

CS1021 AFTER READING WEEK

Mid-Semester Test

- NOW Thurs 8th Nov @ 9am in Goldsmith Hall (ALL students to attend at 9am)

Final 2 Labs

- Lab5 2-Nov-18, due 16-Nov-18 (2 weeks duration)
- Lab6 16-Nov-19, due 30-Nov-18 (2 weeks duration)

End of Semester Exam

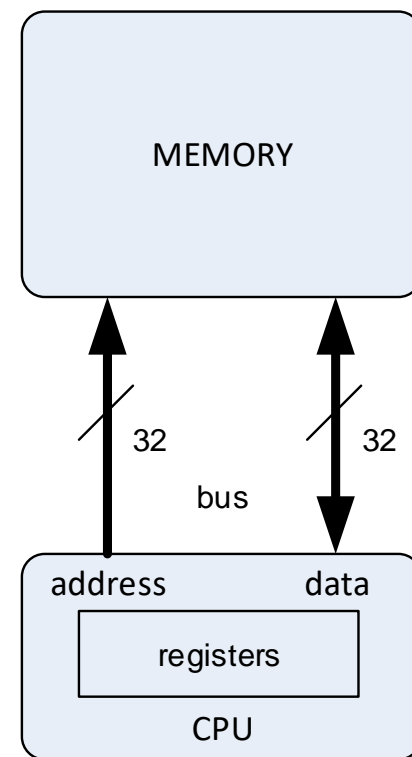
- written 2 hour exam (week of 12-Dec-18)
- answer 3 out of 4 questions (not 2 out of 3 as in recent CS1021 exams)
- 40 mins per question
- questions similar to previous CS1021 exams (shorter, less parts)

Yet to cover

- reading and writing to memory, stacks and subroutines

ARM Memory System

- ARM system comprises CPU and memory
- instructions and data stored in memory
- CPU can read (**LOAD**) data from memory into a register
- CPU can write (**STORE**) data from a register into memory
- called a load / store architecture
- to operate on data in memory, the data must first be **loaded** into register(s), updated in the registers and then **stored** back to memory



Memory Revision

- memory comprises an array of memory locations
- each location stores a byte of data
- each location has a unique 32 bit address
0x00000000 to 0xFFFFFFFF
- the address space, 2^{32} bytes (4GB), is the amount of memory that can be physically attached to the CPU

0xFFFFFFFF	0xFF
0xFFFFFFFFE	0xEE
0xFFFFFFFFD	0xDD
0xFFFFFFFFC	0xCC
⋮	
0x00000005	0x05
0x00000004	0x04
0x00000003	0x11
0x00000002	0x22
0x00000001	0x33
0x00000000	0x44

memory as an
array of BYTES

Memory Revision...

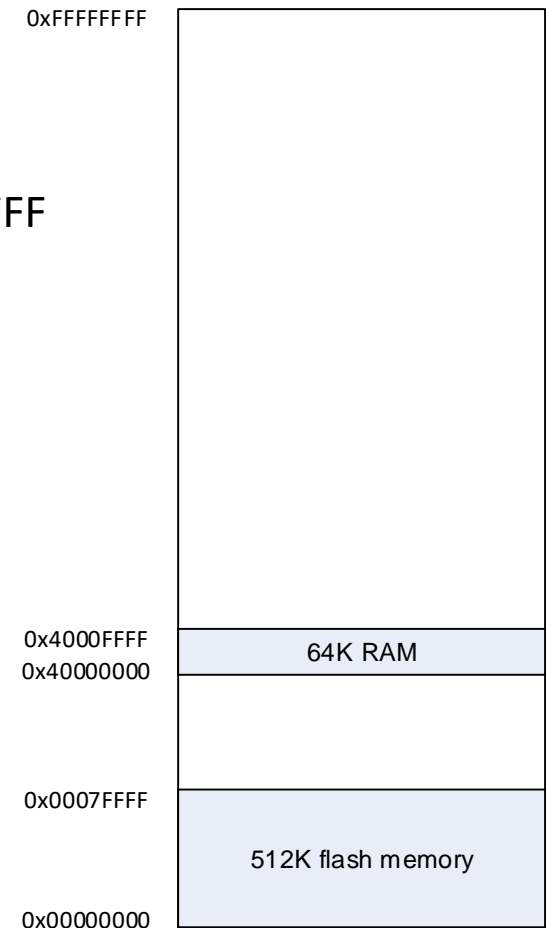
- often easier to view memory as an array of WORDs (32 bits) rather than an array of BYTEs
- as each WORD location is aligned on a 4 byte boundary, the low order 2 bits of each address is 0
- making a comparison with the previous slide, the byte of data stored at memory location 0 is the least significant byte of the WORD stored in location 0
- this way of storing a WORD is termed LITTLE ENDIAN - the least significant byte is stored at the lowest address (the other way is BIG ENDIAN)
- ARM CPUs can be configured to be LITTLE ENDIAN or BIG ENDIAN (term from [Gulliver's Travels](#))

0xFFFFFFFFC	0xFFEEDDCC
0xFFFFFFFF8	0xF8F8F8F8
⋮	
0x0000000C	0x87654321
0x00000008	0x8ABCDEF0
0x00000004	0x07060504
0x00000000	0x11223344

memory as an array of
WORDS

NXP LPC2468 Memory Map

- address space NOT fully populated with memory devices
- 512K of flash memory at address 0x00000000 to 0x0007FFFF
- 64K RAM at address 0x40000000 to 0x4000FFFF
- flash memory
 - read ONLY (programmed electronically - “flashed”)
 - retains data when power removed
- RAM (random access memory)
 - read write
 - loses its data when power removed
- uVision projects are configured to simulate this memory map
- code placed in flash memory starting at address 0x00000000



NXP LPC2468 memory map
(NOT to scale)

Load Instructions - LDR and LDRB

- memory address specified in a register

- load word

```
LDR    R1, =0x40000000 ; R1 -> 0x40000000 (in RAM)
LDR     R0, [R1]        ; R0 = MWORD[0x40000000]
```

R1 points to 0x40000000 (in RAM)

load data from 0x40000000 (in RAM)

- load byte

```
LDR     R1, =0x40000003 ; R1 -> 0x40000003 (in RAM)
LDRB    R0, [R1]        ; R0 = MBYTE[0x40000003]
```

R1 points to 0x40000003 (in RAM)

load data from 0x40000003 (in RAM)

LDR and LDRB

- load word
 - reads 4 bytes from memory address into a register
 - address must be even (LS address bit = 0)
 - normally used with an address aligned on a 4 byte boundary (address ends with $\dots00_2$) BUT ...
 - if address end with $\dots10_2$, it accesses memory as though the address ended with $\dots00_2$ but swaps the high and low 16 bits
- load byte
 - reads byte from memory address and stores in LS byte of register
 - clears MS bytes of register

LDR and LDRB...

- load word

```
LDR R1, =0x40000000
LDR R0, [R1]
```

R0

0x04030201

- load byte

```
LDR R1, =0x40000003
LDRB R0, [R1]
```

R0

0x00000004

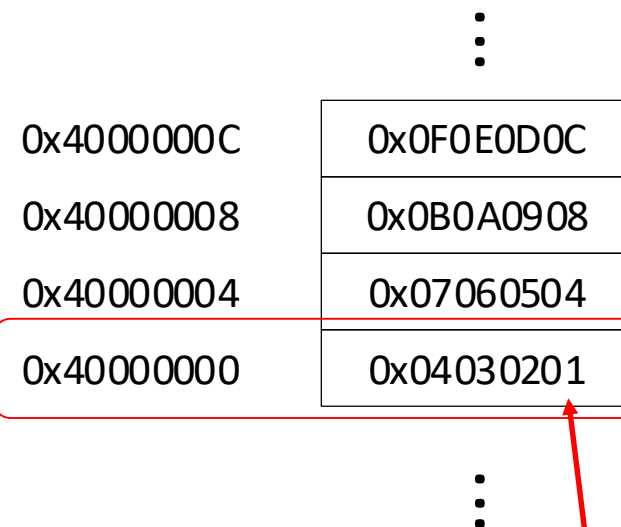
- load word (address ends with $..10_2$)

```
LDR R1, =0x40000002
LDR R0, [R1]
```

R0

0x02010403

high and low 16 bits swapped



memory

little endian
0x01 in address 0x40000000
0x04 in address 0x40000003

Store Instructions – STR and STRB

- memory address specified in a register

- store word

R1 points to 0x40000000 (in RAM)

```
LDR    R1, =0x40000000 ; R1 -> 0x40000000 (in RAM)
STR     R0, [R1]        ; MWORD[0x40000000] = R0
```

- store byte

R1 points to 0x40000002 (in RAM)

```
LDR     R1, =0x40000001 ; R1 -> 0x40000002 (in RAM)
STRB    R0, [R1]        ; MBYTE[0x40000002] = R0 (LS byte)
```


STR and STRB

- store word
 - writes ALL 4 bytes of register to memory address
 - address must be aligned on a 4 byte boundary (address ends with $\dots 00_2$)
- store byte
 - writes LS byte of register to memory address

Example

- a, b and c are 32-bit signed binary integers stored in memory locations 0x40000000, 0x40000004 and 0x40000008 respectively
- compute $c = a + b$

R1 points to 0x40000000 (where a is stored in RAM)



```
LDR    R1, =0x40000000    ; R1 -> a
LDR    R0, [R1]           ; R0 = a
LDR    R1, =0x40000004    ; R1 -> b
LDR    R1, [R1]           ; R1 = b
ADD    R0, R0, R1         ; R0 = a + b
LDR    R1, =0x40000008    ; R1 -> c
STR    R0, [R1]          ; c = a + b
```

ASCII strings

- American Standard Code for Information Interchange
- ASCII is a standard used to encode alphanumeric and other characters
- each character is stored as a single byte (8 bits)
- upper and lower case characters have different ASCII codes
- ASCII only uses 7 bits to encode the character, giving 128 possible characters
- MSB may be used as a parity bit
 - ODD or EVEN parity
 - parity bit set so that number of 1 bits in a character is either odd or even
 - used to detect transmit and receive errors
- originally used to transmit characters from a computer to a tele printer (terminal)

MEMORY

ASCII Table (hex values)

ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_	SP 0020	! 0021	" 0022	# 0023	\$ 0024	% 0025	& 0026	' 0027	(0028) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C] 005D	^ 005E	_ 005F
6_	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	 007C	} 007D	~ 007E	DEL 007F

'H' = 0x48

'e' = 0x65

'l' = 0x6C

'o' = 0x6F

Letter Number Punctuation Symbol Other undefined Changed from '63 version

ASCII ...

- the string “Hello”, if at address 0x1000, stored as follows

H	e	l	l	o	NUL	ASCII code address
0x48	0x65	0x6c	0x6c	0x6f	0x00	
0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	

- ASCII strings early always NUL terminated
- only 96 ASCII characters are printable, the remainder are control codes
- example control codes

0x0A	LF	line feed
0x0D	CR	carriage return
0x08	BS	backspace
0x09	HT	horizontal tab
0x1B	ESC	escape
0x00	NUL	NUL

Example

- copy a NULL terminated ASCII string from 0x1000 (in read-only flash memory) to 0x40000000 (RAM)

```
LDR    R1, =0x1000      ; R1 -> src
LDR    R2, =0x40000000  ; R2 -> dst
L LDRB  R0, [R1]         ; get char from src string
STRB   R0, [R2]         ; store char in dst string
ADD    R1, R1, #1       ; move to next src char
ADD    R2, R2, #1       ; move to next dst char
CMP    R0, #0           ; char == 0 ?
BNE    L               ; next character if not finished
```

R1 points to src string in RAM

R2 points to dst string in RAM

CS1021 Mid-Semester Test

- Thurs 8th Nov @ 9am in Goldsmith Hall
- ALL students to attend at 9am (no Tutorial @ 13.00)
- NO calculators, phones, laptop etc.
- 20 Questions (like Tutorial questions)
- ALL questions carry equal marks (some are easier than others)
- Remember to fill in exam number, student ID and name on answer booklet

uVision Console and Memory Windows

The screenshot displays the uVision IDE with the following components and annotations:

- Registers Window:** Shows the current state of registers. The CPSR register is highlighted with a value of 0x00000003.
- Disassembly Window:** Shows assembly code for the current address. A red arrow points from the "console window" callout to this window.
- Source Window (lab5.s):** Shows the C source code. A red arrow points from the "memory window" callout to this window.
- UART #1 Window:** Shows the serial output. It displays "type string" and "123456". A red arrow points from the "memory address" callout to the address field in the Memory 1 window.
- Memory 1 Window:** Shows the memory dump starting at address 0x40000000. The first row of data is 0x00 00 00 00 00 00 00 00, which corresponds to the ASCII bytes for "0123456" + 0x0D (CR). A red arrow points from the "memory data" callout to this row.

Annotations:

- console window (UART #1)**
[View][Serial Windows][UART #1]
- memory window**
[View][Memory Windows][Memory 1]
- memory address**
- memory data**
ASCII bytes for "0123456" + 0x0D (CR)

DCB, DCW and DCW Assembler Directives

- can use assembler to initialise the contents of **flash** memory with constant data
- if RAM needs to be initialised, it is normally initialised at start-up by copying data from flash memory (very microcontroller centric)

C0 DCB “Hello World!”, 0 ; NUL terminated ASCII string

C1 DCD 1, 2, 3, 4, 5, 6, 7, 8 ; 8 x 32-bit (word) constants (why DCD for words?)

C2 DCW 1, 2, 3, 4, 5, 6, 7, 8 ; 8 x 16-bit (halfword) constants

- load address of constant string using

points to

LDR R0, =C0 ; R0 -> “Hello World!”

- need to place constants where they will not be mistakenly executed as code

DCB, DCW and DCW Assembler Directives ...

The screenshot shows the uVision IDE with the following components:

- Registers Panel:** Shows the current state of registers R0 through R15, CPSR, and SPSR. R0 is highlighted.
- Disassembly Panel:** Shows assembly code for the test.axf file. A red arrow points from the annotation "constant data stored in memory @ address 0x50" to the instruction `36: STOP B STOP ; infinite loop`.
- test.s Panel:** Shows the source code for test.s. It includes:
 - Line 28: `LDR R1, =0x40000000 ; address`
 - Line 29: `LDR R0, [R1, #4]`
 - Line 30: `LDR R0, [R1], #4`
 - Line 31: `31: LDR R0, [R1, -R4]`
 - Line 32: `32: LDR R0, [R1, -R4, LSL #2]`
 - Line 33: `33: STOP B STOP ; infinite loop`
 - Line 38: `38: DCB "Hello World!", 0 ; NUL terminated ASCII string`
 - Line 39: `39: DCD 1, 2, 3, 4, 5, 6, 7, 8 ; 8 x 32-bit (word) constants (why DCD?)`
 - Line 40: `40: DCW 1, 2, 3, 4, 5, 6, 7, 8 ; 8`
 - Line 41: `41: END`
- Memory Panel:** Shows a memory dump starting at address 0x40. Red arrows point from the annotations "C0 @ address = 0x50 (padded to be a multiple of 4 bytes)", "C1 @ address = 0x60", and "C2 @ address = 0x80" to the corresponding memory locations in the dump.

Annotations:

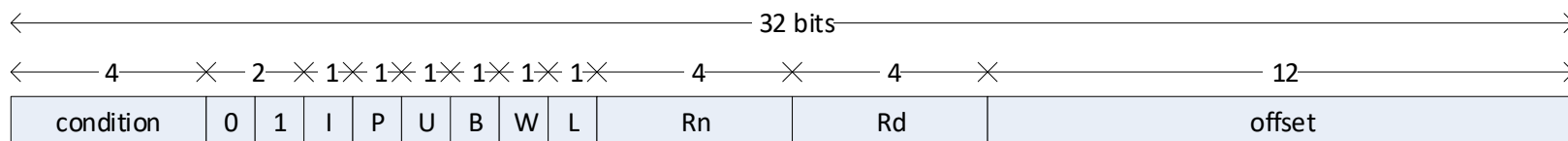
- constant data stored in memory @ address 0x50** (points to the `STOP B STOP` instruction in the disassembly panel).
- C0 @ address = 0x50 (padded to be a multiple of 4 bytes)** (points to the memory location 0x50 in the memory dump).
- C1 @ address = 0x60** (points to the memory location 0x60 in the memory dump).
- C2 @ address = 0x80** (points to the memory location 0x80 in the memory dump).

Additional features of LDR / STR instructions

- additional features can be used to reduce the number of instructions that have to be written and the number of instructions executed at runtime
- `LDR R0, [R1]` ; R1 used as an address register as R1 contains an address
- `STR R0, [R1]` ; R1 used as an address register as R1 contains an address

Some Advanced features of LDR / STR instructions

- best understood by examining LDR / STR machine code fields



LDR STR machine code fields

- I - immediate bit
 - if 0, offset field specifies a 12 bit offset (0 .. 4095)
 - if 1, offset field specifies a register + shift operation
- P - pre or post indexing
 - if 0 (post indexing), add offset to base register after transfer
 - if 1 (pre indexing), add offset to base register before transfer
- U (up/down) - 0 for subtract and 1 to add offset to base register
- B - 0 for word and 1 for byte transfer
- W - 1 for write back of effective address into base register
- L - 0 for store and 1 for load

Examples

immediate offset

- LDR R0, [R1, #4] ; with immediate offset

$R0 = \text{MEM}[R1 + 4]$

I	P	U	B	W	L	offset
0	1	1	0	0	1	4

- LDR R0, [R1], #4 ; post-indexed

$R0 = \text{MEM}[R1]$

$R1 = R1 + 4$; post increment address register

I	P	U	B	W	L	offset
0	0	1	0	1	1	4

can specify a +ve or -ve offset

- LDR R0, [R1, #-4]! ; pre-indexed

$R1 = R1 - 4$; pre increment address register

$R0 = \text{MEM}[R1]$

I	P	U	B	W	L	Offset
0	1	0	0	1	1	4

- LDR can read memory and increment address register in a single instruction

Examples...

register offset

- LDR R0, [R1, R4] ; register offset

$R0 = \text{MEM}[R1 + R4]$

I	P	U	B	W	L	offset
1	1	1	0	0	1	R4

- LDR R0, [R1], R4 ; register offset post-indexed

$R0 = \text{MEM}[R1]$

$R1 = R1 + R4$; post increment address register

I	P	U	B	W	L	offset
1	0	1	0	1	1	R4

can specify a +ve or -ve offset

- LDR R0, [R1, -R4]! ; pre-indexed register offset

$R1 = R1 - R4$

$R0 = \text{MEM}[R1]$

; pre increment address register

I	P	U	B	W	L	offset
1	1	1	0	1	1	R4

- LDR can read memory and increment address register in a single instruction

Examples...

scaled register offset

- LDR R0, [R1, R4, LSL #2] ; scaled register offset

$R0 = \text{MEM}[R1 + R4 * 4]$

I	P	U	B	W	L	offset
1	1	1	0	0	1	R4, LSL #2

- LDR R0, [R1], R4, LSL #2 ; post-indexed scaled register offset

$R0 = \text{MEM}[R1]$

$R1 = R1 + R4 * 4$

can specify a +ve or -ve offset

I	P	U	B	W	L	offset
1	0	1	0	1	1	R4, LSL #2

- LDR R0, [R1, -R4, LSL #2]! ; pre-indexed scaled register offset

$R1 = R1 - R4 * 4$

$R0 = \text{MEM}[R1]$

I	P	U	B	W	L	offset
1	1	1	0	1	1	-R4, LSL #2

- LDR can read memory and increment address register in a single instruction

Example 1: string copy

- copy a NULL terminated ASCII string from 0x1000 (in read-only flash memory) to 0x40000000 (RAM)

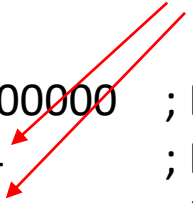
```
LDR    R1, =0x1000      ; R1 -> src string
LDR    R2, =0x40000000  ; R2 -> dst string
L LDRB  R0, [R1], #1     ; load ch from src string AND post increment R1
STRB   R0, [R2], #1     ; store ch in dst string AND post increment R2
CMP    R0, #0           ; ch == 0? has NUL ch been copied?
BNE    L                ; next ch if NOT finished
```

Example 2: $c = a + b$

- a , b and c are 32-bit signed binary integers stored in memory locations 0x40000000, 0x40000004 and 0x40000008 respectively

- compute $c = a + b$

post increment R1



```
LDR    R1, =0x40000000    ; R1 -> a
LDR    R0, [R1], #4        ; R0 = a; R1 = R1 + 4 -> b
LDR    R2, [R1], #4        ; R2 = b; R1 = R1 + 4 -> c
ADD    R0, R0, R2          ; R0 = a + b
STR    R0, [R1]            ; c = a + b
```

- exploiting (1) a , b and c are stored in sequential memory locations AND (2) can post increment $R1$ as part of LDR instruction

Example 3: reverse string

- if a zero terminated string of ASCII characters is stored at memory address 0x40000000, write ARM assembly language instructions to reverse the string in situ
- step 1: find R2 such that R2->last ch in string (excluding terminating 0)

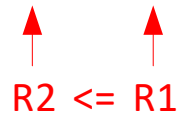
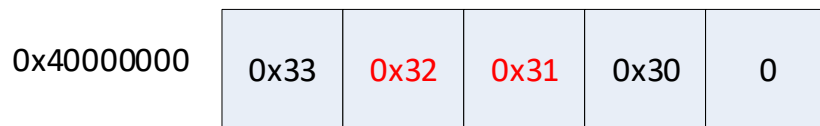
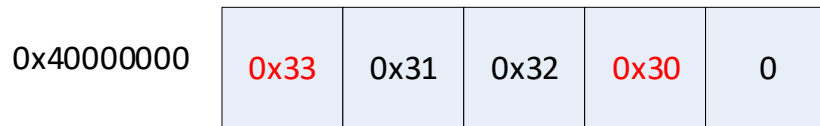
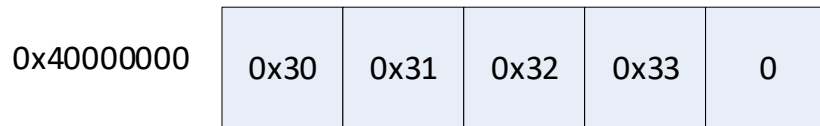
```
;
; R1 -> first ch in string
;
LDR    R1, =0x40000000    ; R1 -> string
MOV    R2, R1             ; R2 -> string
LO     LDRB    R0, [R2], #1    ; load next ch of string AND R2 = R2 + 1
      CMP    R0, #0           ; 0?
      BNE    LO               ; next ch
      SUB    R2, R2, #2       ; R2 -> last ch of string (excluding terminating 0)
```

post increment R1

decrement R2 by 2 as R2 was one past NUL terminator

Example 3: reverse string ...

- step 2: swap first and last characters and work towards middle



- loop terminating condition

Example 3: reverse string ...

```
;
; R1 -> first ch in string
; R2 -> last ch in string (excluding termination zero)
;
; swap first and last characters and work towards middle
;
L1    CMP    R2, R1                ; if R2 <= R1? ...
      BLS    L2                    ; finished
      LDRB   R0, [R1]              ; read "first" character
      LDRB   R3, [R2]              ; read "last" character
      STRB   R0, [R2], #-1          ; write "first" character to "last" slot
      STRB   R3, [R1], #1          ; write "last" character to "first" slot
      B      L1
L2
```

post increment R2 by - 1

post increment R1 by 1

Example 4: Array Access

- consider an array **a** of 32-bit signed integers stored in memory at address 0x40000000

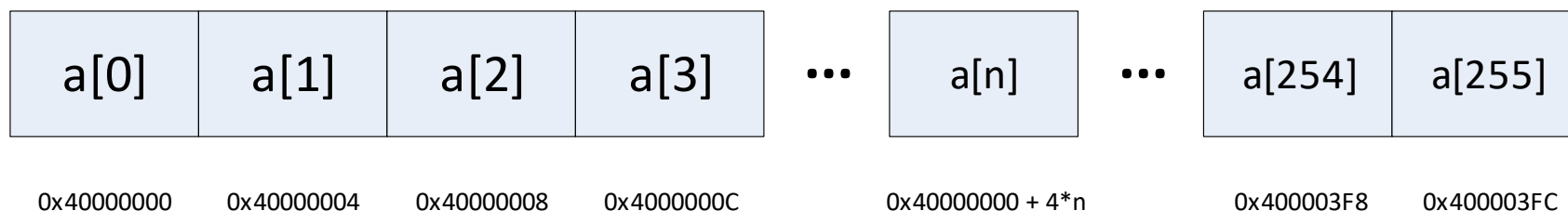
```
int a[256];    // array a contains 256 integers a[0] .. a[255]
```

```
a[0] stored @ MEM[0x40000000]    // increasing by 4...  
a[1] stored @ MEM[0x40000004]    // because each integer  
a[2] stored @ MEM[0x40000008]    // occupies 4 bytes of memory  
...  
a[n] stored @ MEM[0x40000000 + 4*n]  //  
...  
a[255] stored @ MEM[0x400003FC]    //
```

- array **a** occupies $256 \times 4 = 1024 = 0x400$ bytes of memory
- array **a** occupies memory locations 0x40000000 to 0x400003FF

Example 4: Array Access ...

- array elements stored in consecutive memory locations
- as each array element is a 32 bit integer (4 bytes), address increases by 4 from one element to the next



- if `i` and `j` are two 32-bit signed integer variables stored at memory addresses `0x40000400` and `0x40000404` respectively, write ARM assembly language instructions to compute:

`a[i] = a[5] + a[j];`

Example 4: Array Access...

- $a[5]$ - constant index
- $a[i]$ and $a[j]$ - variable indices

```
LDR    R1, =0x40000000    // R1 -> a
LDR    R0, [R1, #5*4]      // R0 = a[5] (R0 = MEM[a + 5*4])
LDR    R2, =0x40000404    // R2 -> j
LDR    R2, [R2]            // R2 = j
LDR    R2, [R1, R2, LSL #2] // R2 = a[j] (R2 = MEM[a + j*4])
ADD    R0, R0, R2          // R0 = a[5] + a[j]
LDR    R2, =0x40000400    // R2 -> i
LDR    R2, [R2]            // R2 = i
STR    R0, [R1, R2, LSL #2] // a[i] = a[5] + a[j] (MEM[a + i*4] = R0)
```


What has not been covered?

- LDRH load halfword
- STRH store halfword

- LDRSB load byte with sign extend
- LDRSH load halfword with sign extend