Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

6.1 Memory

CSU11021 – Introduction to Computing I

Dr Jonathan Dukes | jdukes@tcd.ie
School of Computer Science and Statistics

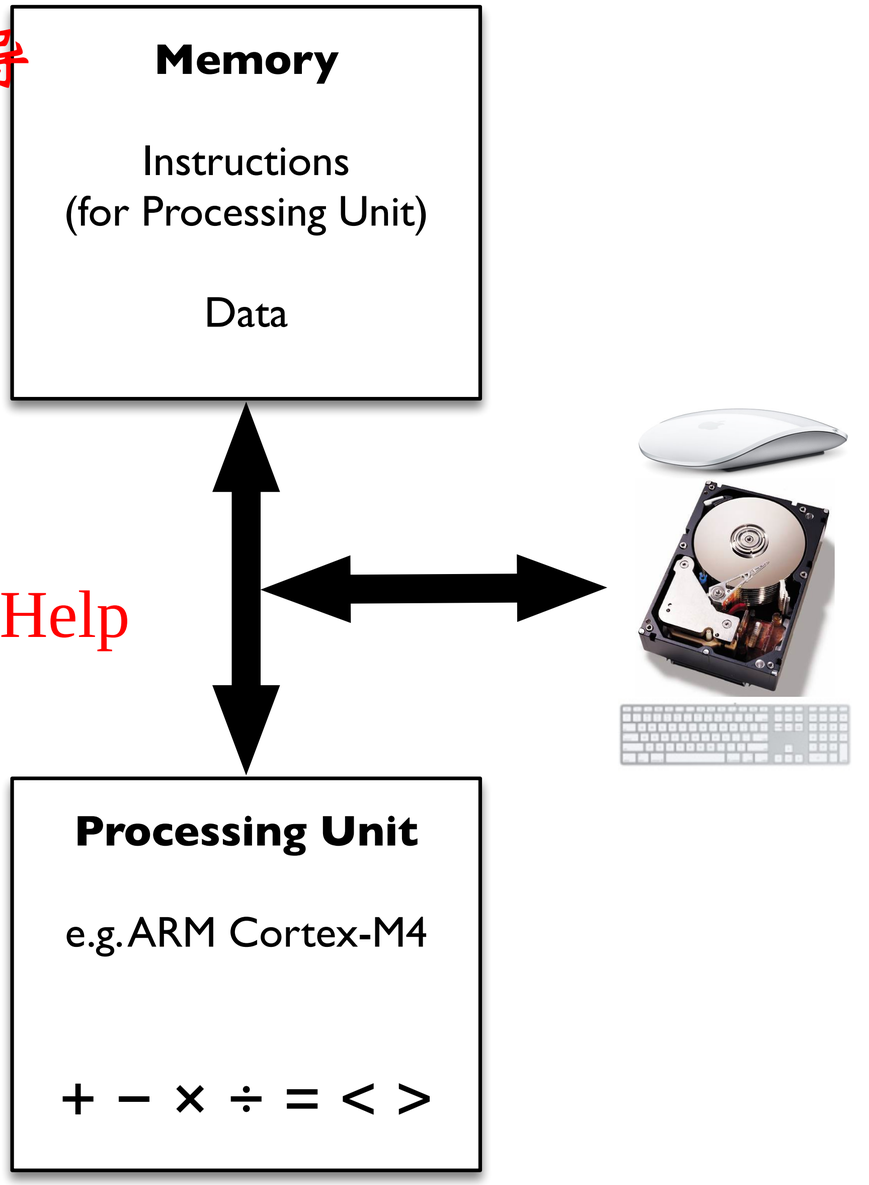
Simple Model of a Microprocessor System

A **processing unit** or **processor** which performs operations on data

Memory, which stores:

- **Data**: representing text, images, videos, sensor readings, π, audio, etc. ...
- **Instructions**: Programs are composed of sequences of instructions that control the actions of the processing unit

So far, all of our data has been stored in registers, internal to the Processing Unit ("processor" or "CPU")



Using Memory: Upper Case String Example

Design and write an assembly language program to convert a string stored in memory to UPPER CASE

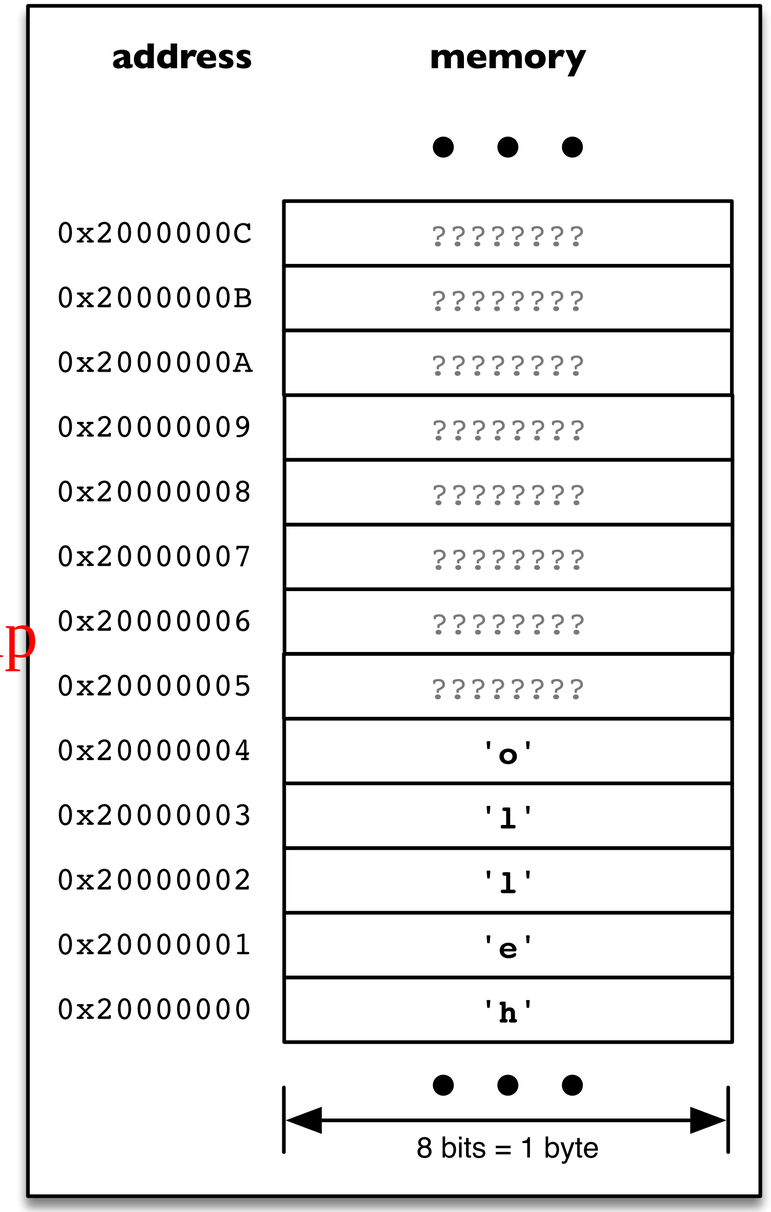
String – sequence of ASCII characters stored in consecutive memory locations

“hello”

```
character = first character in string

while (character not past end of string)
{
    if (character ≥ ‘a’ AND character ≤ ‘z’)
    {
        character = character – 0x20
    }

    character = next character
}
```



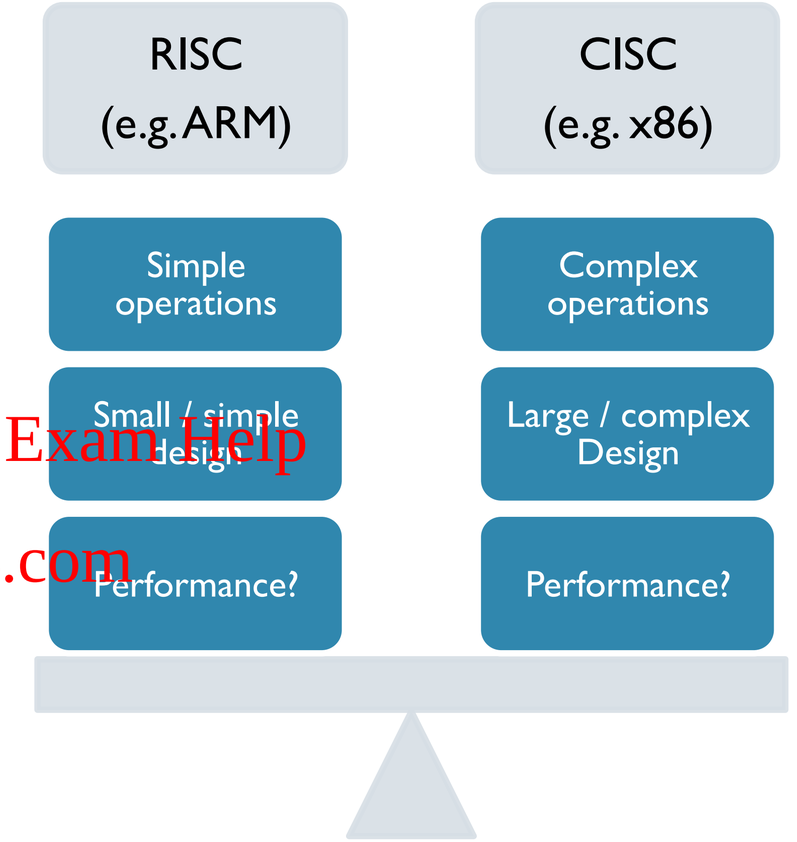
Load – Store Architecture

ARM is a "**Load – Store Architecture**"

Cannot directly perform operations (e.g. addition, subtraction, comparison, ...) on values in memory

Only way to operate on a value stored in memory is to load it into a register, then operate on the register

Only way to change a value in memory is to store the value from a register into memory



Using Memory: Upper Case String Example

Using memory addresses and load/store approach…


```
character = first character in strin

while (character not past end of str
{
    if (character ≥ ‘a’ AND character ≤ ‘z’)
    {
        character = character - 0x20
    }

    character = next character
}
```


```
address = address of first character
ch = byte[address]

while (ch not past end of string)
{
    if (ch ≥ ‘a’ AND char ≤ ‘z’)
    {
        ch = ch - 0x20
        byte[address] = ch
    }

    address = address + 1
    ch = byte[address]
}
```


`ch = byte[address]`

Load the byte-size contents of memory at address **`address`** *into the variable* **`ch`**

`ch` and **`address`** will be values in registers

This is my pseudo-code notation … you are free to use your own!

Using Memory: Upper Case String Example

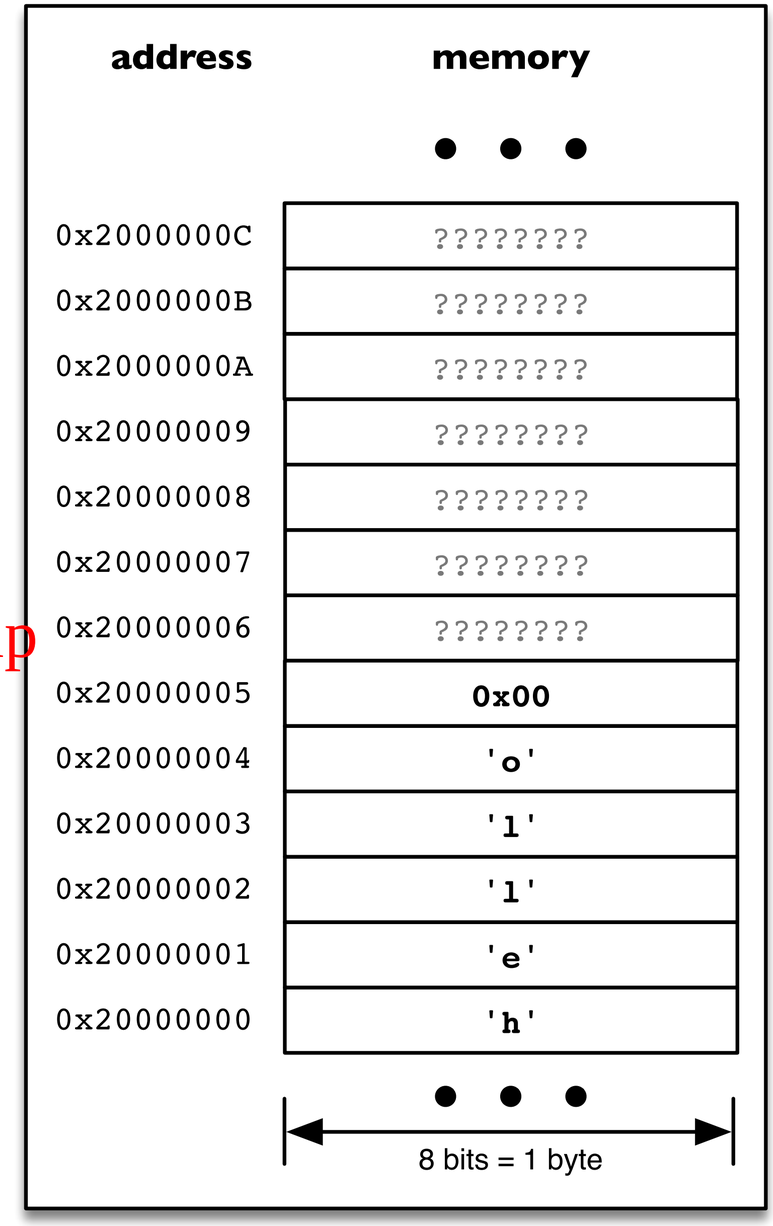
How do we know when we have reached the end of the string?

NULL terminated strings use code 0 (ASCII NULL character code) to mark the end of a string


```
address = 0x20000000;
ch = byte[address];

while (ch != 0)
{
    if (ch >= ‘a’ && ch <= ‘z’)
    {
        ch = ch - 0x20;
        byte[address] = ch;
    }

    address = address + 1;
    ch = byte[address];
}
```



Using Memory: Upper Case String Example


```
Main:
                          @ address initialised in test.s
  LDRB  R2, [R1]          @ ch = byte[address];
While:                    @
  CMP   R2, #0            @ while(ch != 0)
  BEQ   EndWhile          @ {
  CMP   R2, #'a'          @   if(ch >= 'a' && ch <= 'z')
  BLO   EndIfLwr          @   {
  CMP   R2, #'z'          @
  BHI   EndIfLwr          @
  SUB   R2, R2, #0x20     @     ch = ch - 0x20;
  STRB  R2, [R1]          @     byte[address] = ch;
EndIfLwr:                 @   }
  ADD   R1, R1, #1        @   address = address + 1;
  LDRB  R2, [R1]          @   ch = byte[address];
  B     While             @ }
EndWhile:
```


Where does the string in memory come from?

We can initialise memory with a test string using test.s

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

6.2 LDR, STR, bytes, halfwords and words

CSU11021 – Introduction to Computing I

Dr Jonathan Dukes | jdukes@tcd.ie
School of Computer Science and Statistics

Using Memory: Upper Case String Example

Possible optimisation by moving the LDRB to the top of the while loop

… at the expense of less elegant pseudo-code …

```
  LDRB   R2, [R1]        @ ch = byte[address];
While:                   @
  LDRB   R2, [R1]        @ while ((ch = byte[address]) != 0)
  CMP    R2, #0          @
  BEQ    EndWhile        @ {
  CMP    R2, #'a'        @   if (ch >= 'a' && ch <= 'z')
  BLO    EndIfLwr        @   {
  CMP    R2, #'z'        @
  BHI    EndIfLwr        @
  SUB    R2, R2, #0x20   @     ch = ch - 0x20;
  STRB   R2, [R1]        @     byte[address] = ch;
EndIfLwr:                @   }
  ADD    R1, R1, #1      @   address = address + 1;
  LDRB   R2, [R1]        @   ch = byte[address];
  B      While           @ }
EndWhile:
```

Exercise: strlen

Design and write an ARM Assembly Language program that will calculate the length of the string stored in memory beginning at the address contained in R1.

In other words, count the number of characters in the string, up to but excluding the NULL character.

Test yourself by submitting your solution to Submitty

LDR / STR Instructions

Load a word-, half-word- or byte-size value from a specified address into a r

```
LDR    R0, =0x20000000

LDR    R1, [R0]     @ Load word at 0x20000000 (32 bits)
LDRH   R1, [R0]     @ Load half-word at 0x20000000 (16 bits)
LDRB   R1, [R0]     @ Load byte at 0x20000000 (8 bits)
```


Store a word-, half-word- or byte-size value from a register into memory at a specified address

```
LDR    R0, =0x20000000

STR    R1, [R0]     @ Store word at 0x20000000 (32 bits)
STRH   R1, [R0]     @ Store half-word at 0x20000000 (16 bits)
STRB   R1, [R0]     @ Store byte at 0x20000000 (8 bits)
```

address	memory
0x20000005	64
0x20000004	7B
0x20000003	5D
0x20000002	35
0x20000001	27
0x20000000	89
0x1FFFFFFF	82
0x1FFFFFFE	3C
0x1FFFFFFD	8B
0x1FFFFFFC	53
0x1FFFFFFB	A2
0x1FFFFFFA	9F
0x1FFFFFF9	E8
0x1FFFFFF8	4D
0x1FFFFFF7	0A
0x1FFFFFF6	07

Using Memory: Sum Example

Design and write an assembly language program that will calculate the sum of 10 word-size values stored in memory, beginning at the address in R1. Store the sum in R0


```
sum = 0;
i = 0;
while (i < 10)
{
  sum = sum + word[address];
  address = address + 4;
  i = i + 1;
}
```

Using Memory: Sum Example


```
   MOV    R0, #0          @
                          @
   MOV    R2, #0          @
While:                    @
   CMP    R2, #10         @ while (i < 10)
   BHS    EndWhile        @ {
                          @
   LDR    R3, [R1]        @   value = word[address];
   ADD    R0, R0, R3      @   sum = sum + value;
   ADD    R1, R1, #4      @   address = address + 4;
   ADD    R2, R2, #1      @   i = i + 1;
                          @
   B      While           @ }
EndWhile:                 
```

Memory: ROM and RAM

Read Only Memory (ROM)

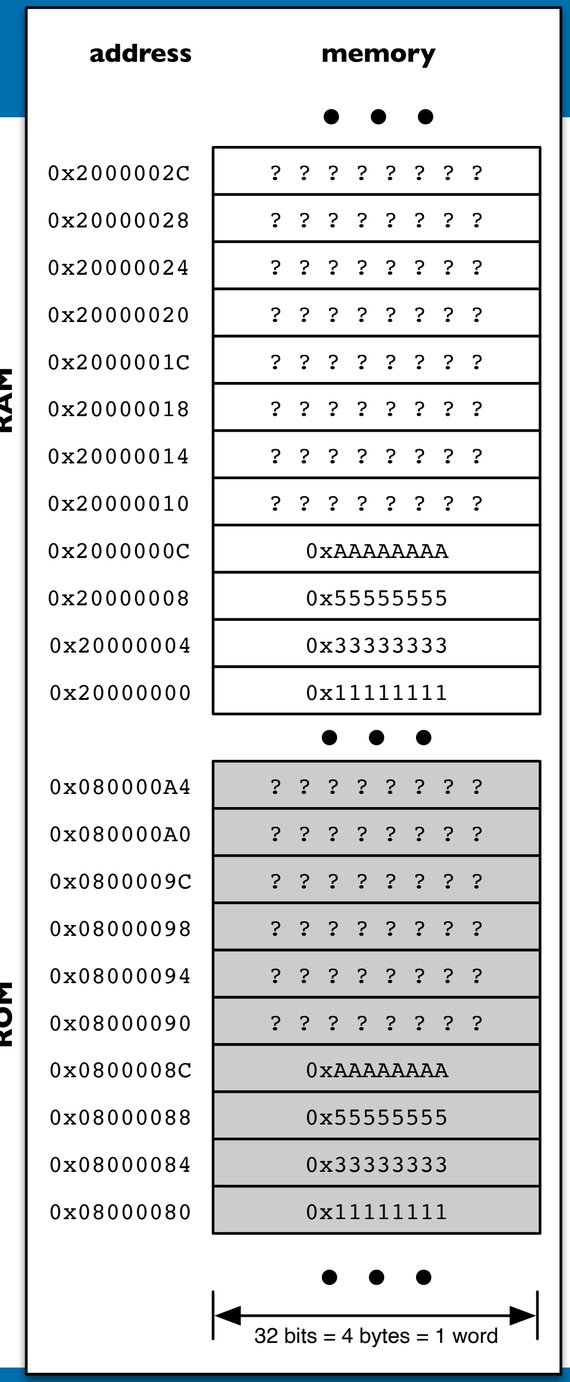
- Cannot be modified by a running program
- Can be read (loaded using LDR) but [illegible] stored using STR)
- Initial contents must be set before a [illegible] rts running
- Initial contents set when our program is built

Random Access Memory (RAM)

- Can be modified by a running program
- Can be read (loaded using LDR) and written (stored using STR)
- Initial contents cannot be set before a program starts

If we want to set initial contents of RAM, we must

- Place the initial contents in ROM when the program is built
- Write a small program to copy the initial contents from ROM to RAM when the program starts



Initialising contents of memory (ROM)

Our examples initialise the contents of ROM in test.s

```
  .section  .rodata

values:
  .word   5, 10, 15, 20, 25, 30, 35, 40, 45, 50

moreValues:
  .hword 10, 12, 100, 125

stillMoreValues:
  .byte    0x10, 0x11, 0xFE, 0xFA
```

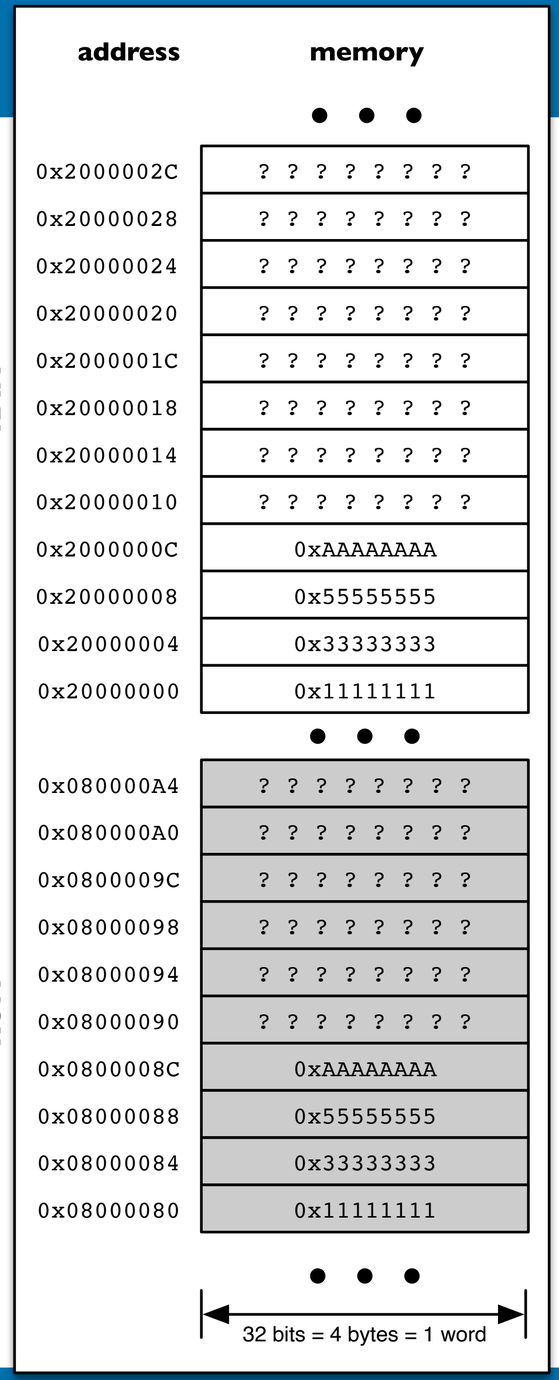
Initialising contents of RAM

If we want to initialise RAM we need to

- place the initial contents in ROM when our program is build
- write a small program (in test.s in our examples) to copy initial contents from ROM to RAM

See test.s in `strupr` exercise for an example of this!

In CSU11021 you will be given an appropriate test.s with this RAM initialisation code written for you



Exercise: strcpy

Design and write an ARM Assembly Language program that will make a copy of a NULL-terminated string stored in memory. The original string is stored in memory starting at the address in R1. Store the new copy of the string in memory beginning at the address in R0.

Test yourself by submitting your solution to Submitty

Exercise: strrev

Design and write an ARM Assembly Language program that will reverse a NULL-terminated string in memory. The original string is stored in memory starting at the address in R1. Your program should store the reversed string in memory beginning at the address in R0.

For example, if the original string is “hello”, your program should create a new string “olleh”.

Test yourself by submitting your solution to Submitty

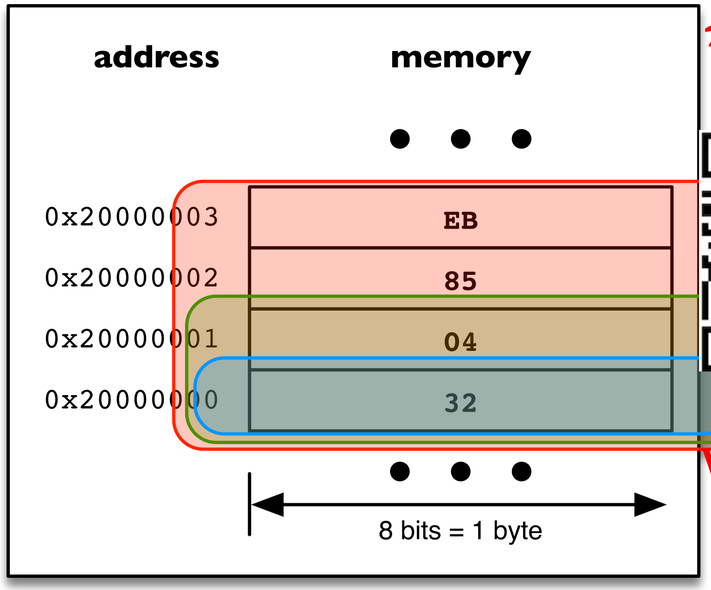
Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

6.3 Memory oddities, “Gulliver’s Travels”

CSU11021 – Introduction to Computing I

Dr Jonathan Dukes | jdukes@tcd.ie
School of Computer Science and Statistics

Load / Store Bytes, Half-words and Words

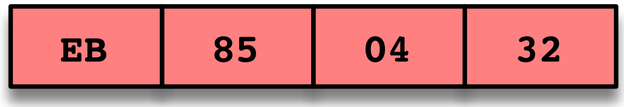


Byte, **half-word** and **word** at address 0x20000000


```
LDR     r0, =0x20000000
LDRB    r1, [r0]
```


```
LDR     r0, =0x20000000
LDRH    r1, [r0]
```

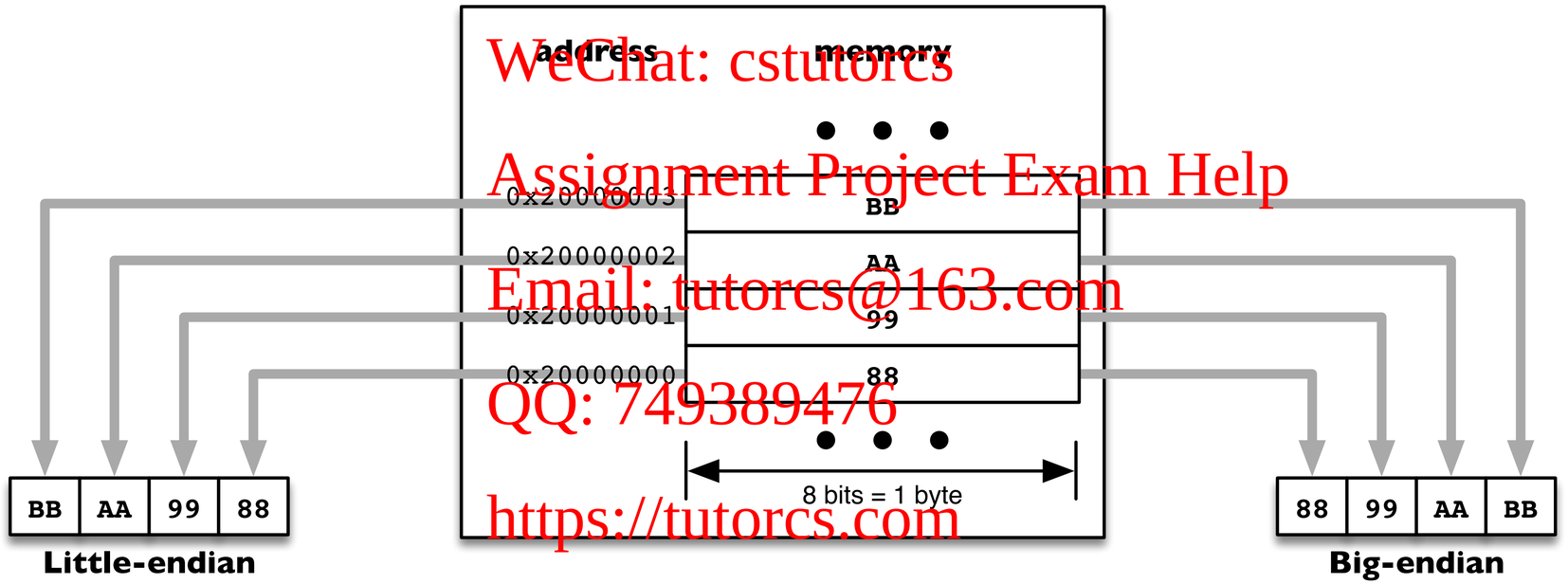

```
LDR     r0, =0x20000000
LDR     r1, [r0]
```



Endian-ness

Little-endian byte ordering – least-significant byte of word or half-word stored at lower address in memory

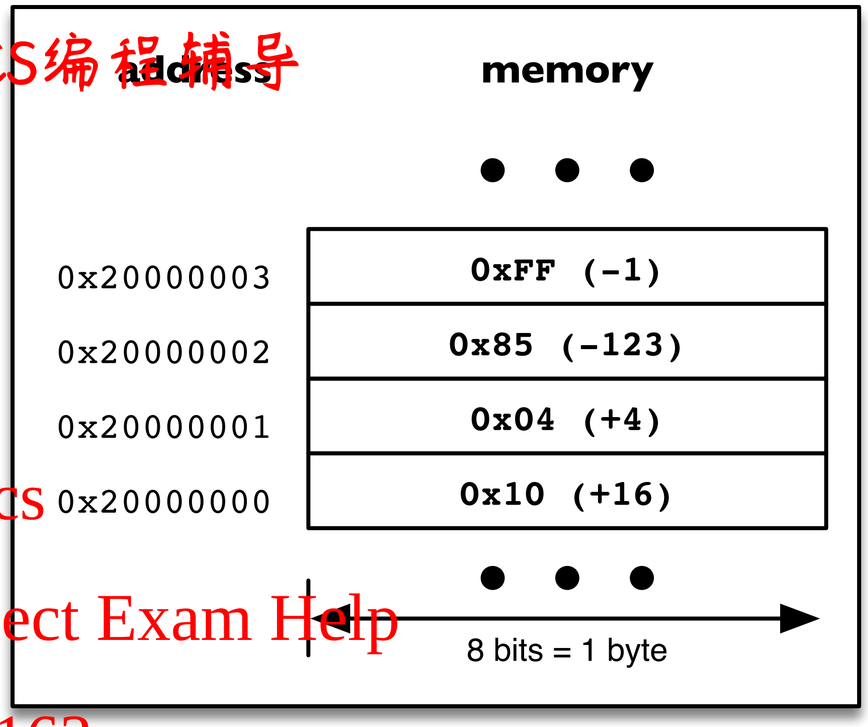
Big-endian byte ordering – most-significant byte of word or half-word stored at lower address in memory



[1] Cohen, Danny, “On Holy Wars and a Plea for Peace”, IETF, IEN 137, April 1980.
[2] Swift, Jonathan, “Gulliver’s Travels”, 1726.

Loading Signed Bytes and Half-words

Consider the four byte-sized signed 2's complement values stored in memory on the right



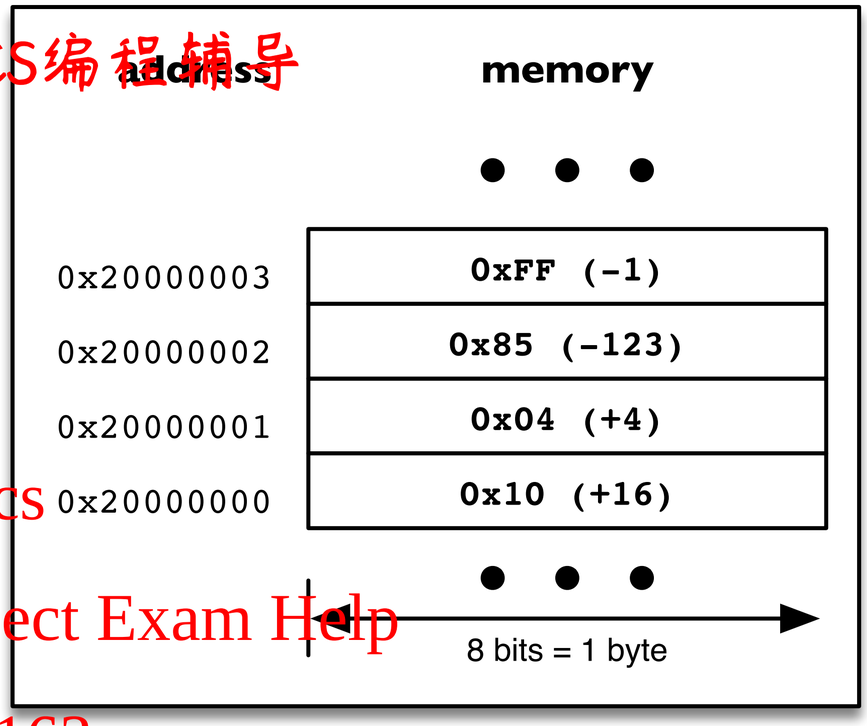
After executing the pair of instructions below, what is the correct signed decimal interpretation of the value loaded in R1?

```
LDR     r0, =0x20000000
LDRB    r1, [r0]
```

(a) -16
(b) +240
(c) +16
(d) -240

Loading Signed Bytes and Half-words

Consider the four byte-sized signed 2's complement values stored in memory on the right



After executing the pair of instructions below, what is the correct signed decimal interpretation of the value loaded in R1?

```
LDR     r0, =0x20000003
LDRB    r1, [r0]
```

(a) +1
(b) +255
(c) -1
(d) -255

Loading Signed Bytes and Half-words

Sign extension performed when loading signed bytes or half-words to facilitate correct subsequent signed arithmetic


```
LDR     r0, =0x20000000
LDRSB   r1, [r0]
```

00	00	00	32

```
LDR     r0, =0x20000000
LDRSH   r1, [r0]
```

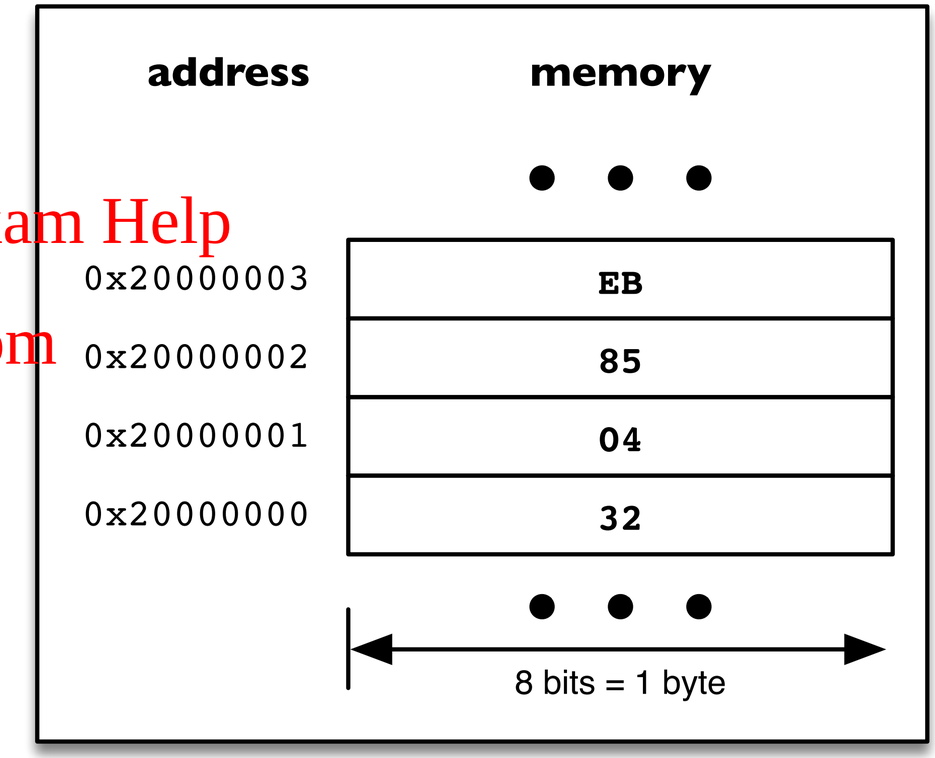
00	00	04	32

```
LDR     r0, =0x20000002
LDRSB   r1, [r0]
```

FF	FF	FF	85

```
LDR     r0, =0x20000002
LDRSH   r1, [r0]
```

FF	FF	EB	85



Loading Signed Bytes and Half-words

Consider the four byte-sized signed 2's complement values stored in memory on the right

After executing the pair of instructions below, what is the correct signed decimal interpretation of the value loaded in R1?

```
LDR      r0, =0x20000001
LDRSB    r1, [r0]
```

(a) -4
(b) -252
(c) +252
(d) +4