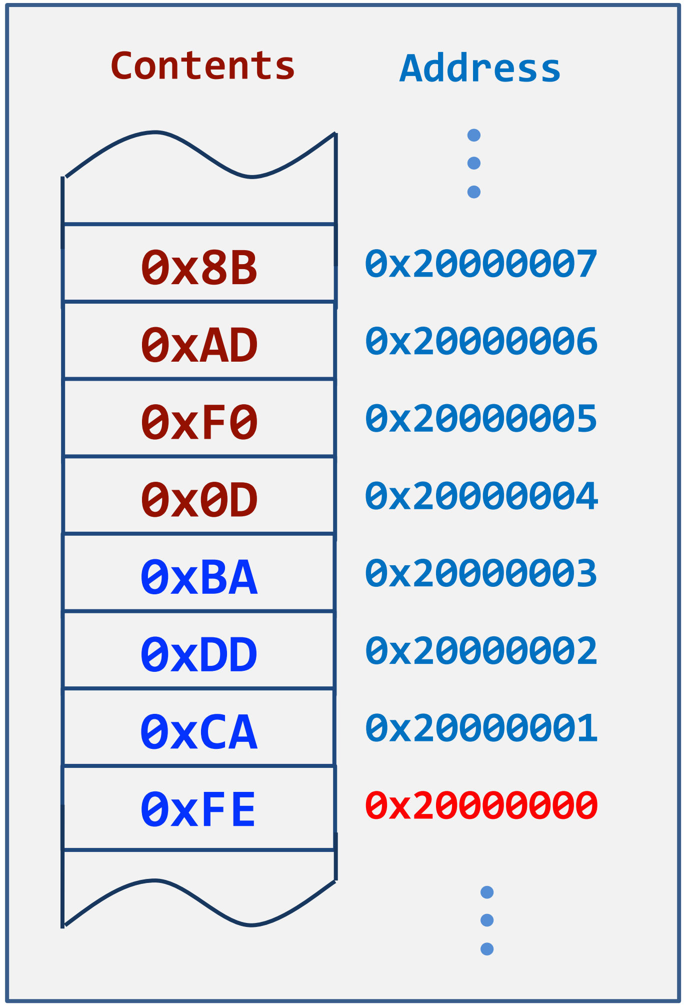
Storing Word to Memory

```
STR r1, [r0] ; memory.word[r0] = r1
; STR stands for Store Register
```



	Registers
r0	0x20000000
r1	0xBADDCAFE
r2	
r3	
. . .	. . .
r15	

Processor Core



Contents	Address
	⋮
0x8B	0x20000007
0xAD	0x20000006
0xF0	0x20000005
0x0D	0x20000004
0xBA	0x20000003
0xDD	0x20000002
0xCA	0x20000001
0xFE	0x20000000
	⋮

Memory

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 **B**yte	uint8_t	unsigned char
LDRH	Load **H**alfword	uint16_t	unsigned short int
LDRSB	Load **S**igned **B**yte	int8_t	signed char
LDRSH	Load **S**igned **H**alfword	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 **B**yte	uint8_t/int8_t	unsigned or signed char
STRH	Store Lower **H**alfword	uint16_t/int16_t	unsigned or signed short

Load a Byte, Half-word, Word

Load a Byte

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

0x00	0x00	0x00	0xE1
31			0

Load a Halfword

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

0x00	0x00	0xE3	0xE1
31			0

Load a Word

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

0x87	0x65	0xE3	0xE1
31			0

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

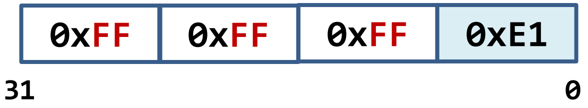
Little Endian

Assume
r0 = 0x20000000

Sign Extension

Load a Signed Byte

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

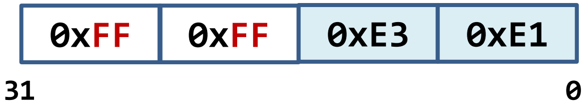


0xFF 0xFF 0xFF 0xE1

31 0

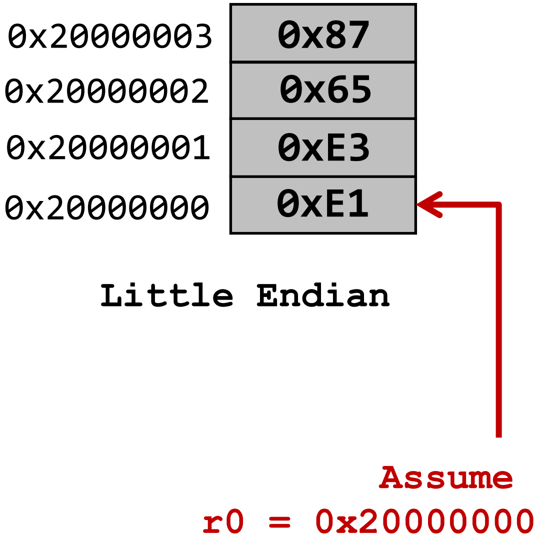
Load a Signed Halfword

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



0xFF 0xFF 0xE3 0xE1

31 0



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

Little Endian

Assume
r0 = 0x20000000

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, #4]
```

- Post-index

```
    LDR r1, [r0], #4
```

- Pre-index with Update

```
    LDR r1, [r0, #4]!
```

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

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

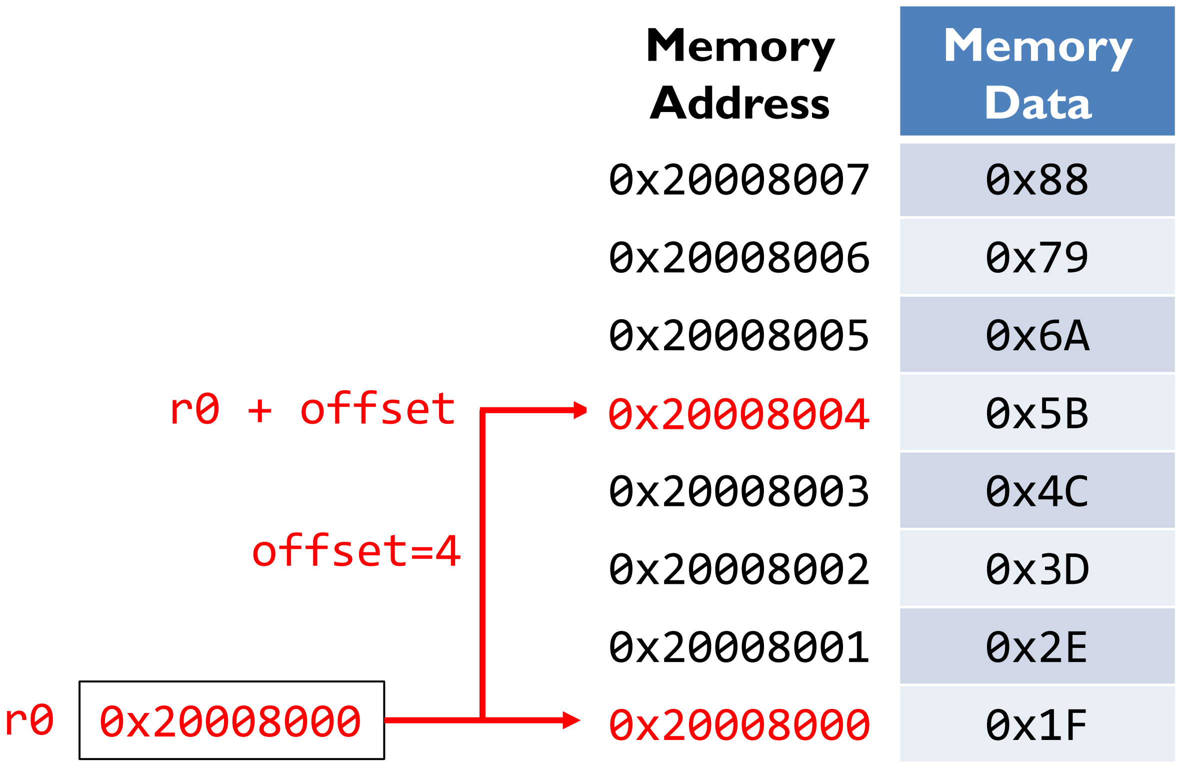
r0 0x20008000

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

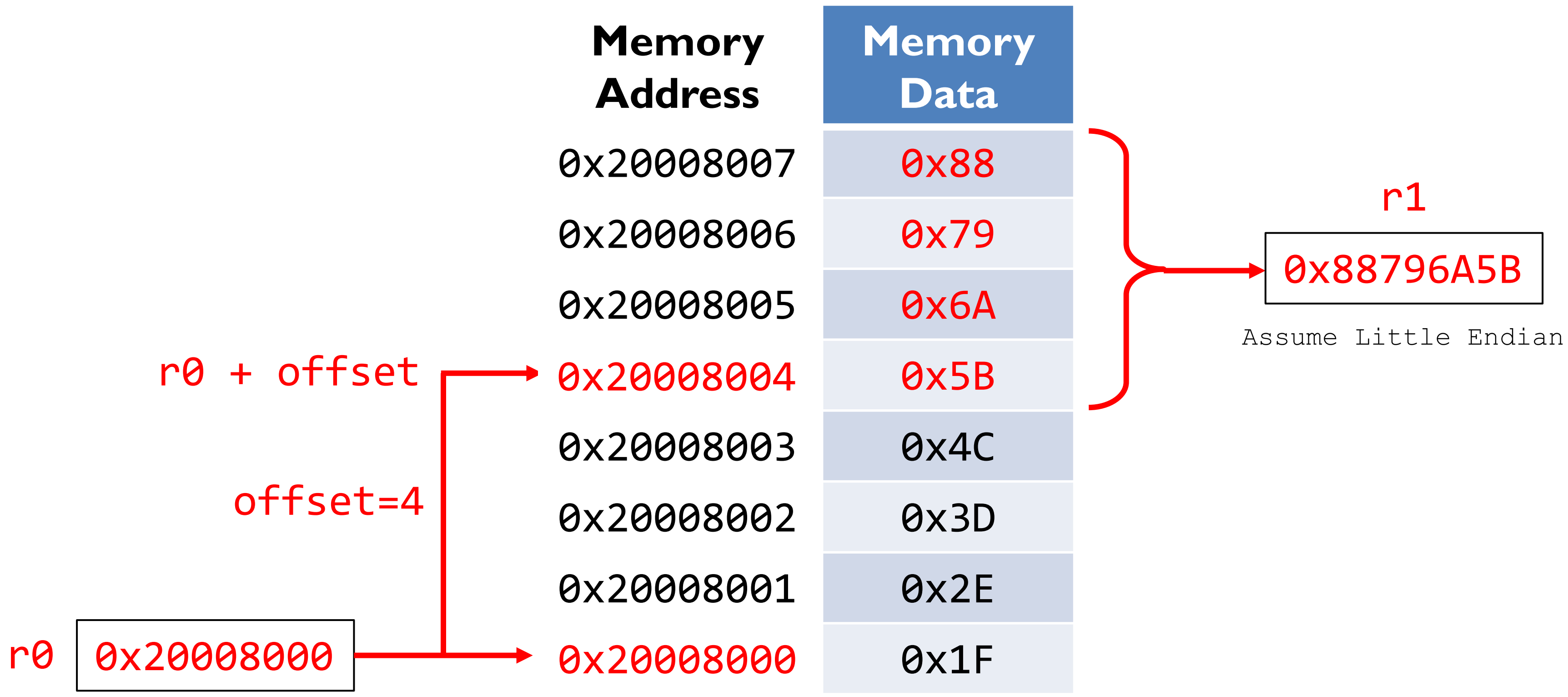
r0 0x20008000

Pre-index

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

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

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

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

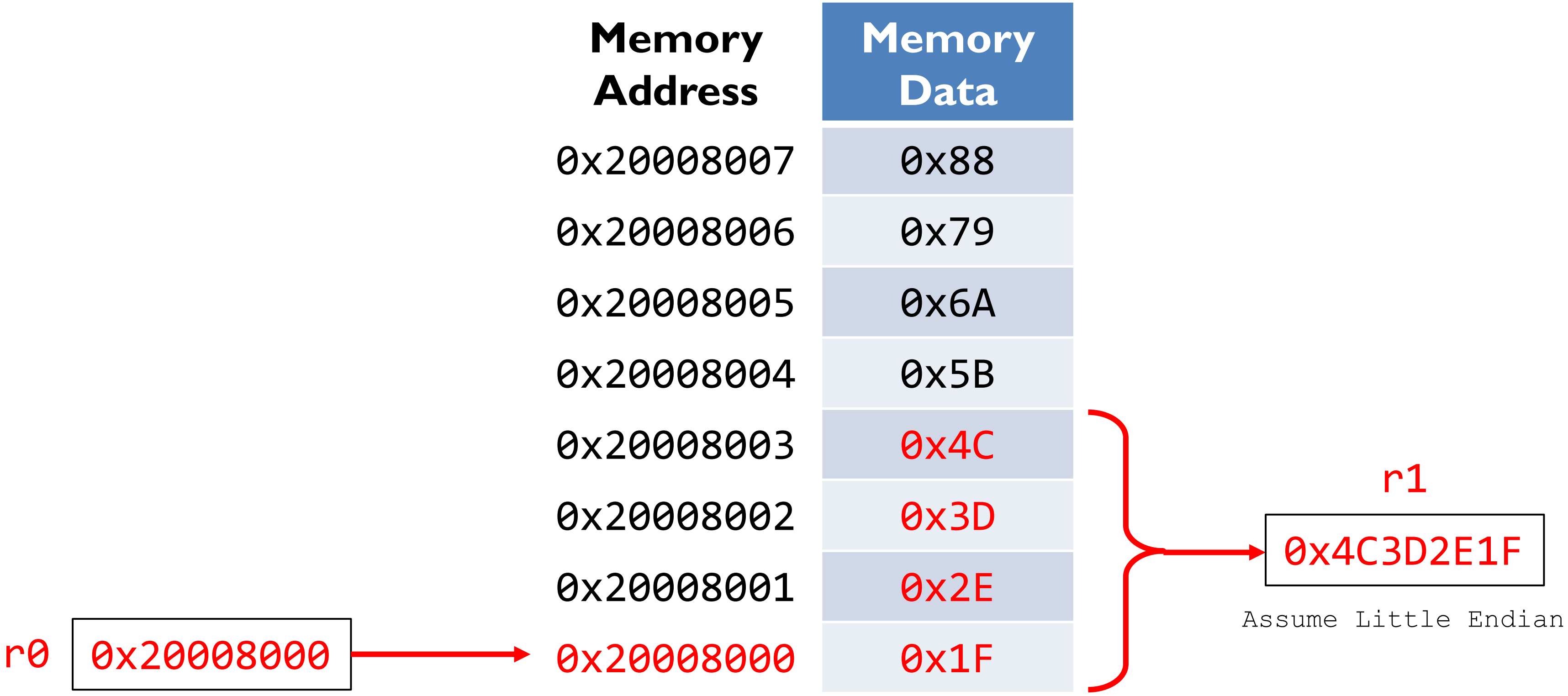
r0 0x20008000

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

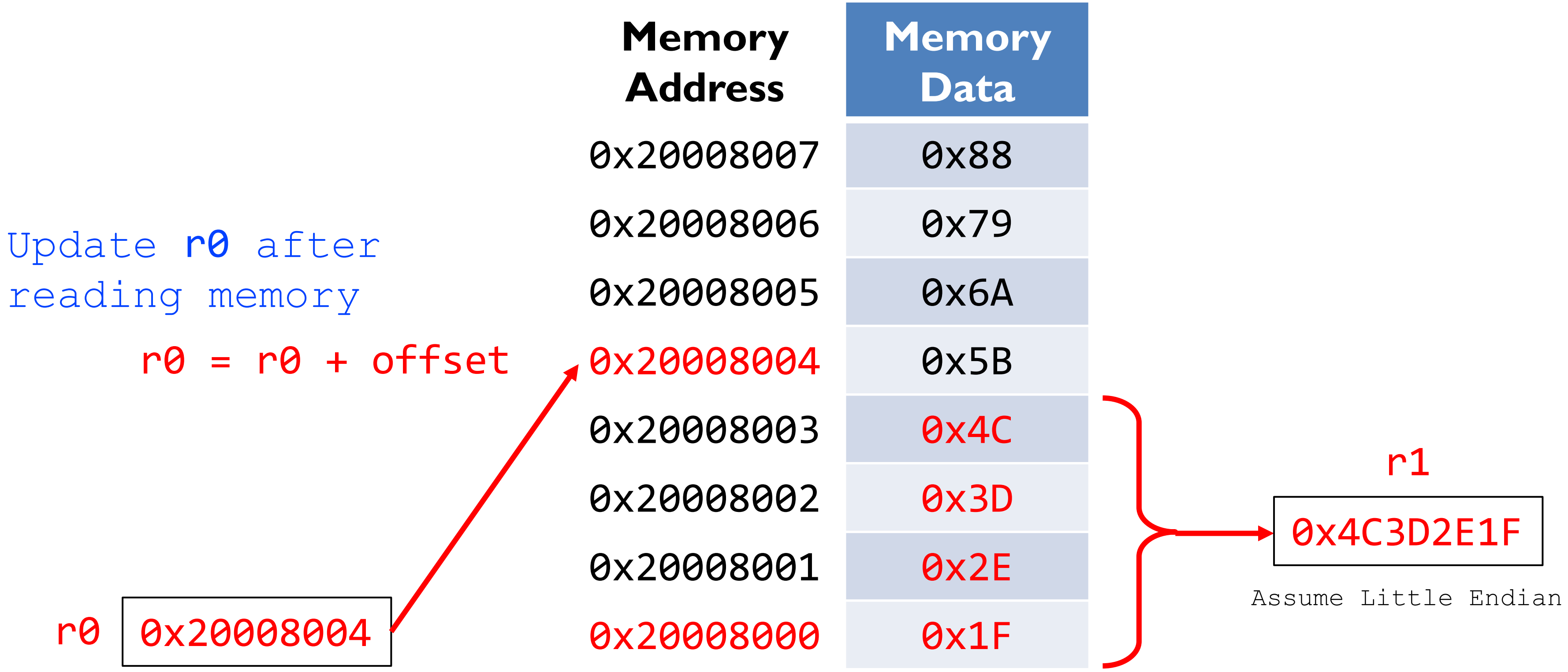
Assume Little Endian

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

Update **r0** after reading memory

r0 = r0 + offset

r0 0x20008004

r1 0x4C3D2E1F

Assume Little Endian

Pre-index with Update

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

Assume: r0 = 0x20008000

Offset: range is -255 to +255

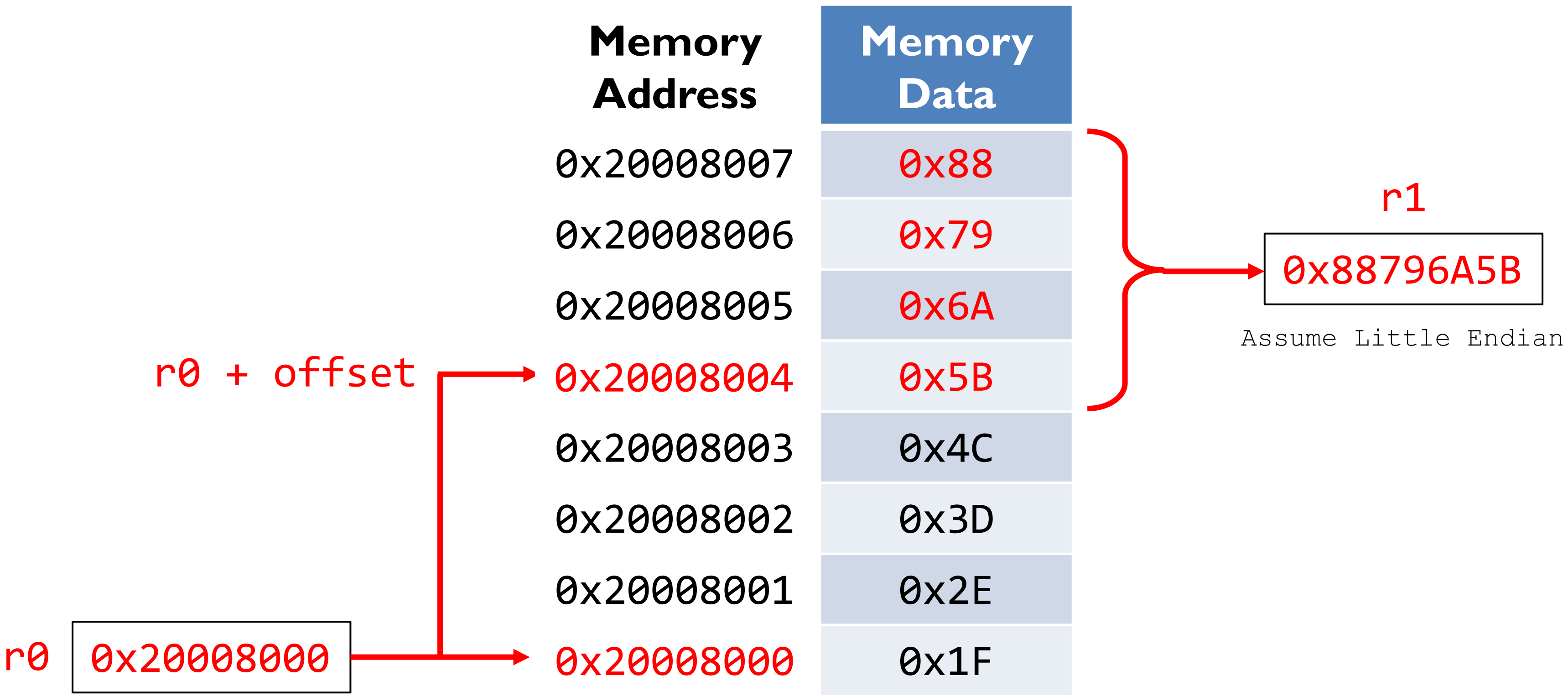
Memory Address	Memory Data
0x20008007	0x88
0x20008006	0x79
0x20008005	0x6A
0x20008004	0x5B
0x20008003	0x4C
0x20008002	0x3D
0x20008001	0x2E
0x20008000	0x1F

Pre-index

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

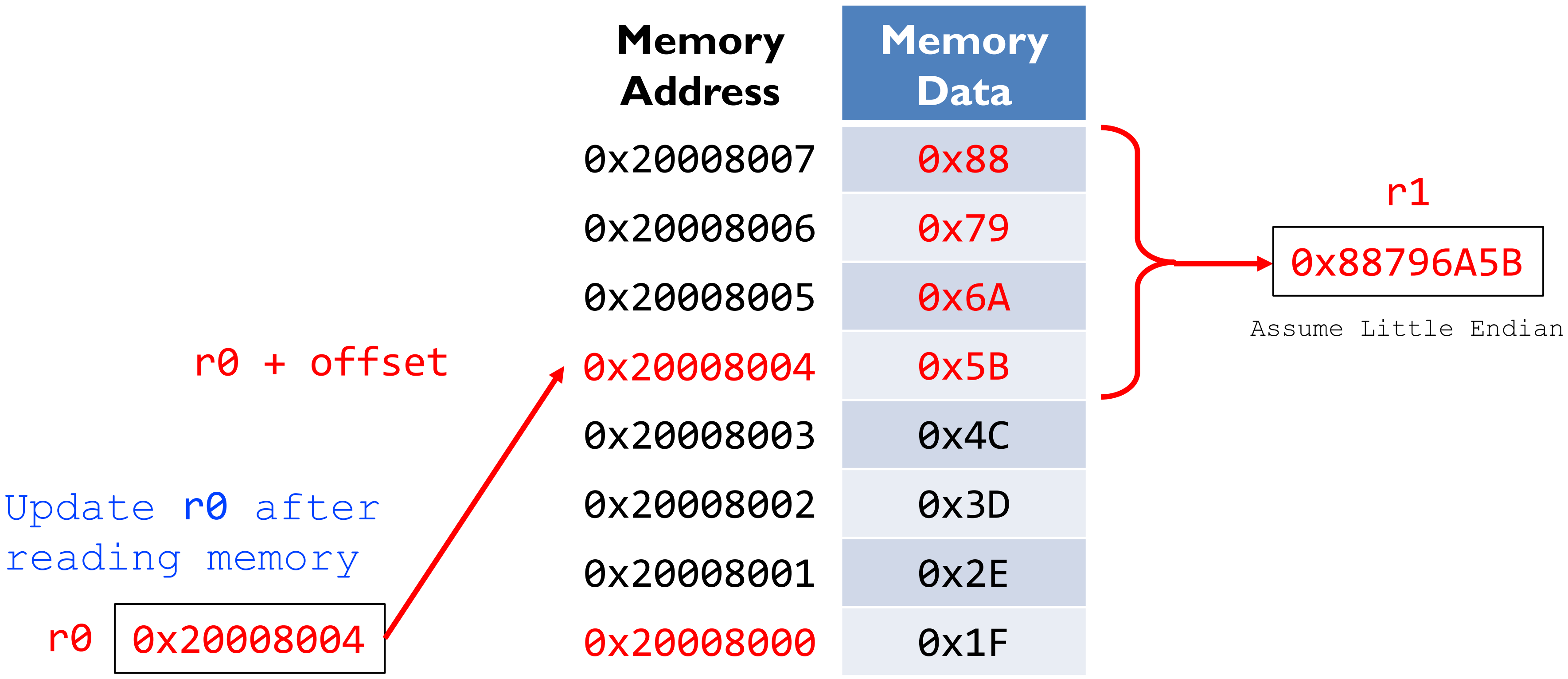
Assume Little Endian

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

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

Offset range is -255 to +255

Example

```
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

```
LDSB 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

```
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
0x20008000	0x00

Example

```
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
0x20008004	0x00
0x20008003	0x76
0x20008002	0x54
0x20008001	0x32
0x20008000	0x10

Example

```
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
0x20008000	0x00

Example

```
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
0x20008000	0x00

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
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
0x20008004	0x10
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Example

If big endianess is used

```
LDR r11, [r0]
; r0 = 0x20008000
```

r11 before load

0x12345678

r11 after load

0xA7908CEE

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

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

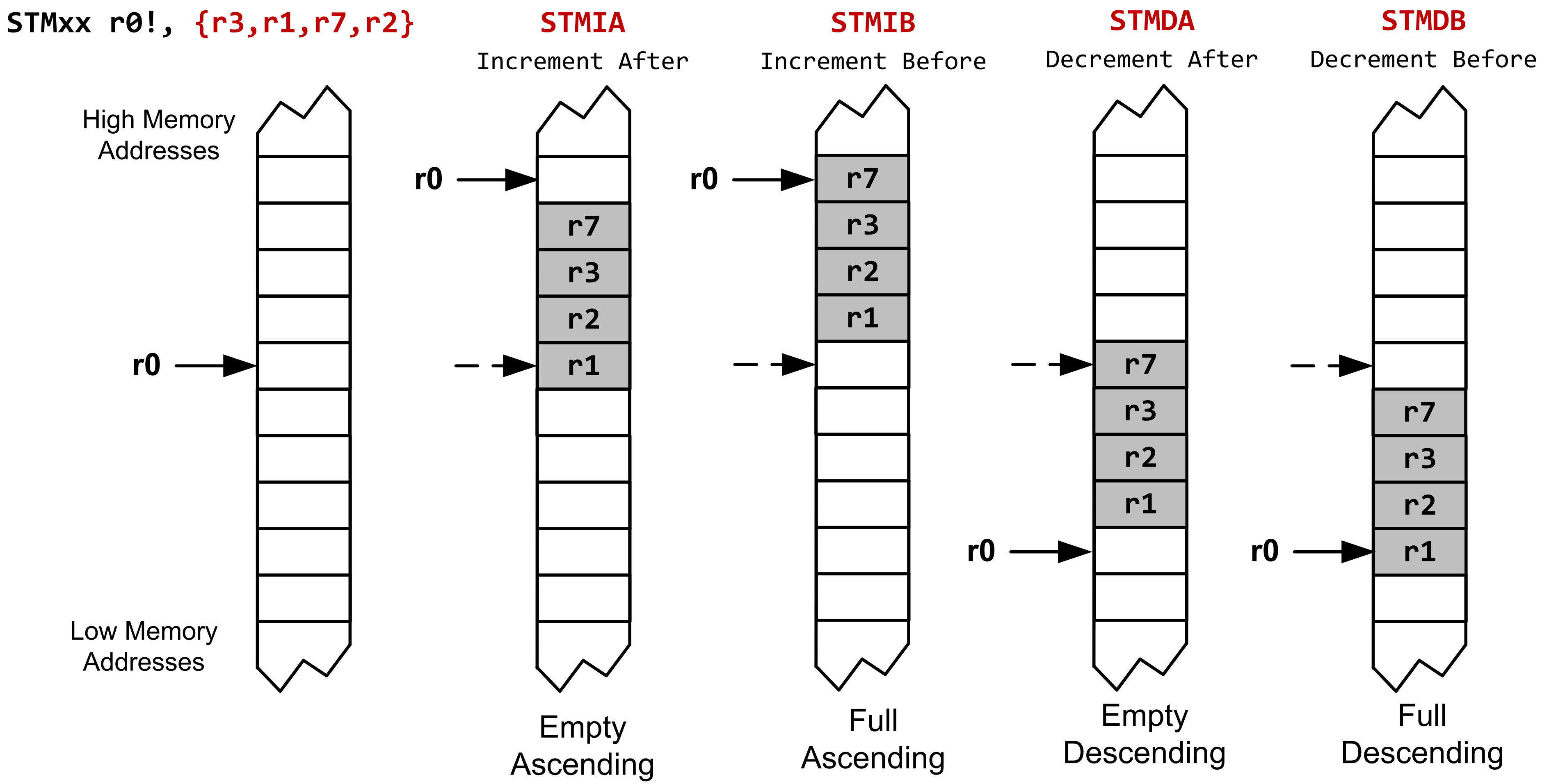
- **IA**: address is incremented by 4 after a word is loaded or stored.
- **IB**: address is incremented by 4 before a word is loaded or stored.
- **DA**: address is decremented by 4 after a word is loaded or stored.
- **DB**: address is decremented by 4 before a word is loaded or stored.

Load/Store Multiple Registers

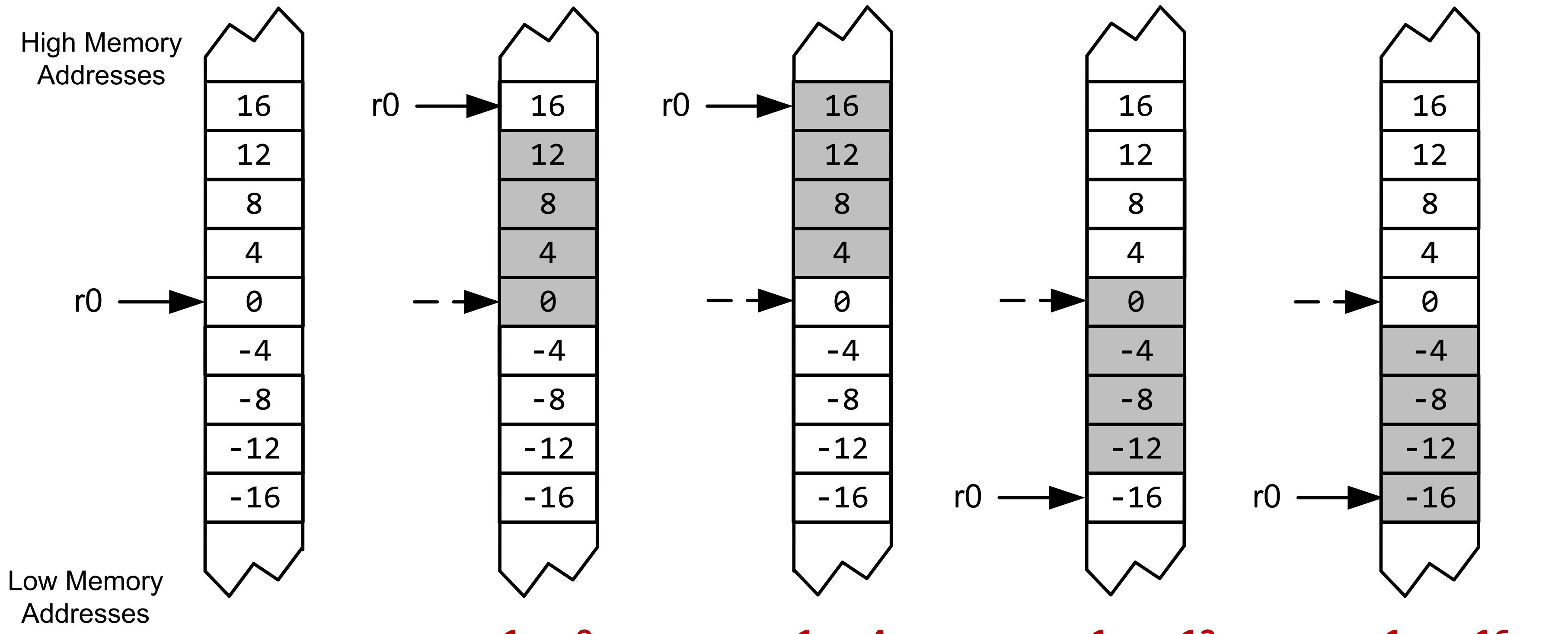
- The following are synonyms.
  - `STM` = `STMIA` (Increment After) = `STMEA` (Empty Ascending)
  - `LDM` = `LDMIA` (Increment After) = `LDMFD` (Full Descending)

- 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



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

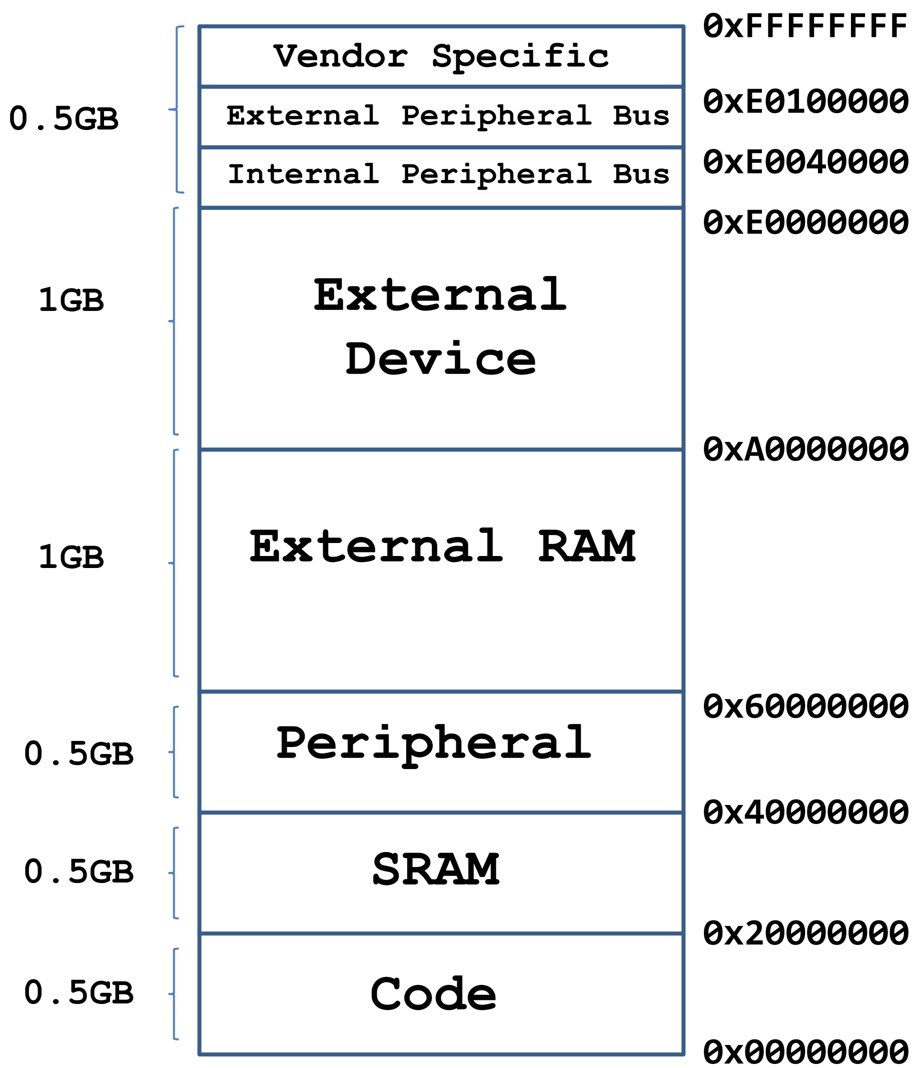
```
r1 = -12
r2 = -8
r3 = -4
r7 = -0
```

LDMDB
Decrement Before

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

Cortex-M3 & Cortex-M4 Memory Map

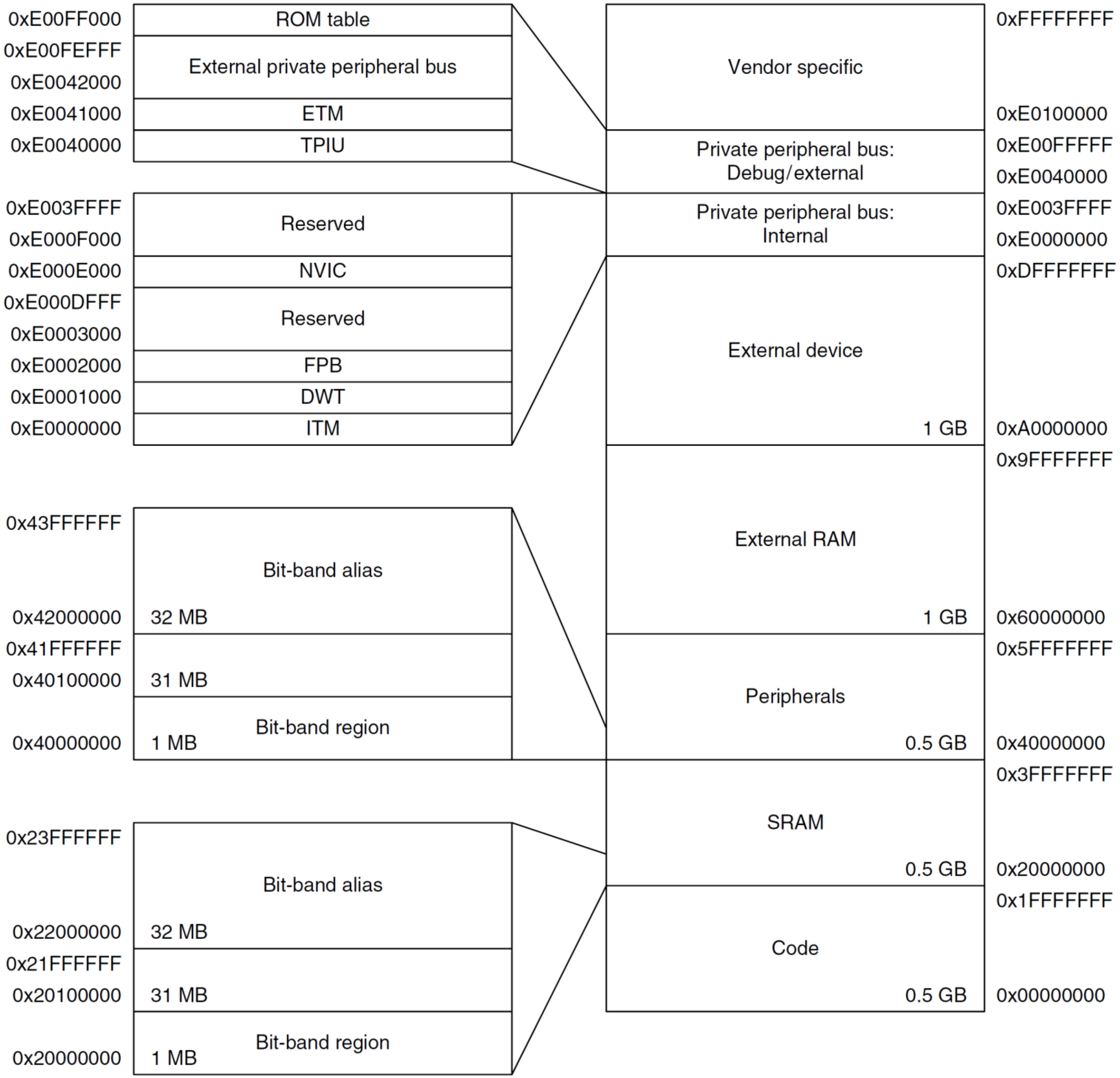
- 32-bit Memory Address
- 2^{32} bytes of memory space (4 GB)
- Harvard architecture: physically separated instruction memory and data memory



Size	Region	Start address
0.5GB	Vendor Specific	0xE0100000
	External Peripheral Bus	0xE0040000
	Internal Peripheral Bus	0xE0000000
1GB	External Device	0xA0000000
1GB	External RAM	0x60000000
0.5GB	Peripheral	0x40000000
0.5GB	SRAM	0x20000000
0.5GB	Code	0x00000000

0xFFFFFFFF

Cortex-M3 Fixed Memory Map



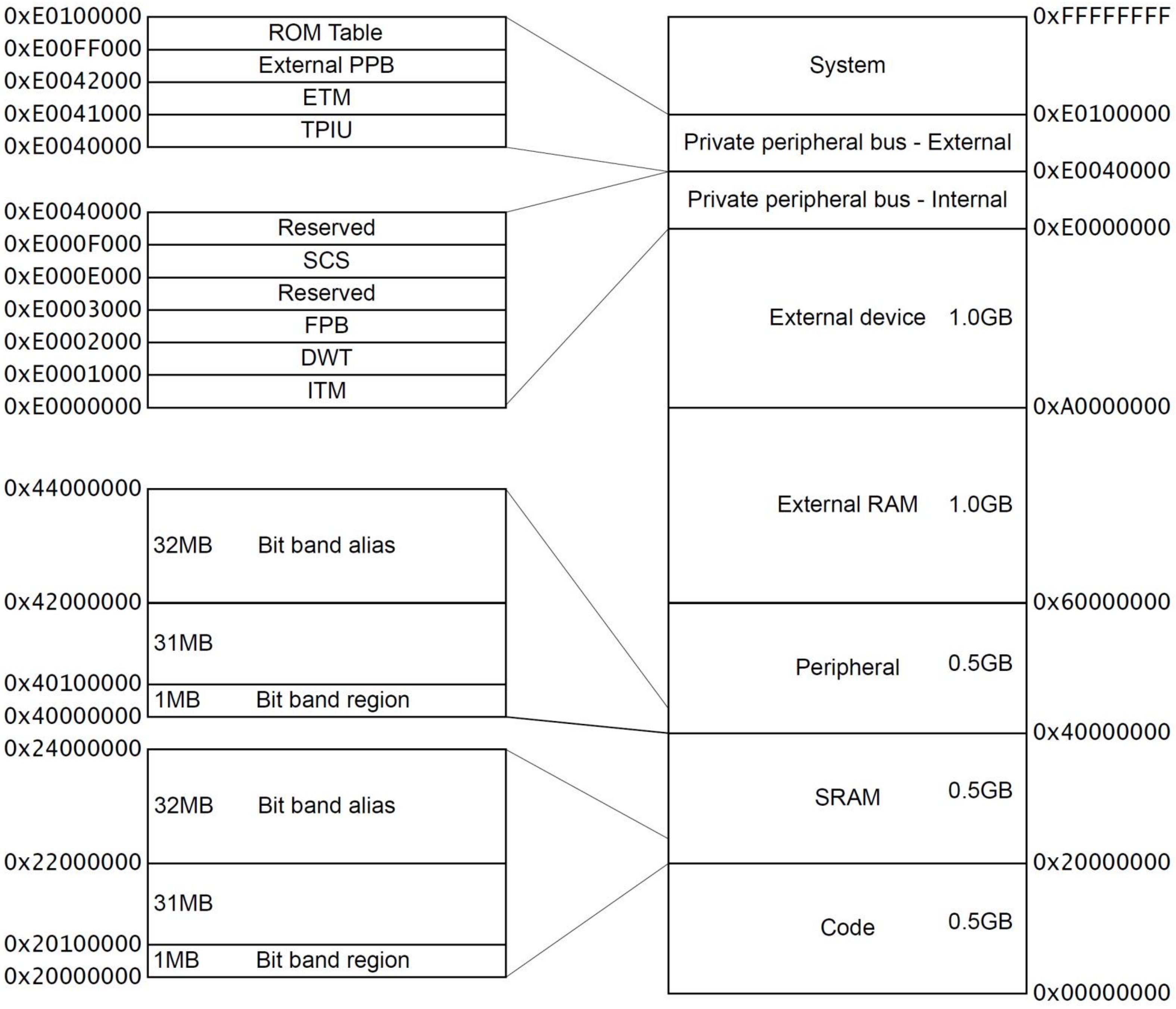
0xE00FF000 ROM table
0xE00FEFFF External private peripheral bus
0xE0042000
0xE0041000 ETM
0xE0040000 TPIU

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
0x40000000 1 MB Bit-band region

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

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



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

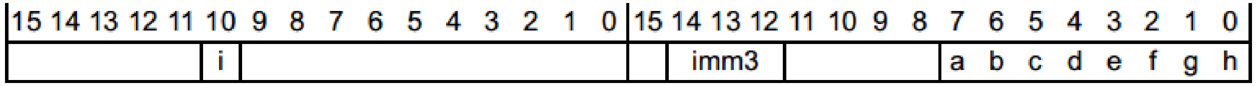
```
LDR Rt, =expr
LDR Rt, =label
```

- If the value of expr can be loaded with **MOV**, **MVN** (16-bit instruction) or **MOVW** (32-bit instruction), the assembler uses that instruction.
- If a valid MOV, MVN, MOVW instruction cannot be used, 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.

12-bit Encoding of Immediate Numbers



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
					i												imm3							a	b	c	d	e	f	g	h

- MOV supports all 8-bit immediate numbers
- Range of 8-bit immediate number: 0 – 255
- Numbers out of this range but with some patterns can be encoded.

Decoding 12-bit Immediate Value:

11	10	9	8	7	6	5	4	3	2	1	0
i	imm3			a	b	c	d	e	f	g	h

Replication Format

i	imm3			a	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
0	0	0	0	x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	a	b	c	d	e	f	g	h
0	0	0	1	x	0	0	0	0	0	0	0	0	a	b	c	d	e	f	g	h	0	0	0	0	0	0	0	0	a	b	c	d	e	f	g	h
0	0	1	0	x	a	b	c	d	e	f	g	h	0	0	0	0	0	0	0	0	a	b	c	d	e	f	g	h	0	0	0	0	0	0	0	0
0	0	1	1	x	a	b	c	d	e	f	g	h	a	b	c	d	e	f	g	h	a	b	c	d	e	f	g	h	a	b	c	d	e	f	g	h

Rotation Format

0	1	0	0	0	1	b	c	d	e	f	g	h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	0	1	b	c	d	e	f	g	h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	1	b	c	d	e	f	g	h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	1	0	0	0	1	b	c	d	e	f	g	h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	1	b	c	d	e	f	g	h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		⋮							⋮																											
1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	b	c	d	e	f	g	h	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	b	c	d	e	f	g	h	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	b	c	d	e	f	g	h	0

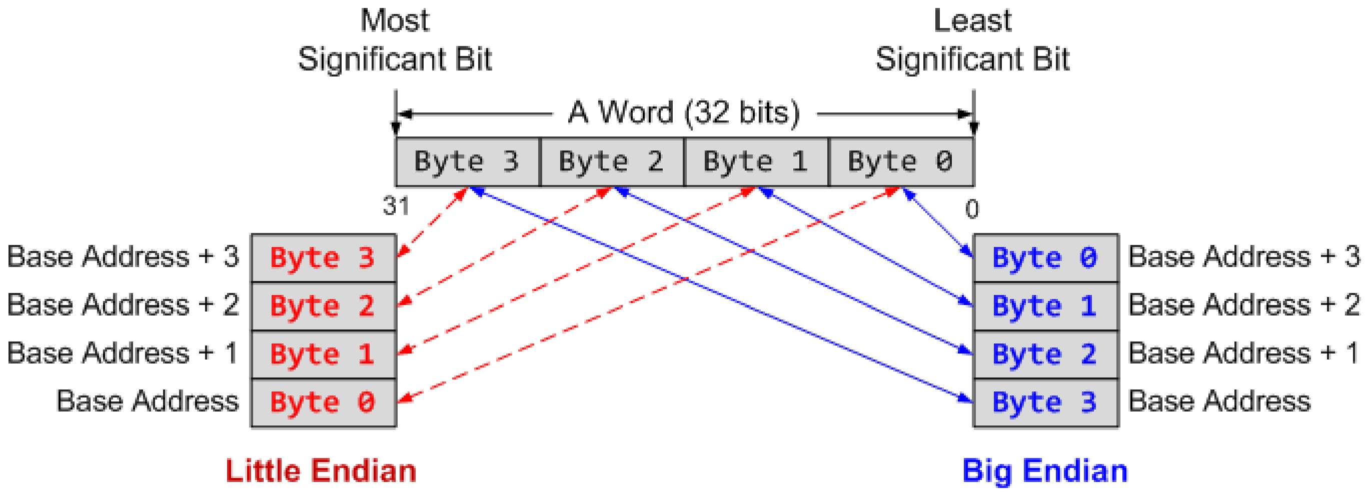
ADR Pseudo-instruction

- **ADR**: loads a program-relative or register-relative address into a register

```
start   MOV r0,#10        ; 32-bit instruction
        ADR r4,start      ; 32-bit instruction
                          ; => SUB r4,pc,#0xc
```

Summary

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