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

1.1 – Memory and Addressing Modes

CSU11022 – Introduction to Computing II

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

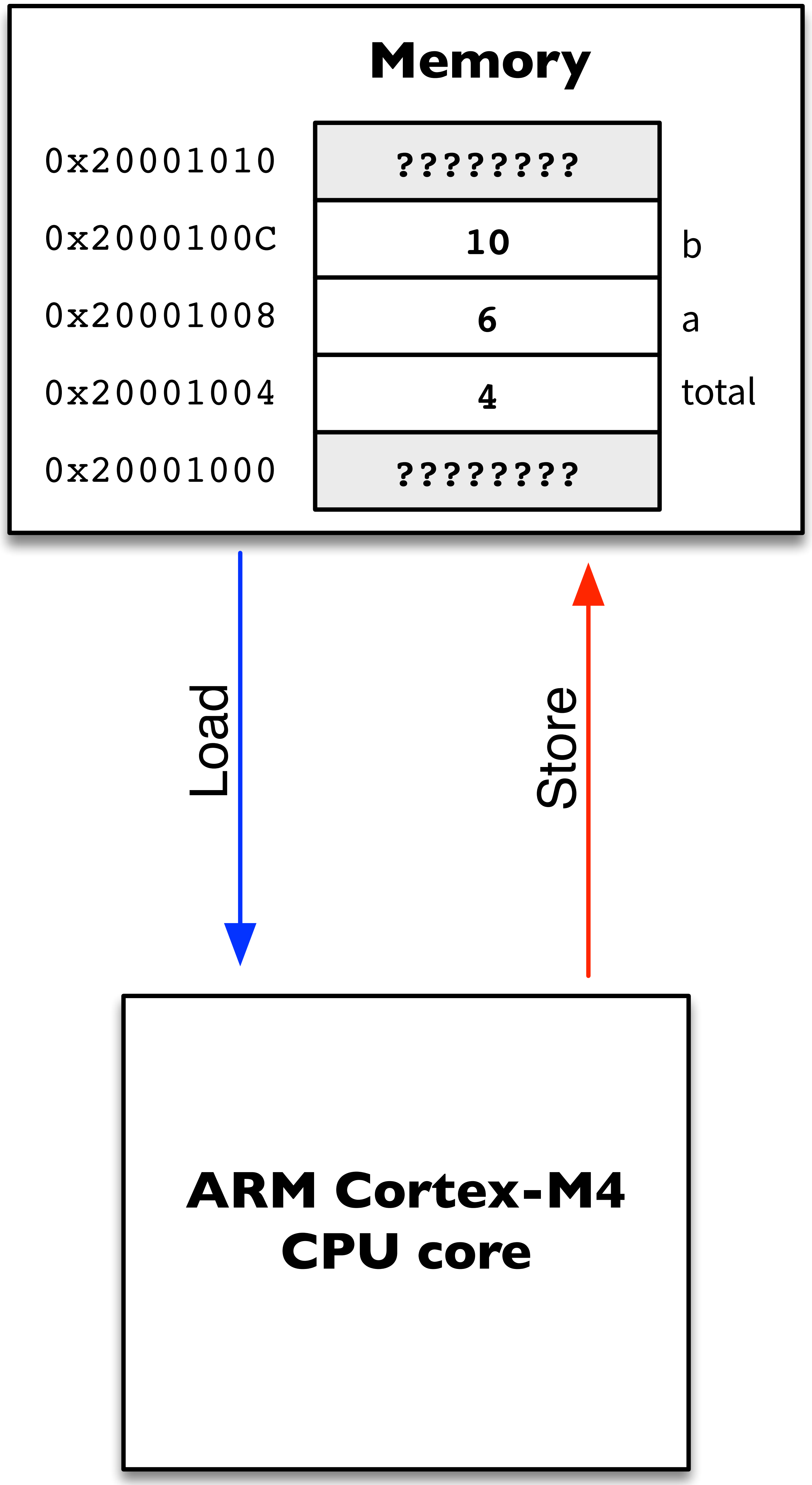
Recap: LDR and STR

LDR and STR Reviewed

How many memory acce[illegible] required to compute

total = total + (a × b)

where **total**, **a** and **b** are stored in memory at the addresses contained in **R0**, **R1** and **R2** respectively?



Memory

Address	Value	
0x20001010	????????	
0x2000100C	10	b
0x20001008	6	a
0x20001004	4	total
0x20001000	????????	

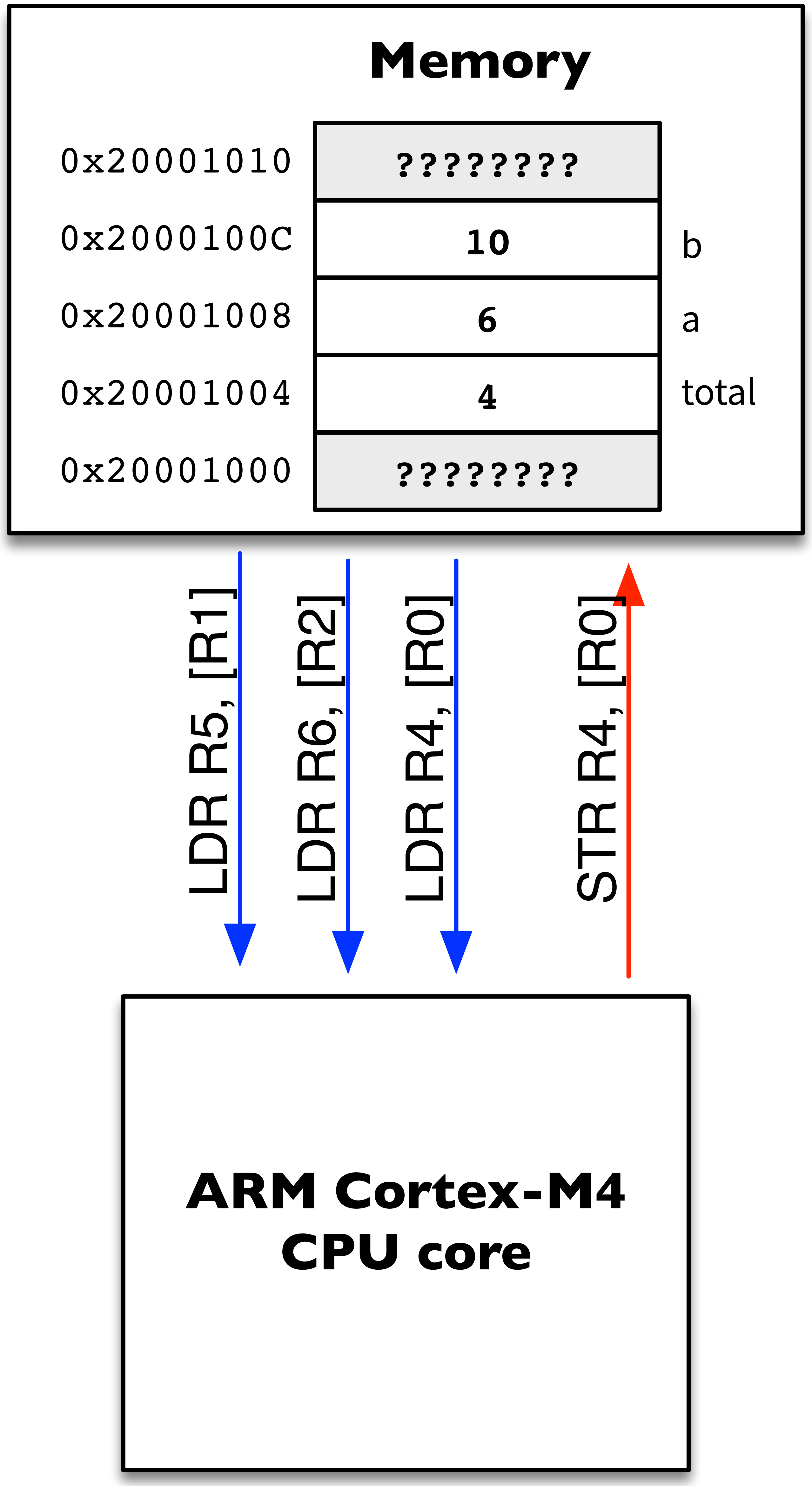
LDR and STR Reviewed

How many memory accesses are required to compute **total = total + (a × b)**, where **total**, **a** and **b** are stored in memory at the addresses contained in **R0**, **R1** and **R2** respectively?

```
LDR     R5, [R1]        @ Load a
LDR     R6, [R2]        @ Load b
MUL     R5, R6, R5      @ tmp = a * b

LDR     R4, [R0]        @ Load total
ADD     R4, R4, R5      @ total = total + (tmp)

STR     R4, [R0]        @ Store total back to memory
```



Recap: Upper Case String Example

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

```
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;
  B     While         @ }
EndWhile:
```

Recap: Sum Example

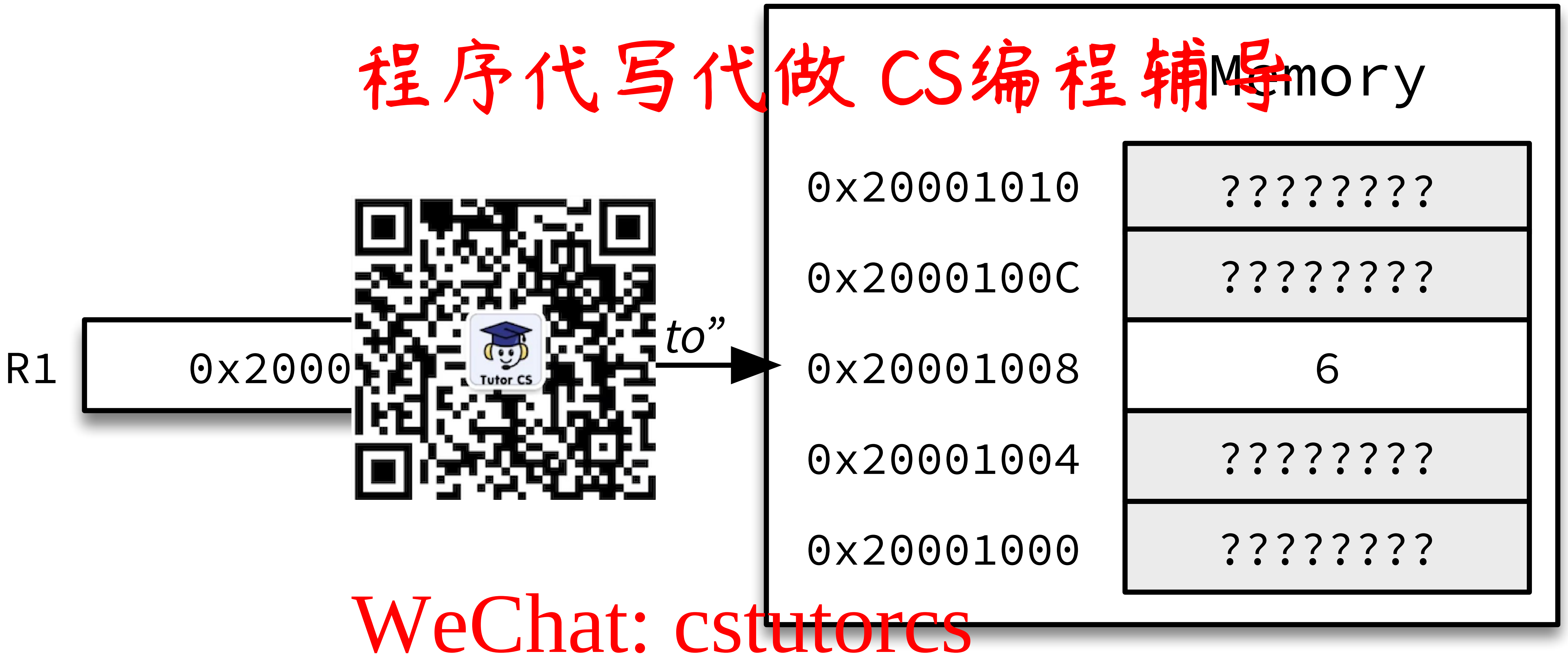
Design and write an assembly language program that will calculate the sum of 10 word-size values stored in memory, beginning at the address in R1. Store the sum in R0.

```
  MOV   R0, #0        @ [illegible];

  MOV   R2, #0        @ i = 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:
```

Addressing Modes: Immediate Offset

Addressing Modes



0x20001010	????????
0x2000100C	????????
0x20001008	6
0x20001004	????????
0x20001000	????????

The syntax **[R1]** is just one of many ways that we can specify the address of the memory location that we want to access using LDR or STR

Remember: [R1] tells the processor to access the value in memory at the address contained in register R1. (We can say that R1 “*points to*” a location in memory.)

The syntax [R1] is an **Addressing Mode**

[R1] is an abbreviated form of [R1, #0] (the #0 is implied if omitted)

The address of the memory location accessed by LDR or STR is called the **Effective Address (EA)**

Immediate Offset Addressing

Addressing Mode Syntax	Operation	Example
`[Rn, #offset]`	Rn + offset	`LDR R0, [R1, #4]`
`[Rn]`	0 *(#0 is assumed)*	`LDR R0, [R1]`

Effective Address is calculated by adding **offset** to the address in the **base register Rn** (note: offset may be negative)

The value in the base register Rn does not change

Example: load three consecutive word-size values from memory into registers R4, R5 and R6, beginning at the address contained in R0

```
LDR     R4, [R0]         @ R4 = word[R0 + 0] (default = 0)
LDR     R5, [R0, #4]     @ R5 = word[R0 + 4]
LDR     R6, [R0, #8]     @ R6 = word[R0 + 8]
```

Addressing Modes: Register Offset

Register Offset Addressing

Addressing Mode Syntax	Operation	Example
[Rn, Rm]	EA = Rn + Rm	LDR R0, [R1, R2]

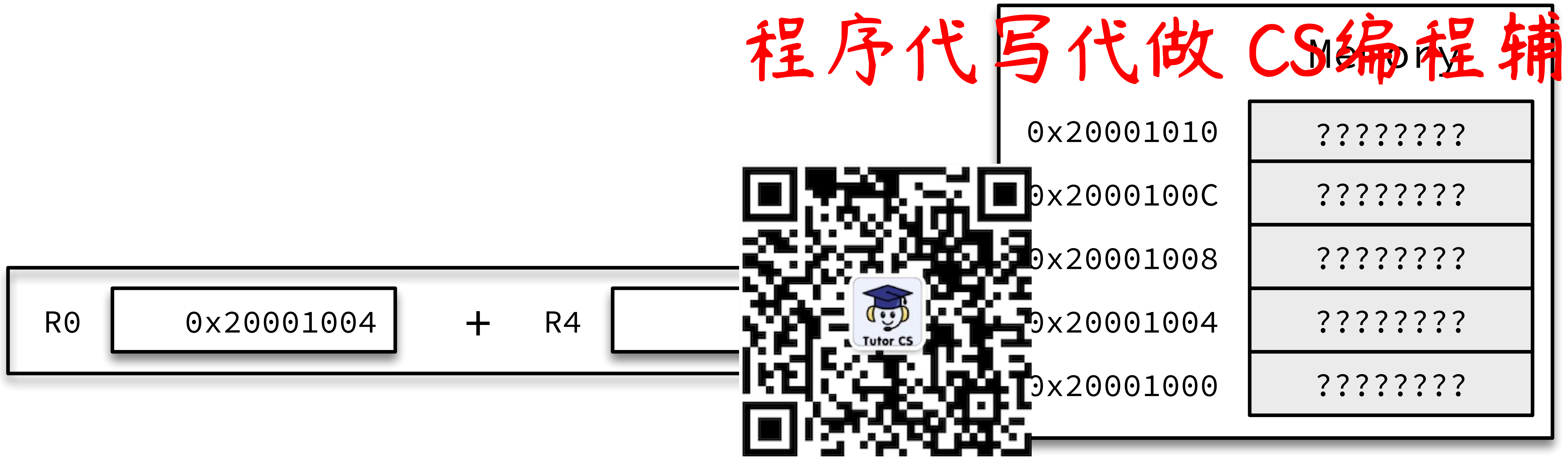
Effective Address is calculated by adding the offset in **Rm** to the address in the base register **Rn**

The values in the base register **Rn** and offset register **Rm** do not change

Example: load three consecutive word values from memory into registers R1, R2 and R3 beginning at the address in R0

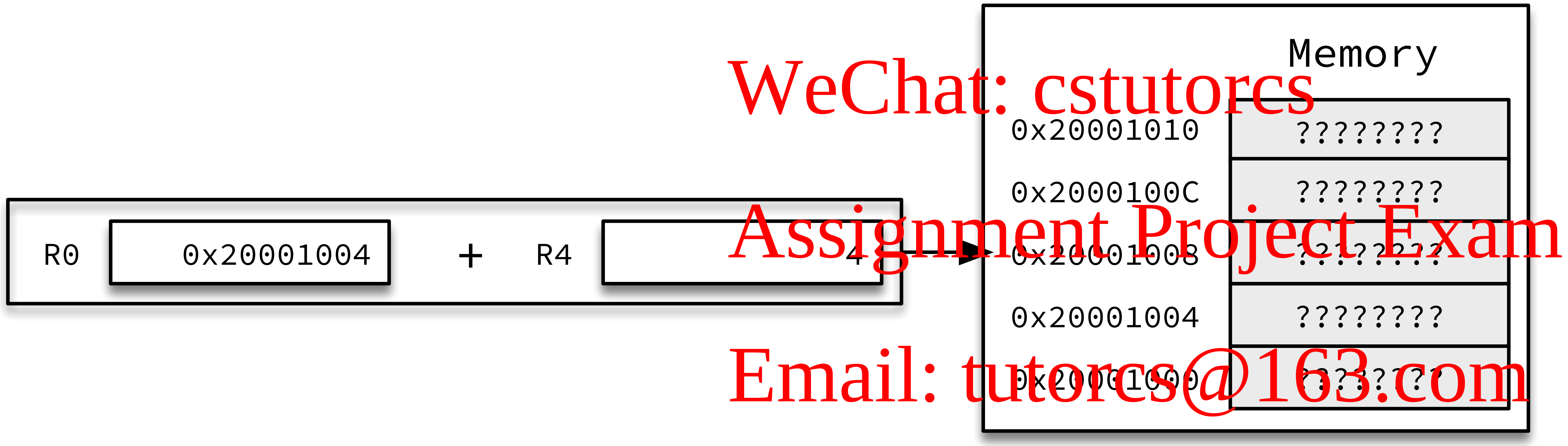
```
LDR     R4, =0          @ Initialise offset register = 0
LDR     R1, [R0, R4]    @ r1 = word[r0 + r4]
ADD     R4, R4, #4      @ r4 = r4 + 4
LDR     R2, [R0, R4]    @ r2 = word[r0 + r4]
ADD     R4, R4, #4      @ r4 = r4 + 4
LDR     R3, [R0, R4]    @ r3 = word[r0 + r4]
```

Register Offset Addressing



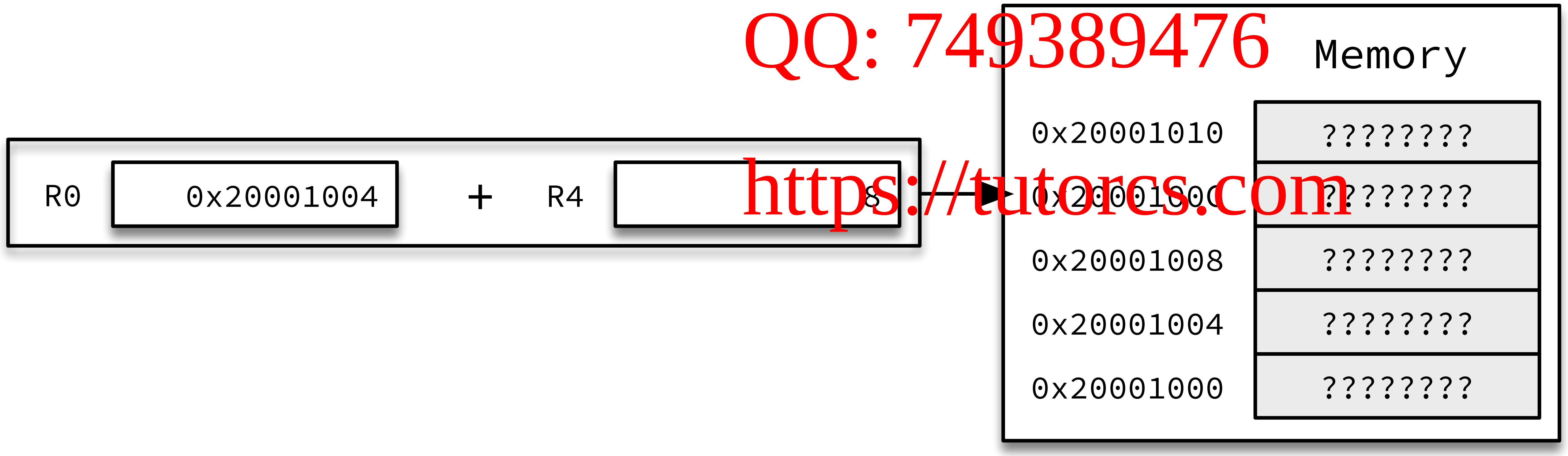
0x20001010	????????
0x2000100C	????????
0x20001008	????????
0x20001004	????????
0x20001000	????????

Loading
1st Word



0x20001010	????????
0x2000100C	????????
0x20001008	????????
0x20001004	????????
0x20001000	????????

Loading
2nd Word



0x20001010	????????
0x2000100C	????????
0x20001008	????????
0x20001004	????????
0x20001000	????????

Loading
3rd Word

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

1.2 – Memory and Addressing Modes (continued)

CSU11022 – Introduction to Computing II

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

UPPER CASE (using Register Offset Addressing)

```
    LDR     R2, =0              @ index = 0

whUpr:
    LDRB    R4, [R1, R2]
    CMP     R4, #0                  ( (char = byte[address + index]) != 0 )
    BEQ     eWhUpr
    CMP     R4, #'a'            @  if (char >= 'a'
    BLO     eIfLwr              @         &&
    CMP     R4, #'z'            @         char <= 'z')
    BHI     eIfLwr              @  {
    BIC     R4, #0x00000020     @    char = char AND NOT 0x00000020
    STRB    R4, [R1, R2]        @    byte[address + index] = char
eIfLwr:                         @  }
    ADD     R2, R2, #1          @  index = index + 1
    B       whUpr               @ }
eWhUpr:                         @

End_Main:
    BX      LR
```

Sum (using Register Offset Addressing)

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

```
    LDR     R0, =0          @ [illegible]
    LDR     R4, =0          @ count = 0
    LDR     R5, =0          @ offset = 0

whSum:
    CMP     R4, #10         @ while (count < 10)
    BHS     eWhSum          @ {
    LDR     R6, [R1, R5]    @   num = word[address + offset]
    ADD     R0, R0, R6      @   sum = sum + num
    ADD     R5, R5, #4      @   offset = offset + 4
    ADD     R4, R4, #1      @   count = count + 1
    B       whSum           @ }
eWhSum:                     @

End_Main:
    BX      LR
```

Addressing Modes: Scaled Register Offset

Scaled Register Offset Addressing

Addressing Mode Syntax	Operation	Example
[Rn, Rm, LSL #shift]	Rn + (Rm x 2^{shift})	LDR R0, [R1, R2, LSL #2]

Effective Address is calculated by adding offset in **Rm**, shifted left by **shift** bits, to the address in the base register **Rn**

The values in the base register **Rn** and offset register **Rm** are not changed

Example: load three consecutive word-size values from memory into registers R1, R2 and R3 beginning at the address in R0

```
LDR     R4, =0                  @ Initialise index register = 0
LDR     R1, [R0, R4, LSL #2]    @ R1 = word[r0 + r4 * 4]
ADD     R4, R4, #1              @ R4 = r4 + 1
LDR     R2, [R0, R4, LSL #2]    @ R2 = word[r0 + r4 * 4]
ADD     R4, r4, #1              @ R4 = r4 + 1
LDR     R3, [R0, R4, LSL #2]    @ R3 = word[r0 + r4 * 4]
```

Sum (using Scaled Register Offset Addressing)

Re-write our program to calculate the sum of 10 word-size values stored in memory, this time using scaled register offset addressing

Allows count to be used f[illegible]!!

```
  LDR     R0, =0                @ sum = 0
  LDR     R4, =0                @ count = 0

whSum:
  CMP     R4, #10               @ while (count < 10)
  BHS     eWhSum                @ {
  LDR     R6, [R1, R4, LSL #2]  @  num = word[address + (count * 4)]
  ADD     R0, R0, R6            @  sum = sum + num
  ADD     R4, R4, #1            @  count = count + 1
  B       whSum                 @ }
eWhSum:                         @

End_Main:
  BX      LR
```

Addressing Modes: Pre- and Post-Indexed Addressing

Pre- and Post-Indexed Addressing

Many programs iterate sequentially through memory (examples?)

Often manifested as an LDR/STR followed by an ADD

```
LDR     R4, [R1]                 word
ADD     R1, R1, #4               ement base address register R1
```

ARM architecture provides a set of addressing modes that incorporate the increment/decrement into the execution of the LDR/STR instruction

Pre-Indexed Addressing	Post-Indexed Addressing
1. Increment / Decrement base address register (Rn) 2. Compute Effective Address	1. Compute Effective Address 2. Increment / Decrement base address register (Rn)

Post-Indexed Addressing Modes

```
LDR     R4, [R1]
ADD     R1, R1, #4
```

⟺

```
LDR     R4, [R1], #4
```

Immediate Offset Addressing

Immediate Post-Indexed Addressing

Syntax: post-indexed addressing modes place the value to be added to the base register Rn **after** the [ ]

Behaviour: (i) the LDR/STR is performed first using the original base register value, (ii) the base register is updated by applying the post-index operation

Modes: Immediate Post-Indexed, Register Post-Indexed, Scaled Register Post-Indexed (LSL #shift)

Pre-Indexed Addressing Modes

```
ADD      R1, R1, #4                LDR      R4, [R1, #4]!
LDR      R4, [R1]          ⟺
```

Immediate Offset Addressing

Immediate Pre-Indexed Addressing

Syntax: pre-indexed addressing modes place the value to be added to the base register Rn **inside** the [ ]

An exclamation mark ! after the [ ] distinguishes pre-indexed addressing from immediate offset addressing (e.g. **[R1, #4]**), which doesn't modify R1

Behaviour: (i) the base register is updated by applying the pre-index operation, (ii) the LDR/STR is performed using the updated base register value

Modes: Immediate Pre-Indexed, Register Pre-Indexed, Scaled Register Pre-Indexed (LSL #shift)

UPPER CASE (using Immediate Post-Indexed Addressing)

```
whUpr:
  LDRB    R4, [R1], #1                  byte[address++]
  CMP     R4, #0                       ( char != 0 )
  BEQ     eWhUpr
  CMP     R4, #'a'           @   if (char >= 'a'
  BLO     eIfLwr             @        &&
  CMP     R4, #'z'           @        char <= 'z')
  BHI     eIfLwr             @  {
  BIC     R4, #0x00000020    @    char = char AND NOT 0x00000020
  STRB    R4, [R1, #-1]      @    byte[address - 1] = char
eIfLwr:                      @  }
  B       whUpr              @ }
eWhUpr:                      @

End_Main:
  BX      LR
```

Sum (using Immediate Post-Indexed Addressing)

```
    LDR     R0, =0          @
    LDR     R4, =0          @          0

whSum:
    CMP     R4, #10         @ while (count < 10)
    BHS     eWhSum          @ {
    LDR     R6, [R1], #4    @   num = word[address]; address = address + 4;
    ADD     R0, R0, R6      @   sum = sum + num
    ADD     R4, R4, #1      @   count = count + 1
    B       whSum           @ }
eWhSum:                     @

End_Main:
    BX      LR
```

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

1.3 – Arrays

CSU11022 – Introduction to Computing II

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

Single-Dimensional Arrays

Arrays

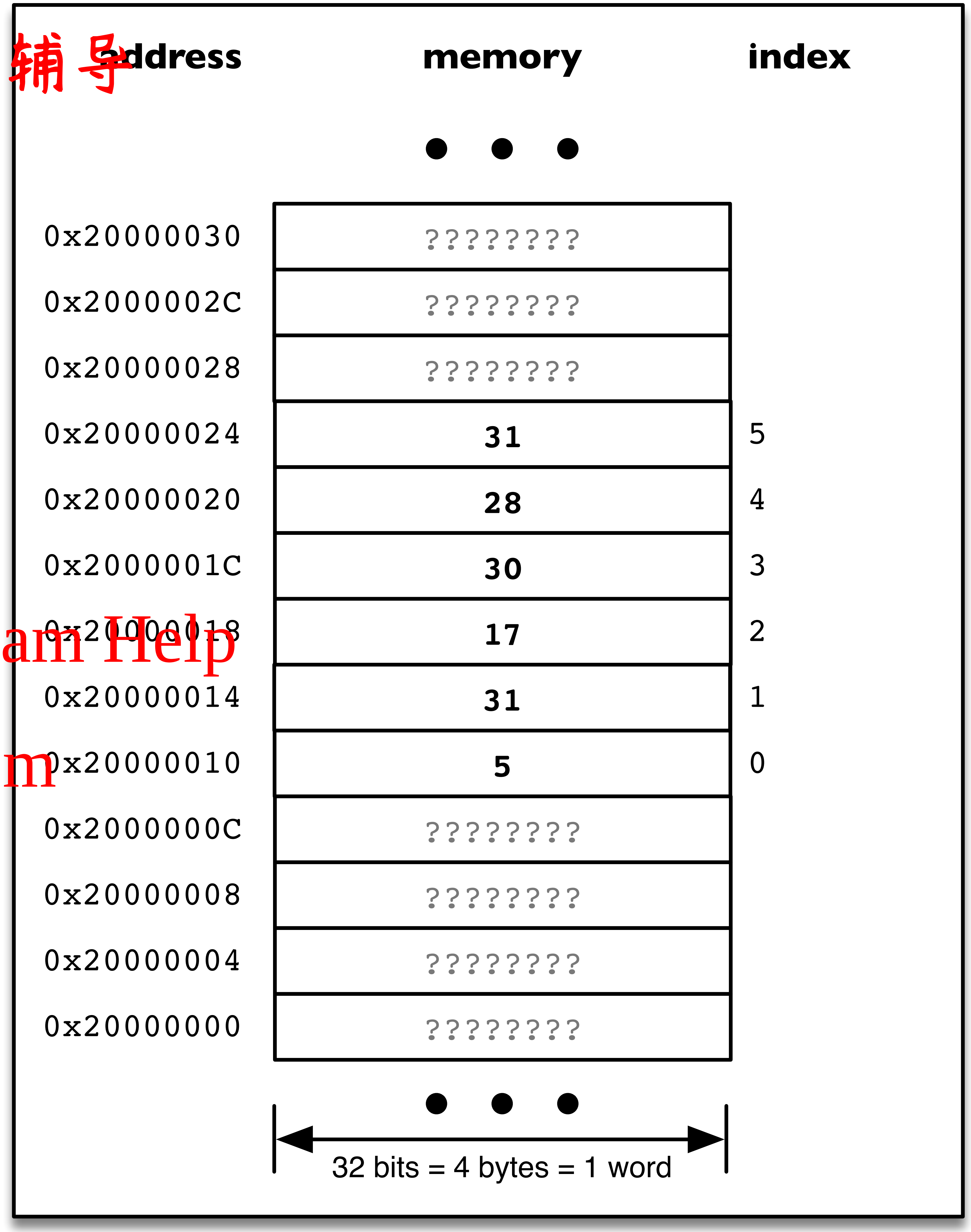
Nothing new here … you have already been using arrays!

Array – an ordered collec homogeneous elements stored sequentially in memory

e.g. integers, ASCII characters

“Homogeneous elements”? (for us, at least with respect to size)

“Dimension” number of elements in array



address	memory	index
	• • •	
0x20000030	????????	
0x2000002C	????????	
0x20000028	????????	
0x20000024	31	5
0x20000020	28	4
0x2000001C	30	3
0x20000018	17	2
0x20000014	31	1
0x20000010	5	0
0x2000000C	????????	
0x20000008	????????	
0x20000004	????????	
0x20000000	????????	
	• • •	

Accessing (loading/storing) Array Elements

Step 1: translate array index into byte offset from start address of array in memory

<byte offset> = <index> × <elem size>

Step 2: add byte offset to array base address to access element

<address> = <array start address> + <byte offset>

Example: retrieve the 4th element (index 3) of an array of words stored in memory beginning at the address in R4

Efficient implementation of random access using **Scaled Register Offset** addressing mode:

```
LDR     R5, =3                  @ index = 3 (4th element)
LDR     R6, [R4, R5, LSL #2]    @ elem = word[arr + (index * 4)]
```

Array Access Example – Sum

```
    LDR       R0, =0                @ sum = 0
    LDR       R4, =0                @ index = 0

whSum:
    CMP       R4, #10                   ile (index < 10)
    BHS       eWhSum
    LDR       R6, [R1, R4, LSL          um = array[index]
    ADD       R0, R0, R6            @   sum = sum + num
    ADD       R4, R4, #1            @   index = index + 1
    B         whSum                 @ }
eWhSum:                             @

End_Main:
    BX        LR
```

Déjà Vu? The pseudo-code comments have changed to use array syntax but the program is identical to our version of Sum with scaled register offset

The register offset is our array index, which is scaled by the size of a single array element to give us the memory address offset for the element we want

You’re ready to attempt Assignment 1 which will be released at the start of next week

Before attempting Assignment 1, you should complete the “Bubblesort” exercise

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

1.4 – Arrays (continued)

CSU11022 – Introduction to Computing II

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

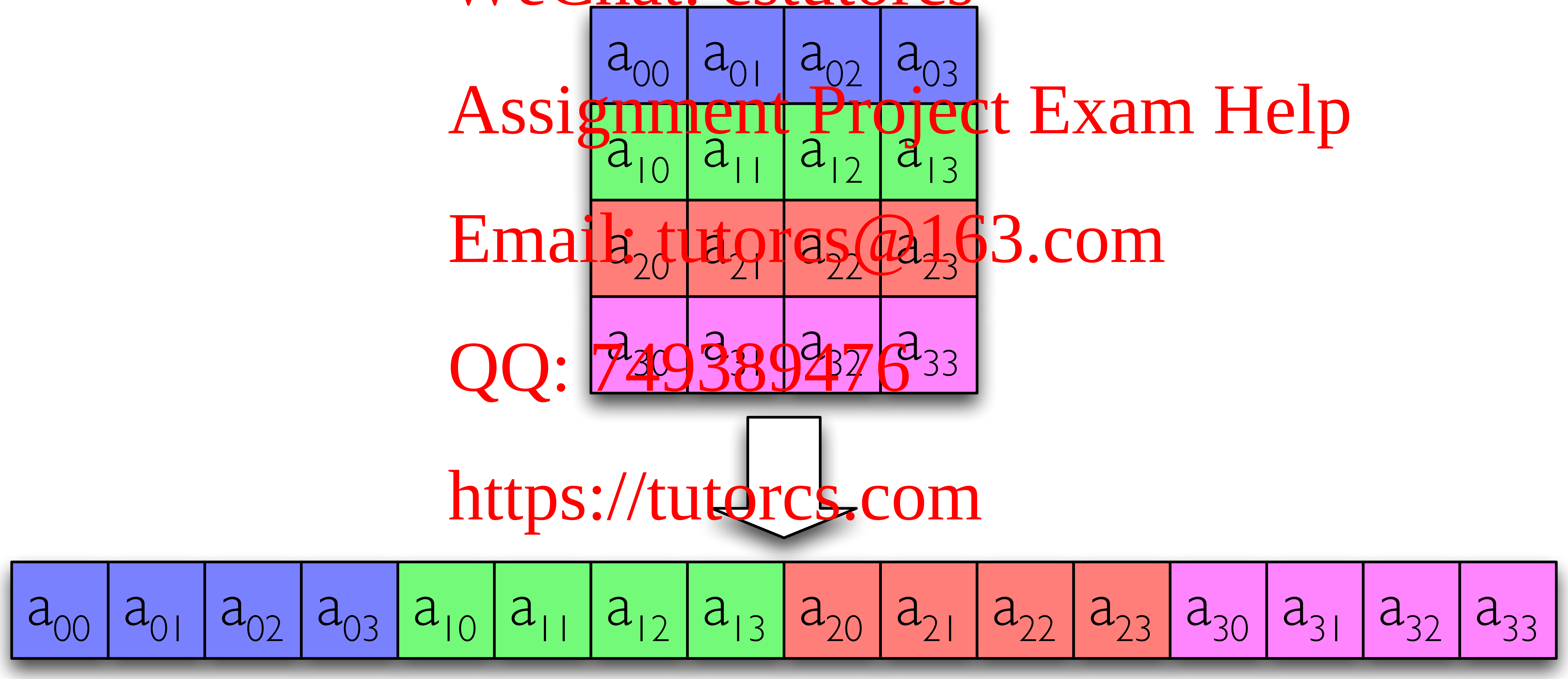
Multi-Dimensional Arrays

2D Arrays

Arrays can have more than one dimension

e.g. a two-dimensional array is analogous to a table containing elements arranged in rows and columns

Stored in memory by mapping the 2D array into 1D memory, e.g.



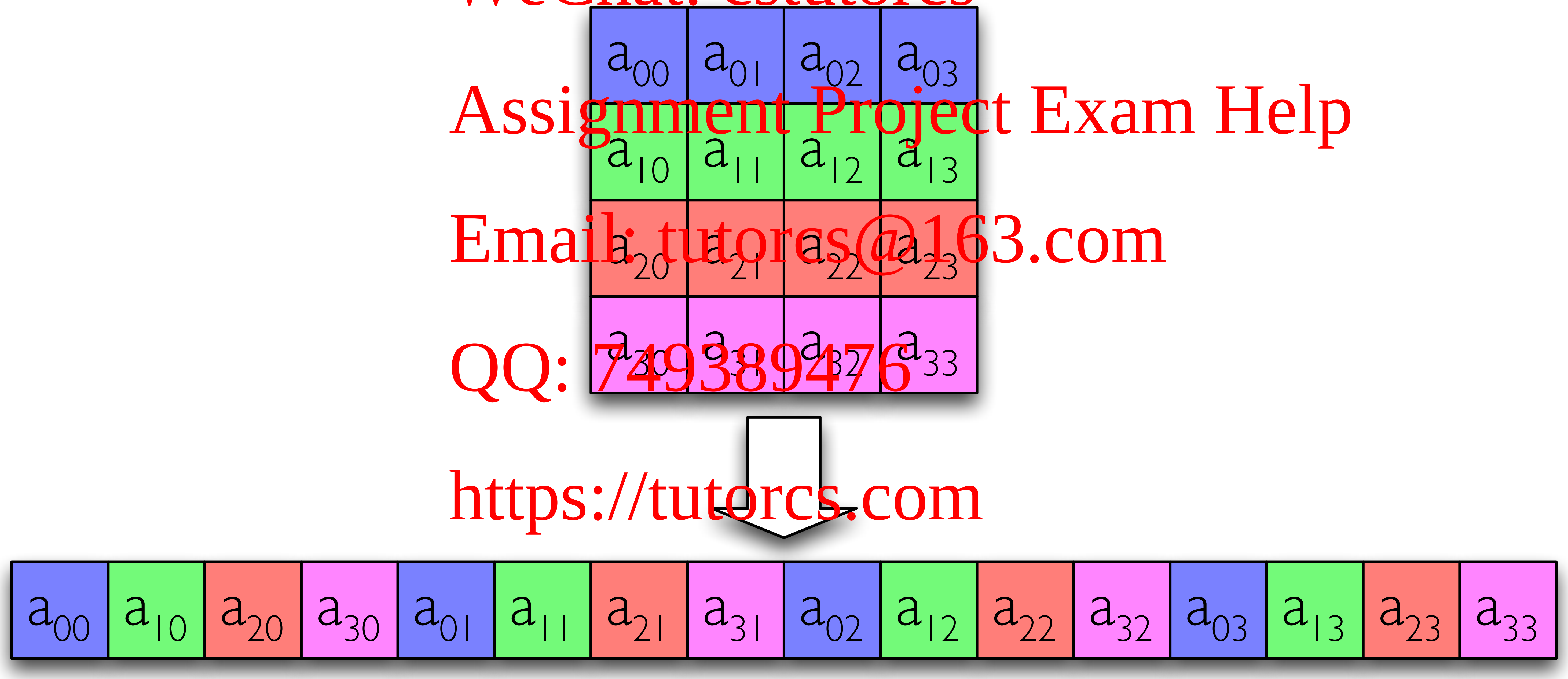
a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

a_{00}	a_{01}	a_{02}	a_{03}	a_{10}	a_{11}	a_{12}	a_{13}	a_{20}	a_{21}	a_{22}	a_{23}	a_{30}	a_{31}	a_{32}	a_{33}

2D Arrays

Row-major order: 2D array is stored in memory by storing each **row** contiguously in memory

Column-major order: 2D [illegible] tored in memory by storing each **column** contiguously in m[illegible] are – **row-major** is the assumed norm)



a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

a_{00}	a_{10}	a_{20}	a_{30}	a_{01}	a_{11}	a_{21}	a_{31}	a_{02}	a_{12}	a_{22}	a_{32}	a_{03}	a_{13}	a_{23}	a_{33}

2D Array Example

2D array declared in memory

```
testMatrix:
    .word    6, 3, 8, 2, 5, 2   @ row 0
    .word    0, 7, 2, 8, 5, 7   @ row 1
    .word    0, 0, 7, 4, 2, 6   @ row 2
    .word    0, 0, 0, 2, 9, 5   @ row 3
    .word    0, 0, 0, 0, 5, 8   @ row 4
    .word    0, 0, 0, 0, 0, 3   @ row 5
```

… or equivalently … but not very clearly … or recommended …

```
testMatrix:
    .word    6, 3, 8, 2, 5, 2, 0, 7, 2, 8
    .word    5, 7, 0, 0, 7, 4, 2, 6, 0, 0
    .word    0, 2, 9, 5, 0, 0, 0, 0, 5, 8
    .word    0, 0, 0, 0, 0, 3
```

Accessing 2D Array Elements

Step 1: translate 2D array index into 1D array index

<index> **> × <row_size>) + <col>**

Step 2: translate 1D arr into byte offset from start address of array in memory

<byte_offset> = <index> × <elem_size>

Step 3: add byte offset to array base address to access element

<address> = <array_base_address> + <byte_offset>

Example: retrieve the element at the 4th row and 3rd column of a 2D array of words with 6 rows and 8 columns – **array[3][2]**

Step 1 is new

Steps 2 & 3 are the same as those for a 1-D array …

… because our 2-D array is just a different interpretation of a 1-D array!

2D Array Access Example

Example: retrieve the element at the 4th row and 3rd column of a 2D array of words – **array[3][2]**

The array starts in memo address in R4. The number of rows in in R5 and the number of is in R6.

```
; looking for array[3][2] (4th row, 3rd column)

 LDR     r1, =3                   @ row = 3
 LDR     r2, =2                   @ col = 2

 ; <byte offset> = ((row * <row_size>) + col) * <elem_size>

 MUL     r7, r1, r6               ; index = row * row_size
 ADD     r7, r7, r2               ; index = index + col
 LDR     r0, [r4, r7, LSL #2]     ; elem = word[ array + (index * elem_size) ]
```

Upper Triangular Matrix

An upper triangular matrix is a square matrix in which all entries below the main diagonal (top-left to bottom right) are zero. Design and write an ARM Assembly Language program to determine if a matrix stored in memory is an Upper Triangular matrix.

1	2	3	4
0	5	6	7
0	0	8	9
0	0	0	10

Assume the matrix is stored in memory at the address in R1 and the number of rows and columns is stored in R2 (it's a square matrix!)

Store 1 in R0 if it is Upper Triangular and 0 in R0 if it is not.

```
result = TRUE;
for (r = 1; r < N; r++)
{
  for (c = 0; c < r; c++)
  {
    elem = matrix[r][c];
    if (elem != 0)
    {
      result = FALSE;
    }
  }
}
```

My solution is on the next slide.

Before looking at it, you should **practice writing the ARM Assembly Language Program yourself**.

You can submit your solution to submitty.scss.tcd.ie for practice.

Upper Triangular Matrix

```
    MOV     R0, #1                  @ result = TRUE;
                                    @
    MOV     R4, #1                  @ for (r = 1; r < N; r++)
whR:                                @
    CMP     R4, R2                  @
    BHS     ewhR                    @
                                    @
    MOV     R5, #0                  @   [illegible]0; c < r; c++)
whC:                                @   {
    CMP     R5, R4                  @
    BHS     ewhC                    @
                                    @
    MUL     R6, R4, R2              @     elem = matrix[r][c];
    ADD     R6, R6, R5              @
    LDR     R7, [R1, R6, LSL #2]    @
                                    @
    CMP     R7, #0                  @     if (elem != 0)
    BEQ     endifz                  @     {
    MOV     R0, #0                  @       result = false;
endifz:                             @     }
    ADD     R5, R5, #1              @
    B       whC                     @   }
ewhC:                               @
    ADD     R4, R4, #1              @
    B       whR                     @ }
ewhR:
```

Multi-dimension arrays

e.g. a 3D array of size DZ×

In general, the index of element a[z][y][x] is:

index = ((z × DY × DX) + (y × DX) + x)

e.g. a 4D array of size DWxDZ×DY×DX

In general, the index of element a[w][z][y][x] is:

index = ((w × DZ × DY × DX) + (z × DY × DX) + (y × DX) + x)