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

01 – Memory and Addressing Modes

CS1022 – Introduction to Computing II

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

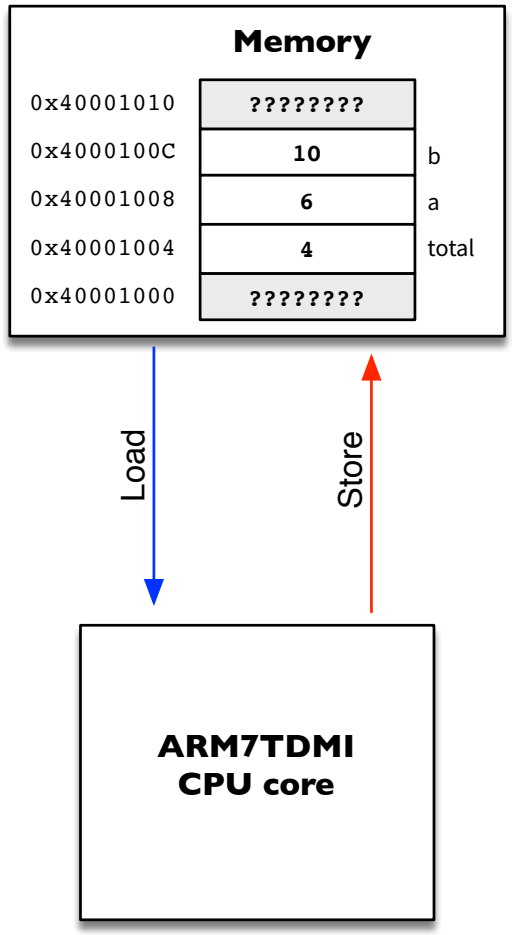
LDR and STR

LDR and STR Reviewed

How many memory accesses are required to compute

$$\textbf{total} = \textbf{total} + (\textbf{a} \times \textbf{b})$$

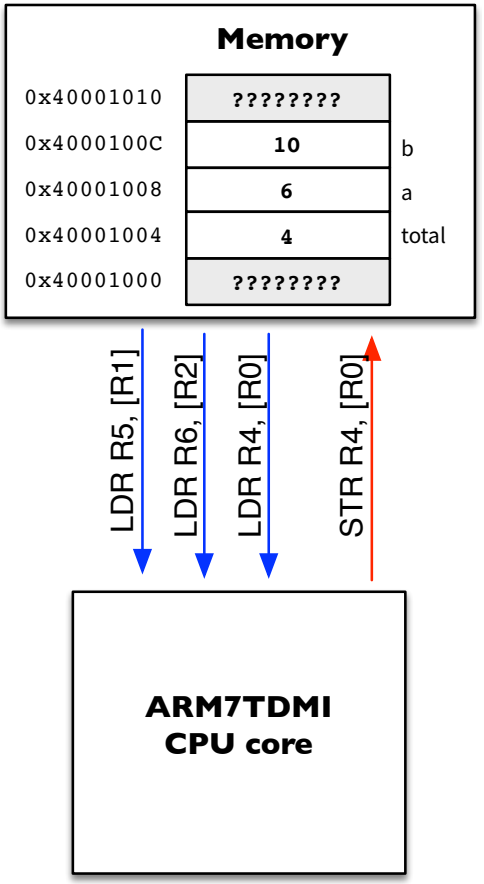
where **total**, **a** and **b** are stored in memory at the addresses contained in **R0**, **R1** and **R2** respectively?



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
```



Assembler Directives for Memory Layout

AREA

Declares a code or data section and defines its attributes (CODE, DATA, READONLY, READWRITE, etc.)

Section – an indivisible sequence of instructions and/or region of memory used to store data that is arranged in memory by the linker

SPACE

Tells the assembler to set aside space in memory to contain data

Does not initialise anything

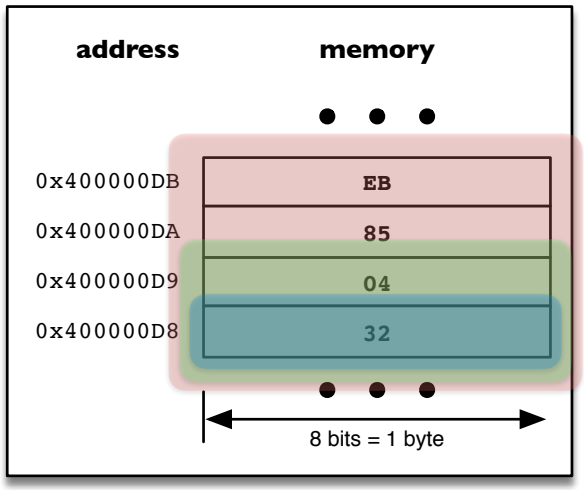
(We would have to write code to initialise data that we want to modify later)

DCB, DCW, DCD

Initialises ROM with constant Byte-, Halfword- or Word-size values

```
    AREA    globals, DATA, READWRITE

L0  SPACE   4                   ; will be placed at 0x400000000


    AREA    RESET, CODE, READONLY
    ENTRY

    LDR     R0, =L0
    LDR     R1, =L1
    LDR     R2, =L2
    LDR     R4, [R1]
    LDR     R5, [R2]
    ADD     R4, R4, R5
    STR     R4, [R0]

STOP
    B       STOP

L1  DCD     6                   ; will be placed after our code
L2  DCD     7

    END
```

Load / Store Bytes, Half-words and Words

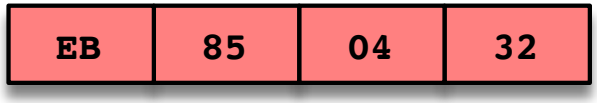


Byte, **half-word** and **word** at address 0xA00000D8

```
LDR  r0, =0x400000D8
LDRB r1, [r0]
```

```
LDR  r0, =0x400000D8
LDRH r1, [r0]
```

```
LDR  r0, =0x400000D8
LDR  r1, [r0]
```



Alignment

ARM7TDMI expects all memory accesses to be **aligned**

Examples:

	Address Examples
Word aligned	`0x00000000, 0x00001008, 0x4100000C`
Not word aligned	`0x00000001, 0x00001006, 0x4100000F`
Half-word aligned	`0x00000000, 0x00001002, 0x4100000A`
Not half-word aligned	`0x00000003, 0x00001001, 0x4100000B`

See ARM Architecture Reference Manual Section A2.8

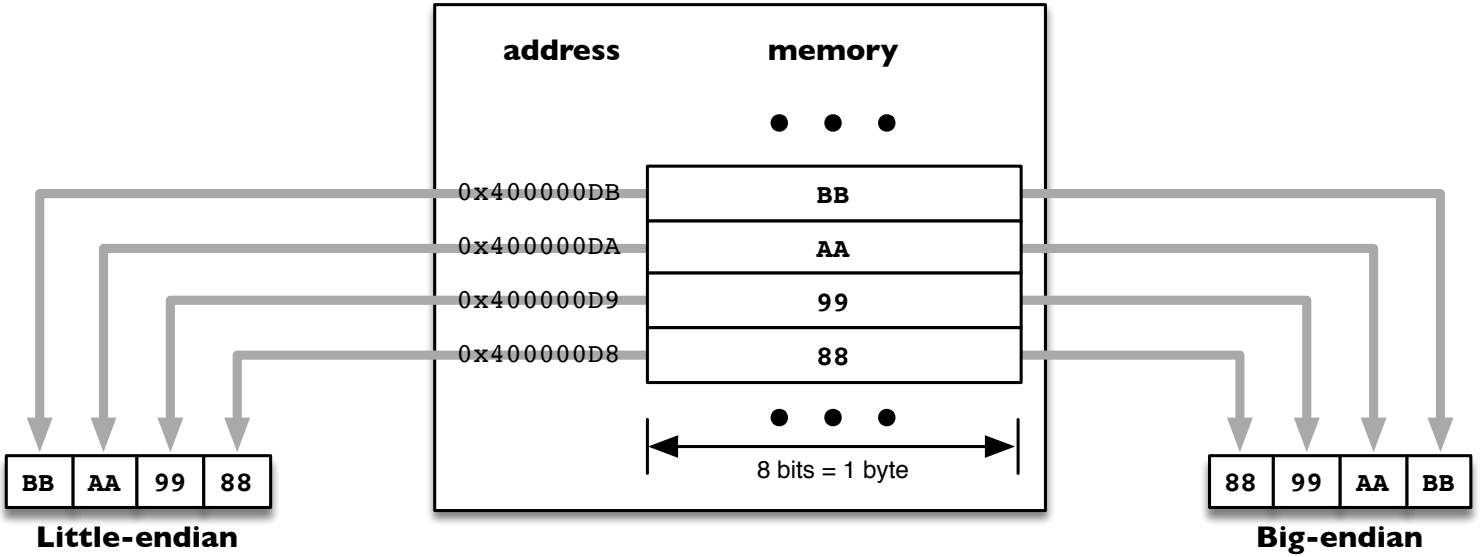
Unaligned accesses are permitted but the result is unlikely to be what was intended

Unaligned accesses are supported by later ARM architecture versions

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



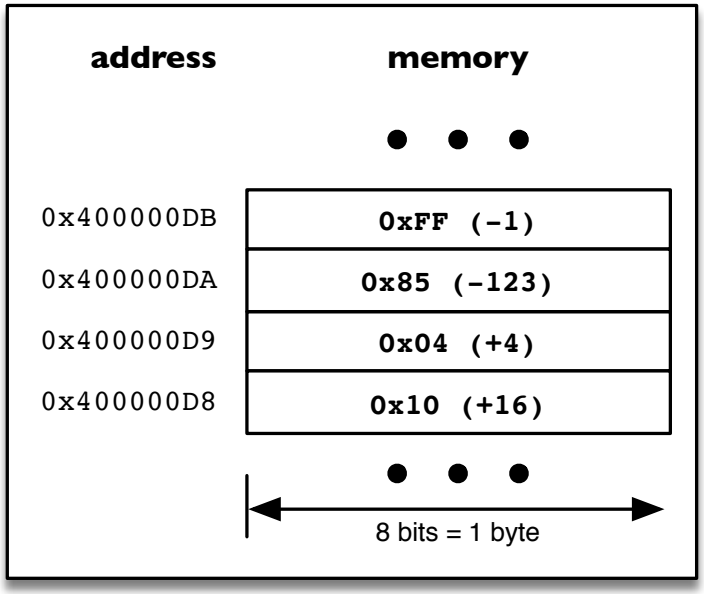
Loading and storing signed values

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, =0x400000D8
LDRB    r1, [r0]
```

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



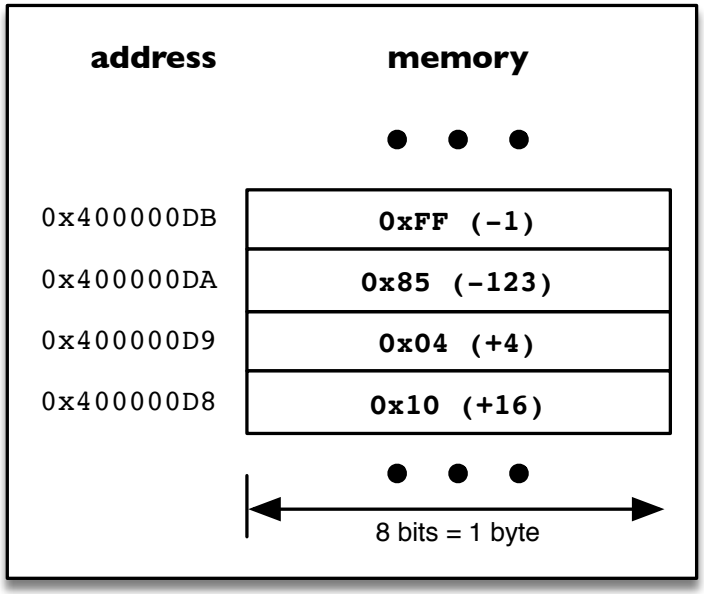
Loading and storing signed values

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, =0x400000DB
LDRB    r1, [r0]
```

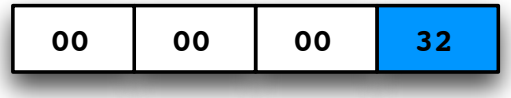
(a) +1
(b) +255
(c) -1
(d) -255



Loading Signed Bytes and Half-words

Sign extension performed when loading signed bytes or half-words to facilitate correct subsequent 32-bit signed arithmetic

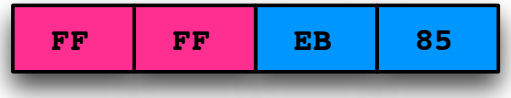
```
LDR    r0, =0x400000D8
LDRSB  r1, [r0]
```

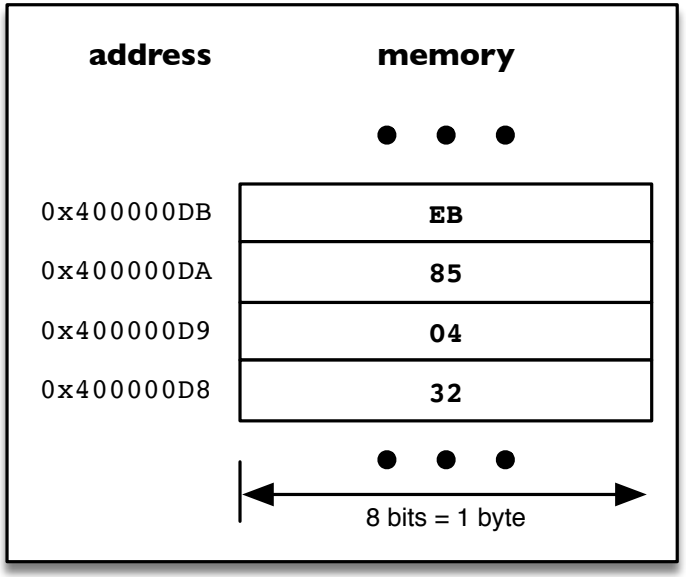



```
LDR   r0, =0x400000D8
LDRSH r1, [r0]
```


```
LDR    r0, =0x400000DA
LDRSB  r1, [r0]
```


```
LDR   r0, =0x400000DA
LDRSH r1, [r0]
```





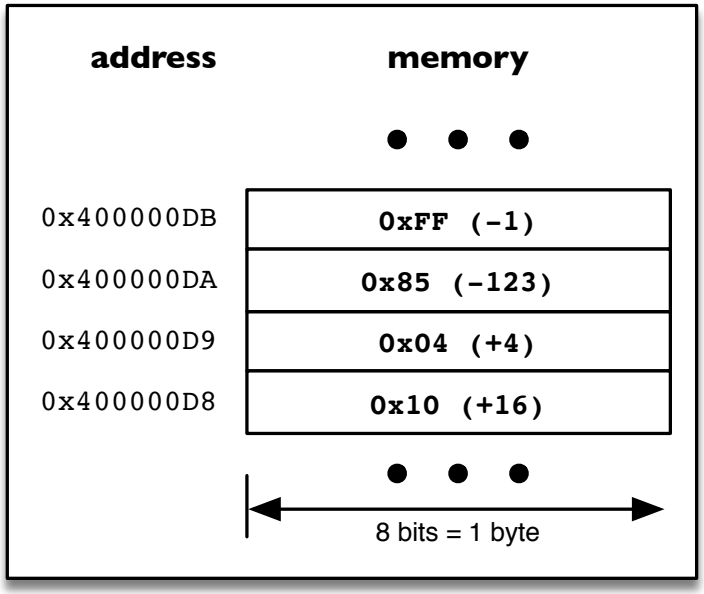
Loading and storing signed values

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, =0x400000DA
LDRB   r1, [r0]
```

(a) +123
(b) -123
(c) +133
(d) -252



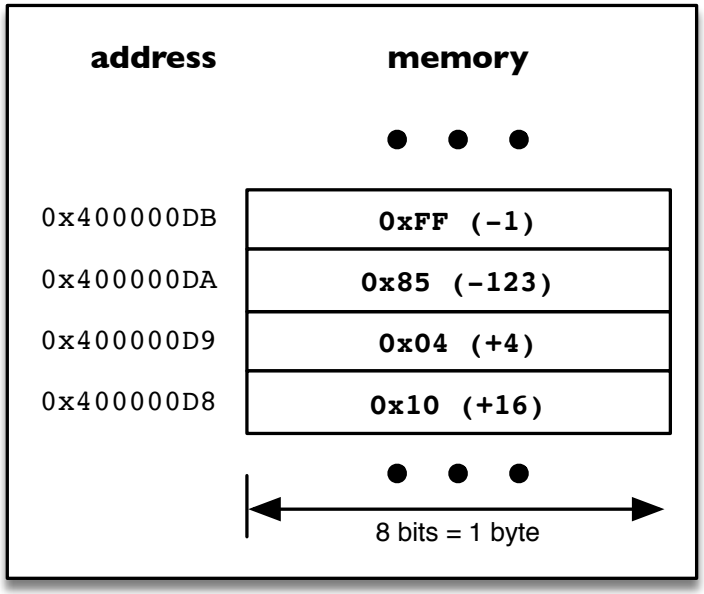
Loading and storing signed values

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?

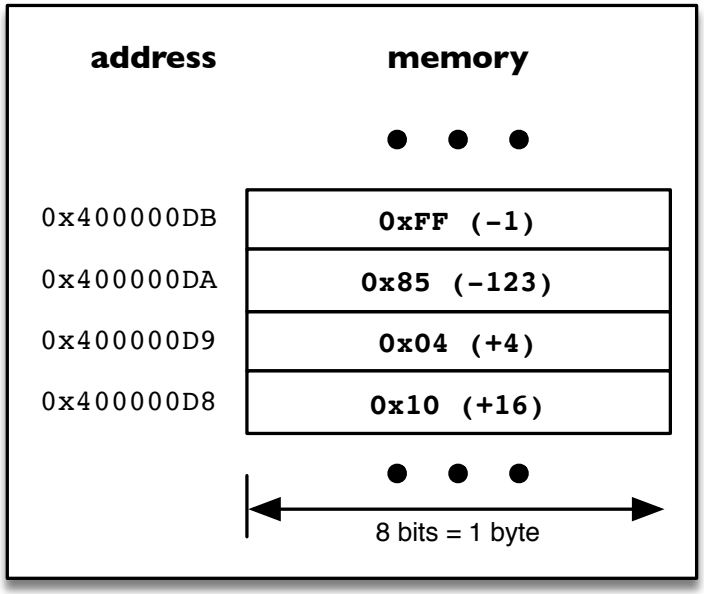
```
LDR    r0, =0x400000DA
LDRSB  r1, [r0]
```

(a) +123
(b) -123
(c) +133
(d) -252



Loading and storing signed values

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



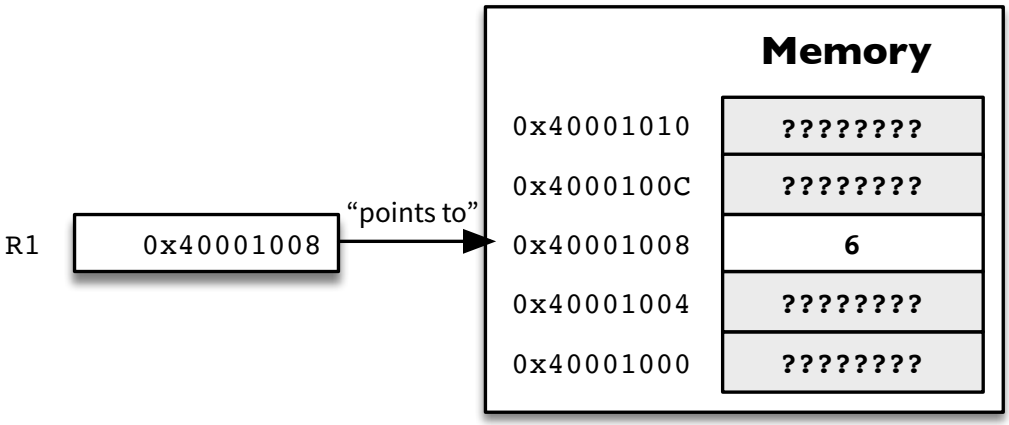
After executing the pair of instructions below, what is the correct signed decimal interpretation of the value loaded in R1?

```
LDR    r0, =0x400000D9
LDRSB  r1, [r0]
```

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

Addressing Modes

Addressing Modes



The syntax **`[R1]`** is just one of many ways that we can specify the the memory address 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

The syntax `[R1]` corresponds to an **Addressing Mode**

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

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

Immediate Offset Addressing

Addressing Mode Syntax	Operation	Example
`[Rn, #offset]`	EA = Rn + offset	`LDR R0, [R1, #4]`
`[Rn]`	EA = 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)

Base register ***Rn*** is not changed

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

```
LDR   R0, =label    ; Initialise base register
LDR   R4, [R0]      ; R4 = Memory.word[R0 + 0] (default = 0)
LDR   R5, [R0, #4]  ; R5 = Memory.word[R0 + 4]
LDR   R6, [R0, #8]  ; R6 = Memory.word[R0 + 8]
```

Register Offset Addressing

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

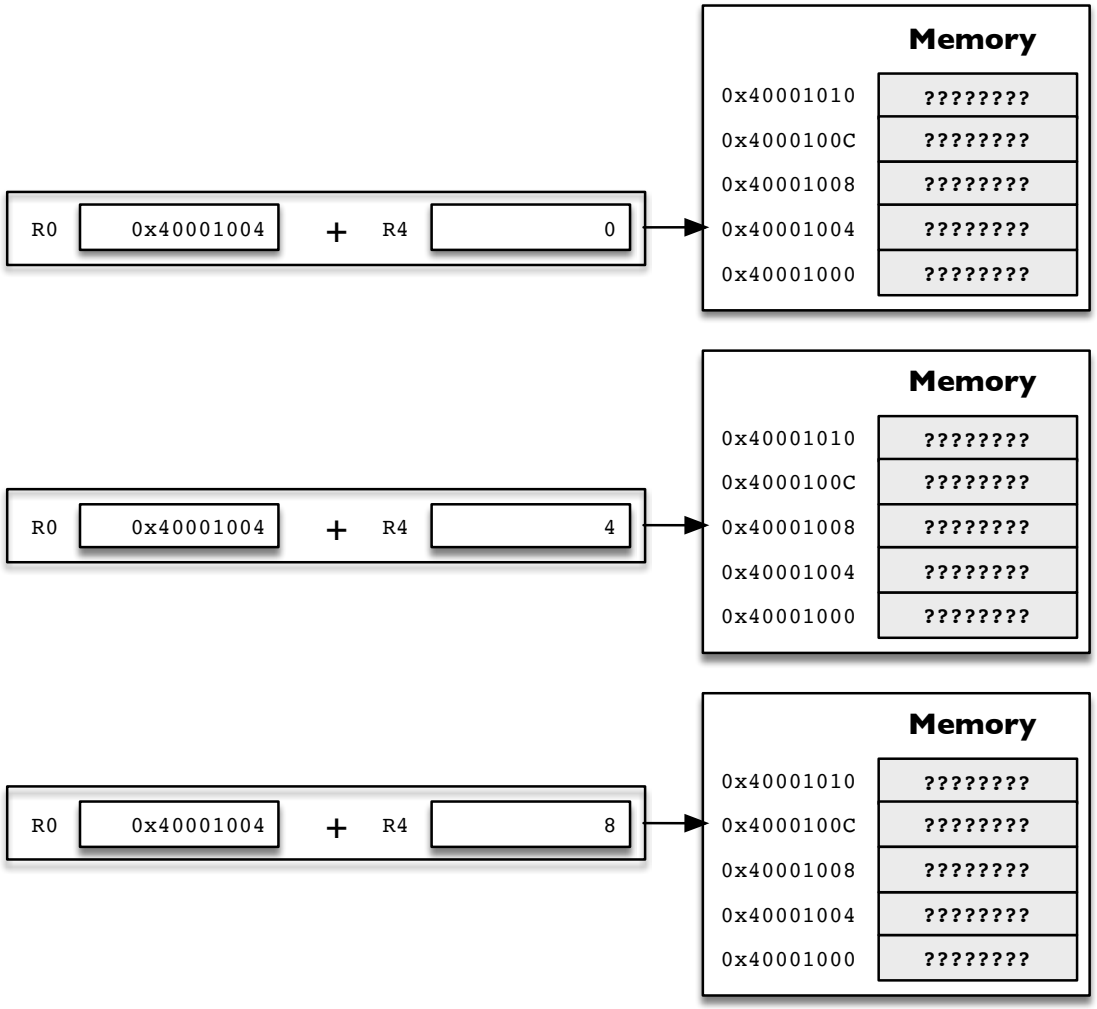
Effective Address is calculated by adding offset in ***Rm*** to the address in the base register ***Rn***

Base register ***Rn*** and offset register ***Rm*** are not changed

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

```
LDR    r0, =label      ; Initialise base register
LDR    r4, =0          ; Initialise offset register = 0
LDR    r1, [r0, r4]    ; r1 = Memory.word[r0 + r4]
ADD    r4, r4, #4      ; r4 = r4 + 4
LDR    r2, [r0, r4]    ; r2 = Memory.word[r0 + r4]
ADD    r4, r4, #4      ; r4 = r4 + 4
LDR    r3, [r0, r4]    ; r3 = Memory.word[r0 + r4]
```

Register Offset Addressing



Example: Convert to UPPER CASE (using Register Offset Addressing)


```
   LDR   r1, =TESTSTR     ; address = TESTSTR
   LDR   r2, =0           ; index = 0

   LDRB  r0, [r1, r2]     ; char = Memory.Byte[address + index]
L1
   CMP   r0, #0           ; while ( char != 0 )
   BEQ   L3               ; {
   CMP   r0, #'a'         ;  if (char >= 'a'
   BLO   L2               ;   &&
   CMP   r0, #'z'         ;   char <= 'z')
   BHI   L2               ;  {
   BIC   r0, #0x00000020  ;   char = char AND NOT 0x00000020
   STRB  r0, [r1, r2]     ;   Memory.byte[address + index] = char
L2                        ;  }
   ADD   r2, r2, #1       ;  index = index + 1
   LDRB  r0, [r1, r2]     ;  char = Memory.Byte[address + index]
   B     L1               ; }
L3                        ;
```

Example: Convert to UPPER CASE (using Register Offset Addressing)


```
   LDR   r1, =TESTSTR     ; address = TESTSTR
   LDR   r2, =0           ; index = 0

L1
   LDRB  r0, [r1, r2]     ; while ( (char = Memory.Byte[address + index])
   CMP   r0, #0           ;               != 0 )
   BEQ   L3               ; {
   CMP   r0, #'a'         ;  if (char >= 'a'
   BLO   L2               ;   &&
   CMP   r0, #'z'         ;   char <= 'z')
   BHI   L2               ;  {
   BIC   r0, #0x00000020  ;   char = char AND NOT 0x00000020
   STRB  r0, [r1, r2]     ;   Memory.byte[address + index] = char
L2                        ;  }
   ADD   r2, r2, #1       ;  index = index + 1
   LDRB  r0, [r1, r2]     ;  char = Memory.Byte[address + index]
   B     L1               ; }
L3                        ;
```

By moving the label whStr to include the top LDRB, we can eliminate the bottom LDRB. We will use this construct from now on.

pseudo-code in blue evaluates to the newly loaded value of char (i.e. just like Java, C, etc.!)

no longer needed!

LDR and STR Addressing Modes

Mode	Pseudo-code Operation *	Example
Offset modes – the base address register *is not* modified		
[Rn]	EA = Rn	`LDR R0, [R1]`
[Rn, #offset]	EA = Rn + offset	`LDR R0, [R1, #4]`
[Rn, Rm]	EA = Rn + Rm	`LDR R0, [R1, R2]`
[Rn, Rm, LSL #shift]	EA = Rn + (Rm << shift)	`LDR R0, [R1, R2, LSL #2]`
[Rn, Rm, LSR #shift]	EA = Rn + (Rm >> shift)	`LDR R0, [R1, R2, LSR #2]`
Pre-indexed modes – the base address register *is* modified *before* accessing memory		
[Rn, #offset]!	Rn = Rn + offset EA = Rn	`LDR R0, [R1, #4]!`
[Rn, Rm]!	Rn = Rn + Rm EA = Rn	`LDR R0, [R1, R2]!`
[Rn, Rm, LSL #shift]!	Rn = Rn + (Rm << shift) EA = Rn	`LDR R0, [R1, R2, LSL #2]!`
[Rn, Rm, LSR #shift]!	Rn = Rn + (Rm >> shift) EA = Rn	`LDR R0, [R1, R2, LSR #2]!`
Post-indexed modes – the base address register *is* modified *after* accessing memory		
[Rn], #offset	EA = Rn Rn = Rn + offset	`LDR R0, [R1], #4`
[Rn], Rm	EA = Rn Rn = Rn + Rm	`LDR R0, [R1], R2`
[Rn], Rm, LSL #shift	EA = Rn Rn = Rn + (Rm << shift)	`LDR R0, [R1], R2, LSL #2`
[Rn], Rm, LSR #shift	EA = Rn Rn = Rn + (Rm >> shift)	`LDR R0, [R1], R2, LSR #2`

***** EA is the *Effective Address*, which is the memory address to which the load or store operation is applied.

Note: only a subset of the above addressing modes can be used to load or store halfwords, signed-halfwords or signed-bytes. See ARM Architecture Reference Manual section A5.3.

STM and LDM – STore and LoaD Multiple

Instruction	Example	
Base register Rn is not modified (no !)		
STM*mode* Rn, {list}	`STMIA R12, {R0-R3}`	Store the contents of R0-R3 in memory at the address contained in R12
LDM*mode* Rn, {list}	`LDMIA R12, {R5,R7,R10}`	Load R5, R7 and R10 with the contents of memory at the address contained in R12
Base register Rn is modified (Rn!)		
STM*mode* Rn!, {list}	`STMFD SP!, {R4-R12,R14}`	Push R4-R12 and R14 on to the system stack, updating the system stack pointer
LDM*mode* Rn!, {list}	`LDMFD SP!, {R4-R12,R14}`	Pop 10 words off the top of the system stack into R4-R12 and R14, updating the system stack pointer

mode

STM – STore Multiple	
Instruction	Stack-Oriented Synonym
STMDB (decrement before)	**STMFD / PUSH** (full descending)
STMIB (increment before)	**STMFA** (full ascending)
STMDA (decrement after)	**STMED** (empty decending)
STMIA (increment after)	**STMEA** (empty ascending)

LDM – LoaD Multiple	
Instruction	Stack-Oriented Synonym
LDMIA (increment after)	**LDMFD / POP** (full descending)
LDMDA (decrement after)	**LDMFA** (full ascending)
LDMIB (increment before)	**LDMED** (empty decending)
LDMDB (decrement before)	**LDMEA** (empty ascending)

Example: 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

```
   LDR    R1, =TESTDATA  ; address = address of first word-wize value
   LDR    R0, =0         ; sum = 0
   LDR    R4, =0         ; count = 0
   LDR    R5, =0         ; offset = 0

L1
   CMP    R4, #10        ; while (count < 10)
   BHS    L2             ; {
   LDR    R6, [R1, R5]   ;  num = Memory.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      L1             ; }
L2                       ;
```

Scaled Register Offset Addressing

Addressing Mode Syntax	Operation	Example
`[Rn, Rm, LSL #shift]`	EA = 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***

Base register ***Rn*** and offset register ***Rm*** are not changed

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

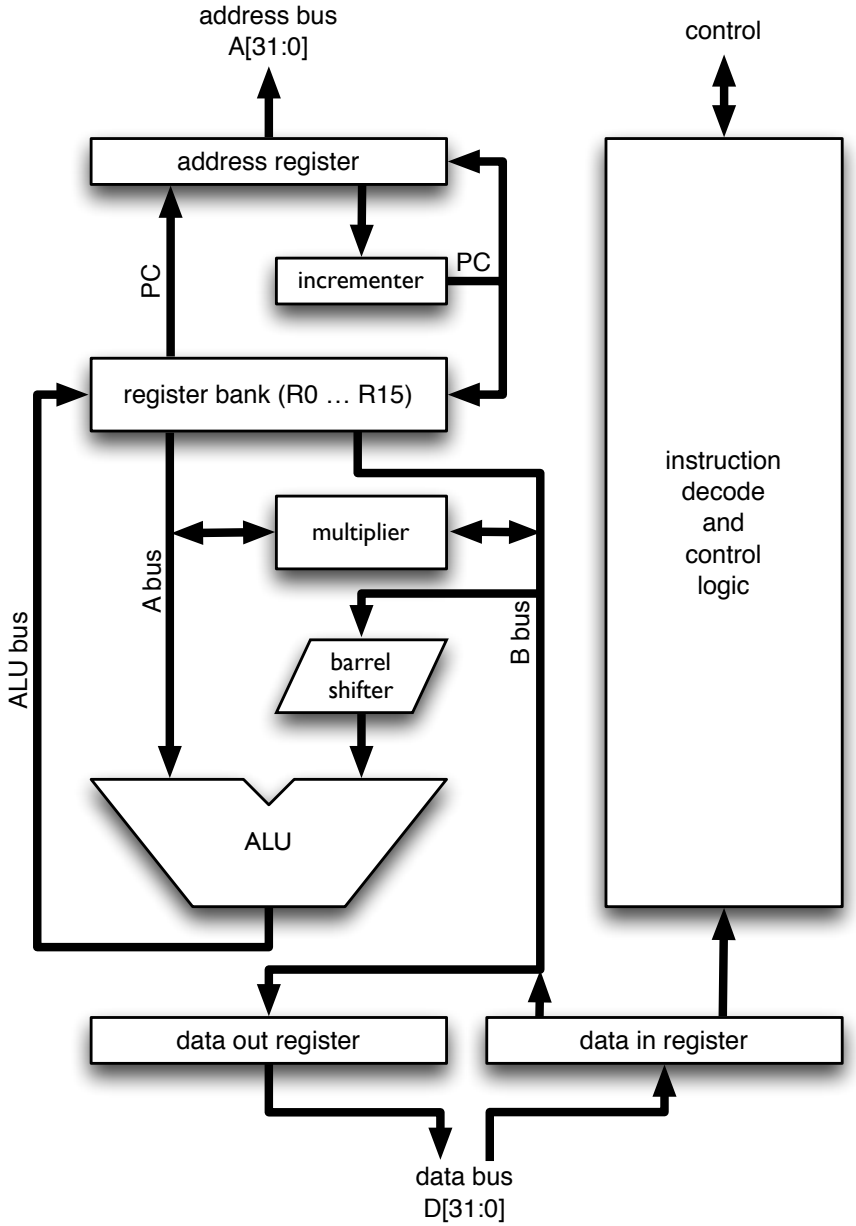
```
LDR    r0, =label             ; Initialise base register
LDR    r4, =0                 ; Initialise index register = 0
LDR    r1, [r0, r4, LSL #2]   ; r1 = Memory.Word[r0 + r4 * 4]
ADD    r4, r4, #1             ; r4 = r4 + 1
LDR    r2, [r0, r4, LSL #2]   ; r2 = Memory.Word[r0 + r4 * 4]
ADD    r4, r4, #1             ; r4 = r4 + 1
LDR    r3, [r0, r4, LSL #2]   ; r3 = Memory.Word[r0 + r4 * 4]
```

Example: 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 for *offset*!!

```
   LDR   R1, =TESTDATA          ; address = address of first word-wize value
   LDR   R0, =0                 ; sum = 0
   LDR   R4, =0                 ; count = 0

L1
   CMP   R4, #10                ; while (count < 10)
   BHS   L2                     ; {
   LDR   R6, [R1, R4, LSL #2]   ;  num = Memory.Word[address + count * 4]
   ADD   R0, R0, R6             ;  sum = sum + num
   ADD   R4, R4, #1             ;  count = count + 1
   B     L1                     ; }
L2                              ;
```



Steve Furber, ***"ARM System-on-Chip Architecture",*** 2nd edition, Addison Wesley Professional, 2000.

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]      ; load
ADD   R1, R1, #4    ; increment 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

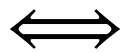
Syntax: post-indexed addressing modes place the immediate/register/scale addressing mode operands **after** the [ ]

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 (LSR/LSL)

Pre-Indexed Addressing Modes


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



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

Immediate Offset Addressing

Immediate Pre-Indexed Addressing

Syntax: pre-indexed addressing modes place the immediate/register/scale addressing mode operands **inside** the [ ]

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 Post-Indexed, Scaled Register Post-Indexed (LSR/LSL)

Example: Upper Case String (using Immediate Post-Indexed Addressing)


```
   LDR   r1, =TESTSTR        ; address = teststr

L1
   LDRB  r0, [r1], #1        ; while ( (char = Memory.Byte[address++])
   CMP   r0, #0              ;       != 0 )
   BEQ   L3                  ; {
   CMP   r0, #'a'            ;  if (char >= 'a'
   BCC   L2                  ;   AND
   CMP   r0, #'z'            ;   char <= 'z')
   BHI   L2                  ;  {
   BIC   r0, #0x00000020     ;   char = char AND NOT 0x00000020
   STRB  r0, [r1, #-1]       ;   Memory.Byte[address - 1] = char
L2                           ;  }
   B     L1                  ;
L3                           ; }
```

Immediate Post-Indexed Addressing

Note the use of Immediate Offset Addressing to (temporarily) compensate for the earlier post-increment

Example: Sum (using Immediate Post-Indexed Addressing)


```
   LDR   R1, =TESTDATA ; address = address of first word-wize value
   LDR   R0, =0        ; sum = 0
   LDR   R4, =0        ; count = 0

L1
   CMP   R4, #10       ; while (count < 10)
   BHS   L2            ; {
   LDR   R6, [R1], #4  ;  num = Memory.Word[address]; address=address+4
   ADD   R0, R0, R6    ;  sum = sum + num
   ADD   R4, R4, #1    ;  count = count + 1
   B     L1            ; }
L2                     ;
```

Review of Single-Dimensional Arrays

Arrays

Nothing new here … you have already been using arrays!

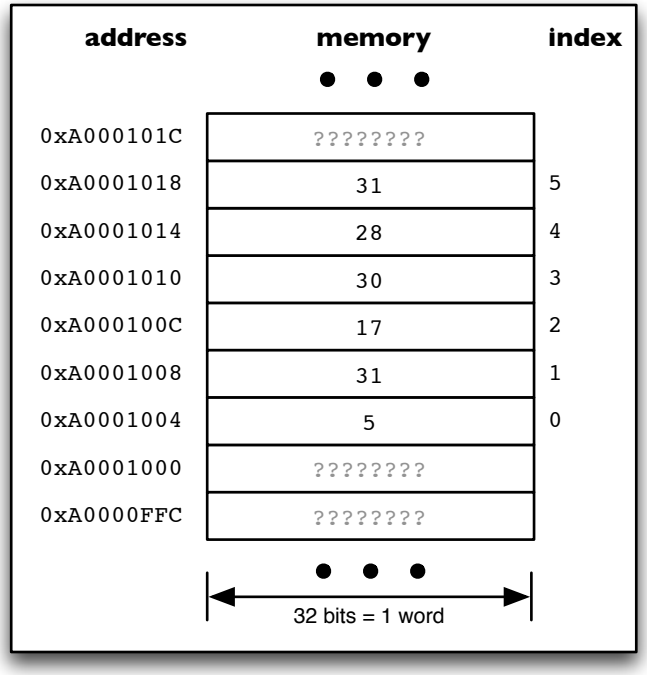
Array – an ordered collection of elements stored sequentially in memory

e.g. integers, ASCII characters, lottery numbers

Homogeneous elements?

(at least with respect to size)

Dimension: number of elements in array



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>

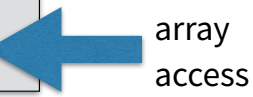
Step 2: add byte offset to array base address to access element

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

Example: retrieve the 4th element (index=3) of an array of words

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

```
LDR   r4, =TESTARR          ; pArr = start address of array
LDR   r5, =3                ; index = 3 (4th element)
LDR   r6, [r4, r5, LSL #2]  ; elem = Memory.Word[pAarr + (index * 4)]
```



Array Access Example - Sum

Déjà Vu!

```
   LDR   R1, =TESTARR          ; start address of myArray
   LDR   R0, =0                ; sum = 0
   LDR   R4, =0                ; count = 0

L1
   CMP   R4, #10               ; while (count < 10)
   BHS   L2                    ; {
   LDR   R6, [R1, R4, LSL #2]  ;  num = myArray[count]
   ADD   R0, R0, R6            ;  sum = sum + num
   ADD   R4, R4, #1            ;  count = count + 1
   B     L1                    ; }
L2                             ;
```

The pseudo-code comments have changed but the program is identical!!!

(See **Addressing Modes**)

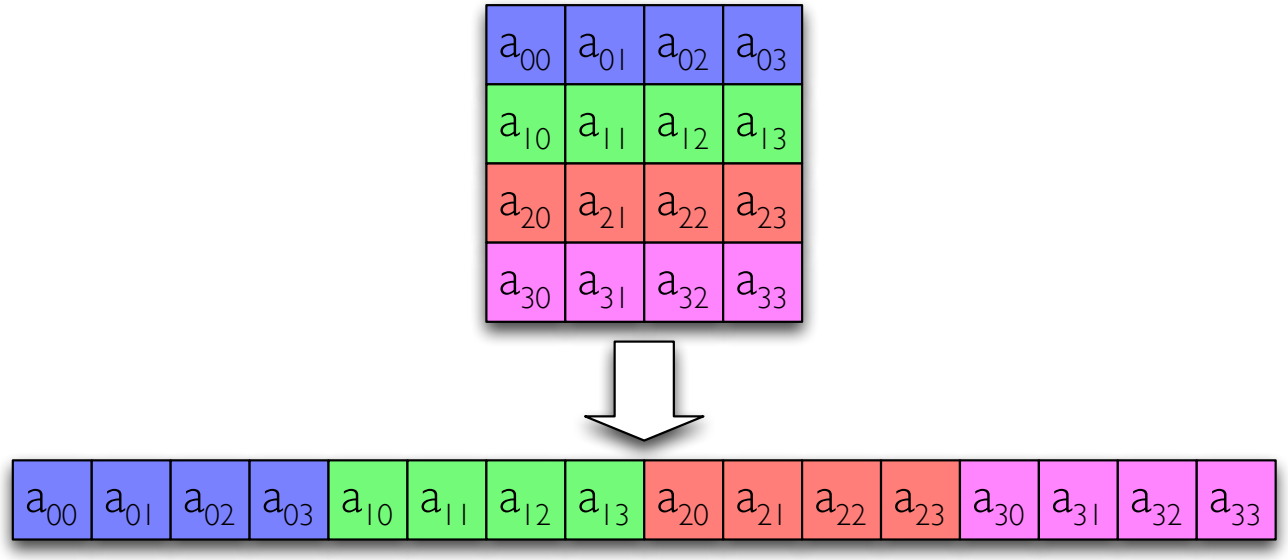
Multi-Dimensional Arrays

2D Arrays

Arrays can have more than one dimension

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

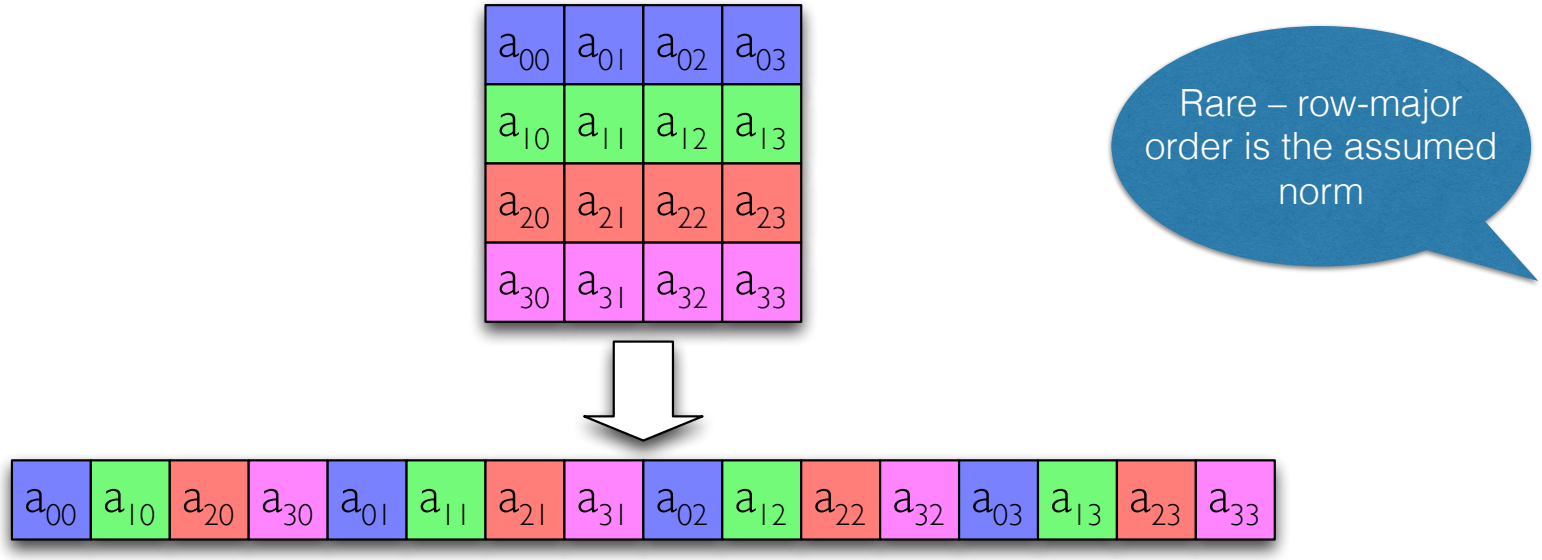
Stored in memory by mapping the 2D array into 1D memory, e.g.



2D Arrays

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

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



2D Array Example

2D array declared in memory

```
        AREA  TestData, DATA, READWRITE

array   DCD   6, 3, 8, 2, 5, 2, 9, 1 ; row 0
        DCD   3, 7, 2, 8, 5, 7, 2, 7 ; row 1
        DCD   2, 4, 7, 4, 2, 6, 7, 4 ; row 2
        DCD   1, 9, 3, 2, 9, 5, 6, 8 ; row 3
        DCD   7, 5, 3, 7, 5, 8, 2, 1 ; row 4
        DCD   6, 4, 8, 9, 0, 3, 2, 5 ; row 5
```

… or equivalently …

Why are these equivalent?

```
        AREA  TestData, DATA, READWRITE

array   DCD   6, 3, 8, 2, 5, 2, 9, 1, 3, 7, 2, 8, 5, 7, 2, 7, 2, 4, 7, 4, 2
        DCD   6, 7, 4, 1, 9, 3, 2, 9, 5, 6, 8, 7, 5, 3, 7, 5, 8, 2, 1, 6, 4
        DCD   8, 9, 0, 3, 2, 5
```

Accessing 2D Array Elements

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

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

<index> = (<row> × <row size>) + <col>

Step 2: translate 1D array index 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>

Step 1 is new

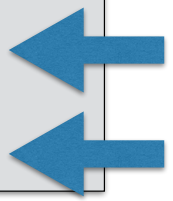
Steps 2 & 3 are the same as those for a 1-D array …

… 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 3rd row and 2nd column of a 2D array of words with 6 rows and 8 columns – array[3][2]

```
LDR   r4, =TESTARR          ; pArr = address of array start
LDR   r5, =COLSZ            ; load col_size
LDR   r6, =ROWSZ            ; load row_size

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

LDR   r1, =3                ; row = 3
LDR   r2, =2                ; col = 4

; <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 = Memory.Word[ pArr + (index*4) ]
```



Multi-dimension arrays

e.g. a 3D array of size sz×sy×sx

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

$$\textbf{index} = ((z \times sy \times sx) + (y \times sx) + x)$$

e.g. a 4D array of size sz×sy×sx×sw

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

$$\textbf{index} = ((z \times sy \times sx \times sw) + (y \times sx \times sw) + (x \times sw) + w)$$

Warning: an array of bytes with odd dimensions may cause the data following the array to be odd aligned, requiring padding of one byte (similarly for half-words)