MEMORY

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

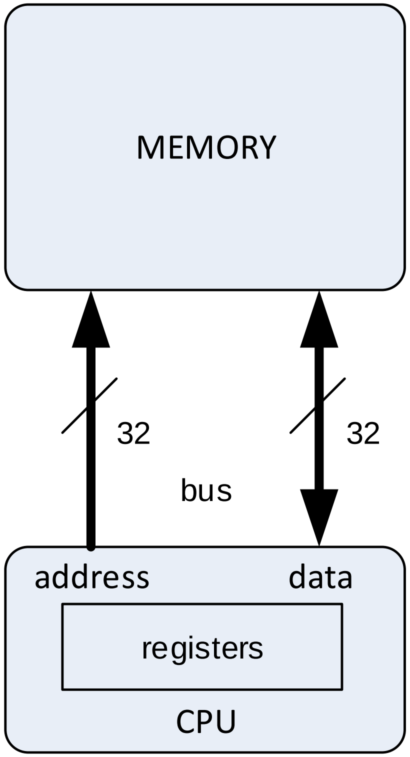
- written 2 hour exam (week of 12-Dec-18)
- answer 3 out of 4 questions (not 2 out of 3 as in recent CS1021 exams)
- 40 mins per question
- questions similar to previous CS1021 exams (shorter, less parts)

Yet to cover

- reading and writing to memory, stacks and subroutines

ARM Memory System

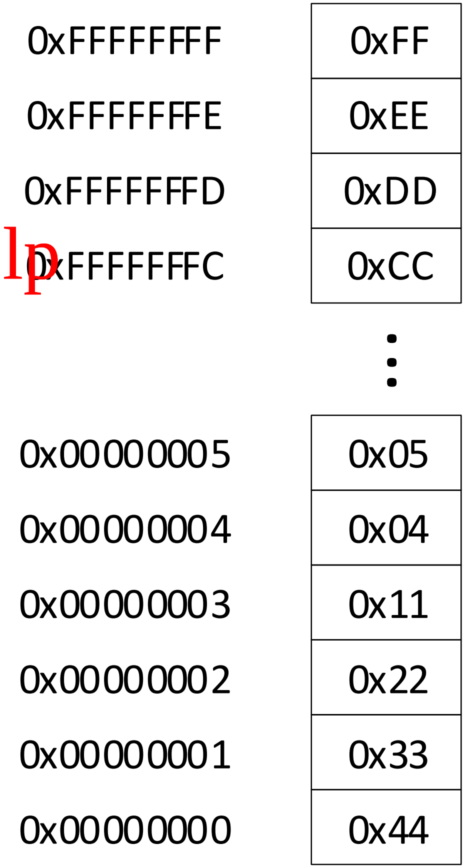
- ARM system comprises CPU and memory
- instructions and data stored in memory
- CPU can read (**LOAD**) data from memory into a register
- CPU can write (**STORE**) data from a register into memory
- called a load / store architecture
- to operate on data in memory, the data must first be **loaded**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



Memory

Memory Revision

- memory comprises an array of memory locations
- each location stores a byte of data
- each location 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
0xFFFFFFFE	0xEE
0xFFFFFFFD	0xDD
0xFFFFFFFC	0xCC
	⋮
0x00000005	0x05
0x00000004	0x04
0x00000003	0x11
0x00000002	0x22
0x00000001	0x33
0x00000000	0x44

memory as an
array of BYTEs

MEMORY

Memory Revision…

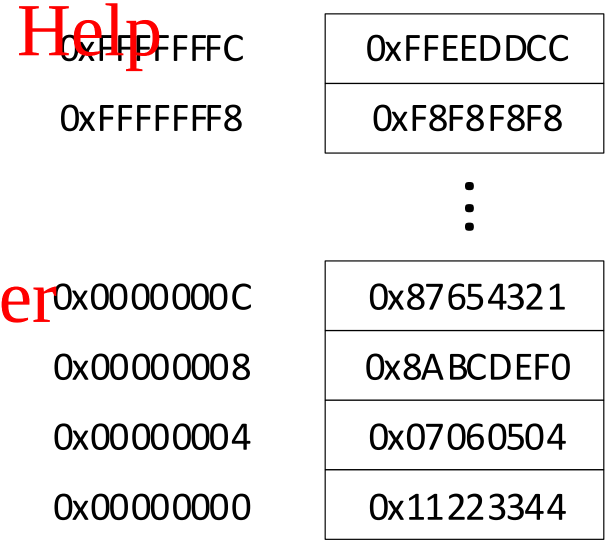
- often easier to view memory as an array of WORDs (32 bits) rather than an array of BYTEs

- 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)



0xFFFFFFFC	0xFFEEDDCC
0xFFFFFFF8	0xF8F8F8F8
	⋮
0x0000000C	0x87654321
0x00000008	0x8ABCDEF0
0x00000004	0x07060504
0x00000000	0x11223344

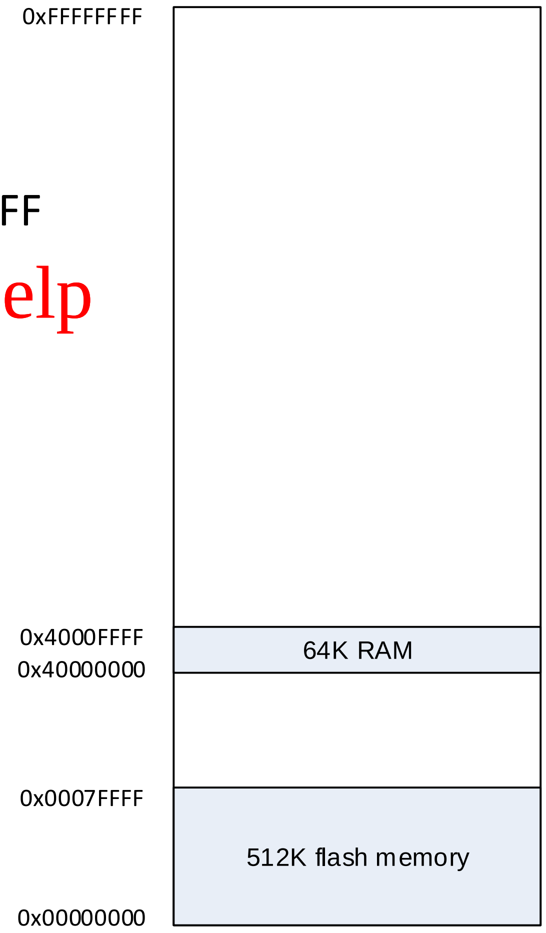
memory as an array of WORDs

Memory

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
  - looses 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)

MEMORY

Load Instructions - LDR and LDRB

- memory address specified in a register

- load word

R1 *points to* 0x40000000 (in RAM)

load data from 0x40000000 (in RAM)

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

- load byte

R1 *points to* 0x40000003 (in RAM)

load data from 0x40000003 (in RAM)

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

MEMORY

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 …00_2) BUT …
  - if address end with …10_2, it accesses memory as though the address ended with …00_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

MEMORY

LDR and LDRB…

- load word

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

R0 0x04030201

- load byte

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

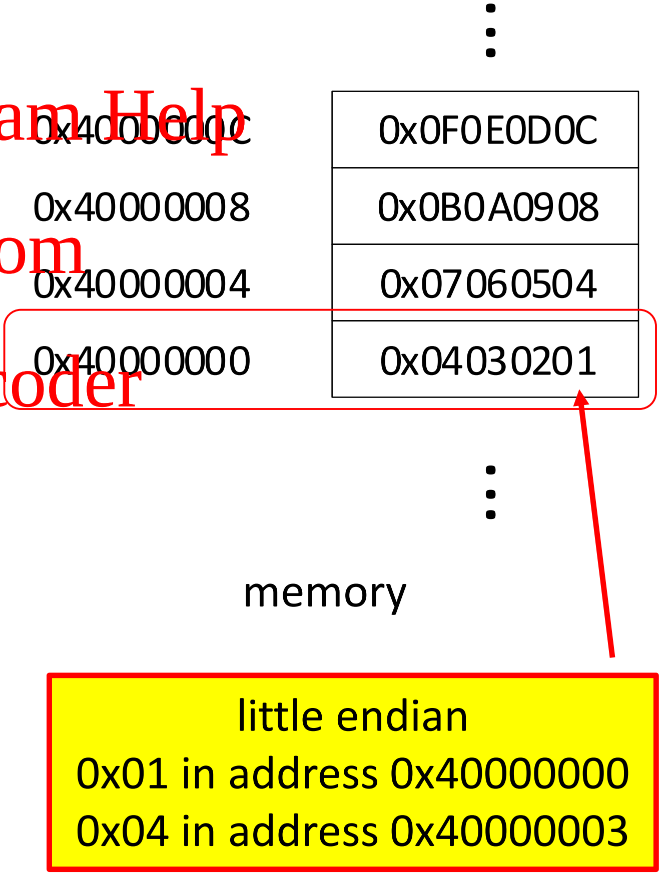
R0 0x00000004

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

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

R0 0x02010403

high and low 16 bits swapped



⋮

0x4000000C	0x0F0E0D0C
0x40000008	0x0B0A0908
0x40000004	0x07060504
0x40000000	0x04030201

⋮

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)
```

MEMORY

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 …00_2)
- store byte
  - writes LS byte of register to memory address

MEMORY

Example

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

- compute c = a + b

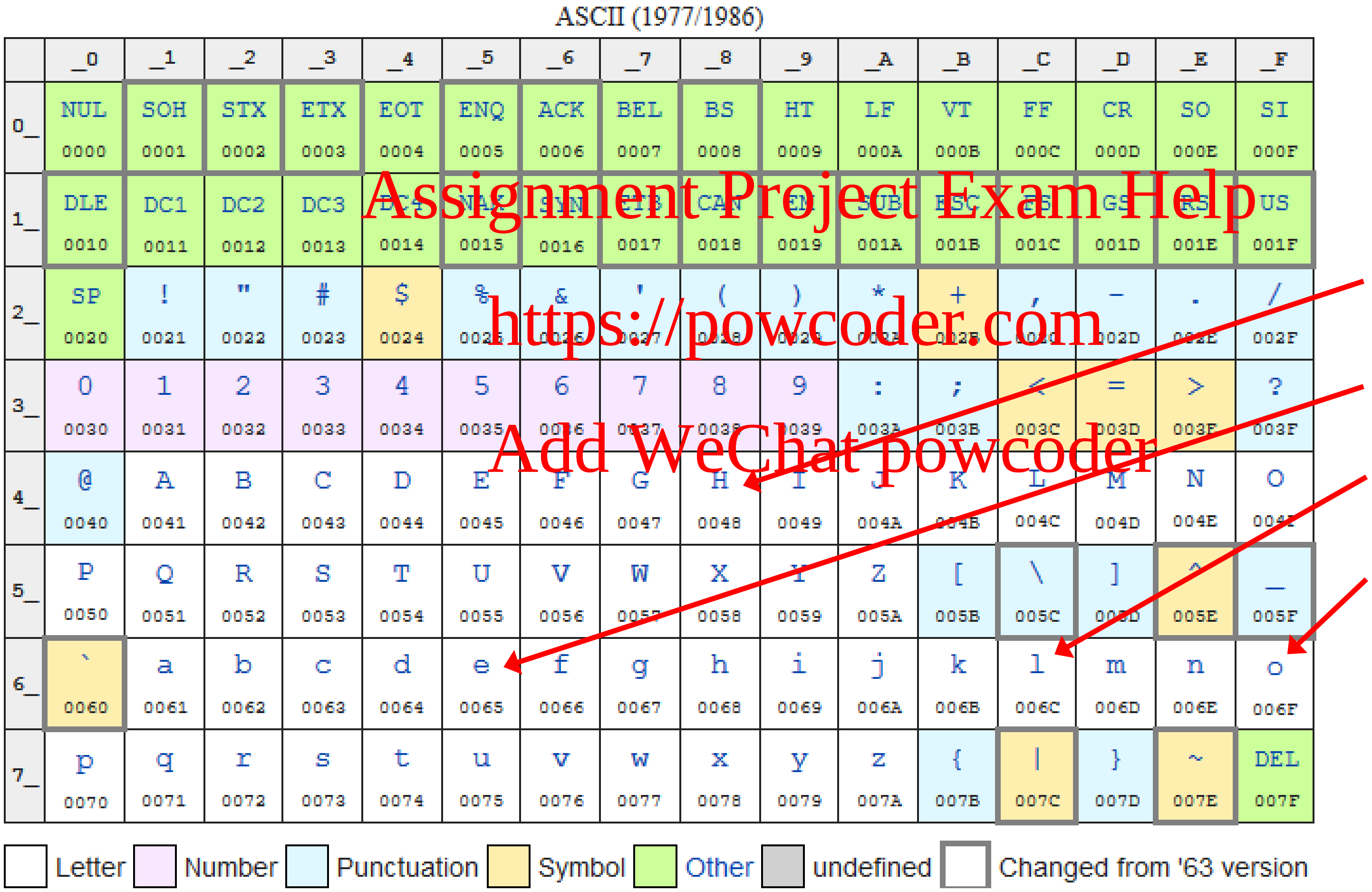
R1 *points to* 0x40000000 (where a is stored in RAM)

```
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)

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

□ Letter □ Number □ Punctuation □ Symbol □ Other □ undefined □ Changed from '63 version

‘H’ = 0x48

‘e’ = 0x65

‘l’ = 0x6C

‘o’ = 0x6F

MEMORY

ASCII …

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

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

- 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

MEMORY

Example

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

R1 *points to* src string in RAM

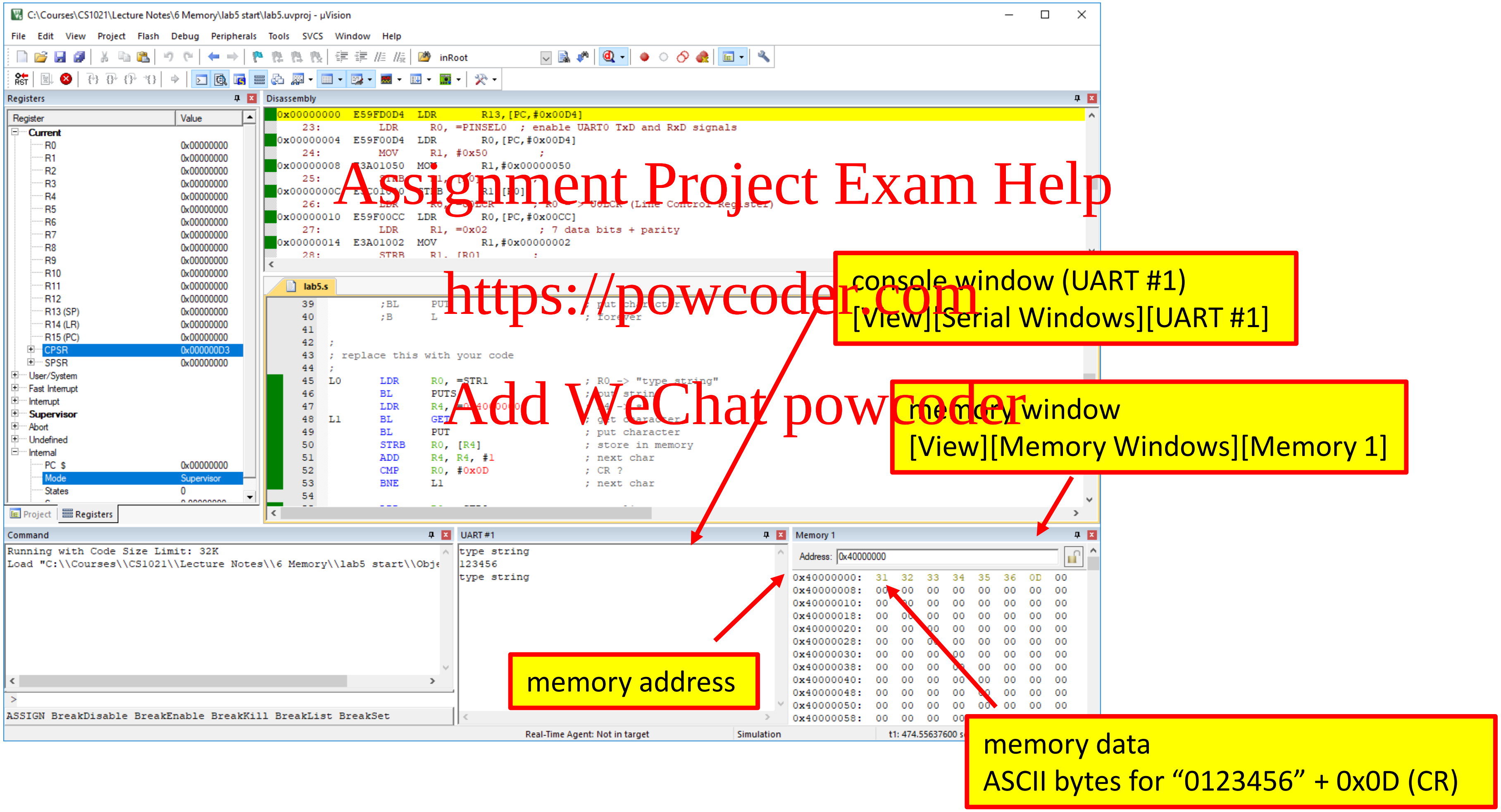
R2 *points to* dst string in RAM

```
        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
```

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



C:\Courses\CS1021\Lecture Notes\6 Memory\lab5 start\lab5.uvproj - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
inRoot
Registers
Disassembly
lab5.s
Project
Registers
Command
UART #1
Memory 1

Register	Value
Current	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000000
CPSR	0x000000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	
PC $	0x00000000
Mode	Supervisor
States	0

```
0x00000000  E59FD0D4  LDR       R13,[PC,#0x00D4]
    23:         LDR     R0, =PINSEL0  ; enable UART0 TxD and RxD signals
0x00000004  E59F00D4  LDR       R0,[PC,#0x00D4]
    24:         MOV     R1, #0x50         ;
0x00000008  E3A01050  MOV       R1,#0x00000050
    25:         STRB    R1, [R0]          ;
0x0000000C  E5C01050  STRB      R1,[R0]
    26:         LDR     R0, =U0LCR        ; R0 -> U0LCR (Line Control Register)
0x00000010  E59F00CC  LDR       R0,[PC,#0x00CC]
    27:         LDR     R1, =0x02         ; 7 data bits + parity
0x00000014  E3A01002  MOV       R1,#0x00000002
    28:         STRB    R1, [R0]          ;
```

```
        ;BL     PUT                 ; put character
        ;B      L                   ; forever

;
; replace this with your code
;
L0      LDR     R0, =STR1           ; R0 -> "type string"
        BL      PUTS                ; put string
        LDR     R4, =0x40000000     ; R4 -> 
L1      BL      GET                 ; get character
        BL      PUT                 ; put character
        STRB    R0, [R4]            ; store in memory
        ADD     R4, R4, #1          ; next char
        CMP     R0, #0x0D           ; CR ?
        BNE     L1                  ; next char
```

```
Running with Code Size Limit: 32K
Load "C:\\Courses\\CS1021\\Lecture Notes\\6 Memory\\lab5 start\\Obje
>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet
```

```
type string
123456
type string
```

Address: 0x40000000

```
0x40000000:  31  32  33  34  35  36  0D  00
0x40000008:  00  00  00  00  00  00  00  00
0x40000010:  00  00  00  00  00  00  00  00
0x40000018:  00  00  00  00  00  00  00  00
0x40000020:  00  00  00  00  00  00  00  00
0x40000028:  00  00  00  00  00  00  00  00
0x40000030:  00  00  00  00  00  00  00  00
0x40000038:  00  00  00  00  00  00  00  00
0x40000040:  00  00  00  00  00  00  00  00
0x40000048:  00  00  00  00  00  00  00  00
0x40000050:  00  00  00  00  00  00  00  00
0x40000058:  00  00  00  00
```

Real-Time Agent: Not in target   Simulation   t1: 474.55637600 s

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)

MEMORY

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

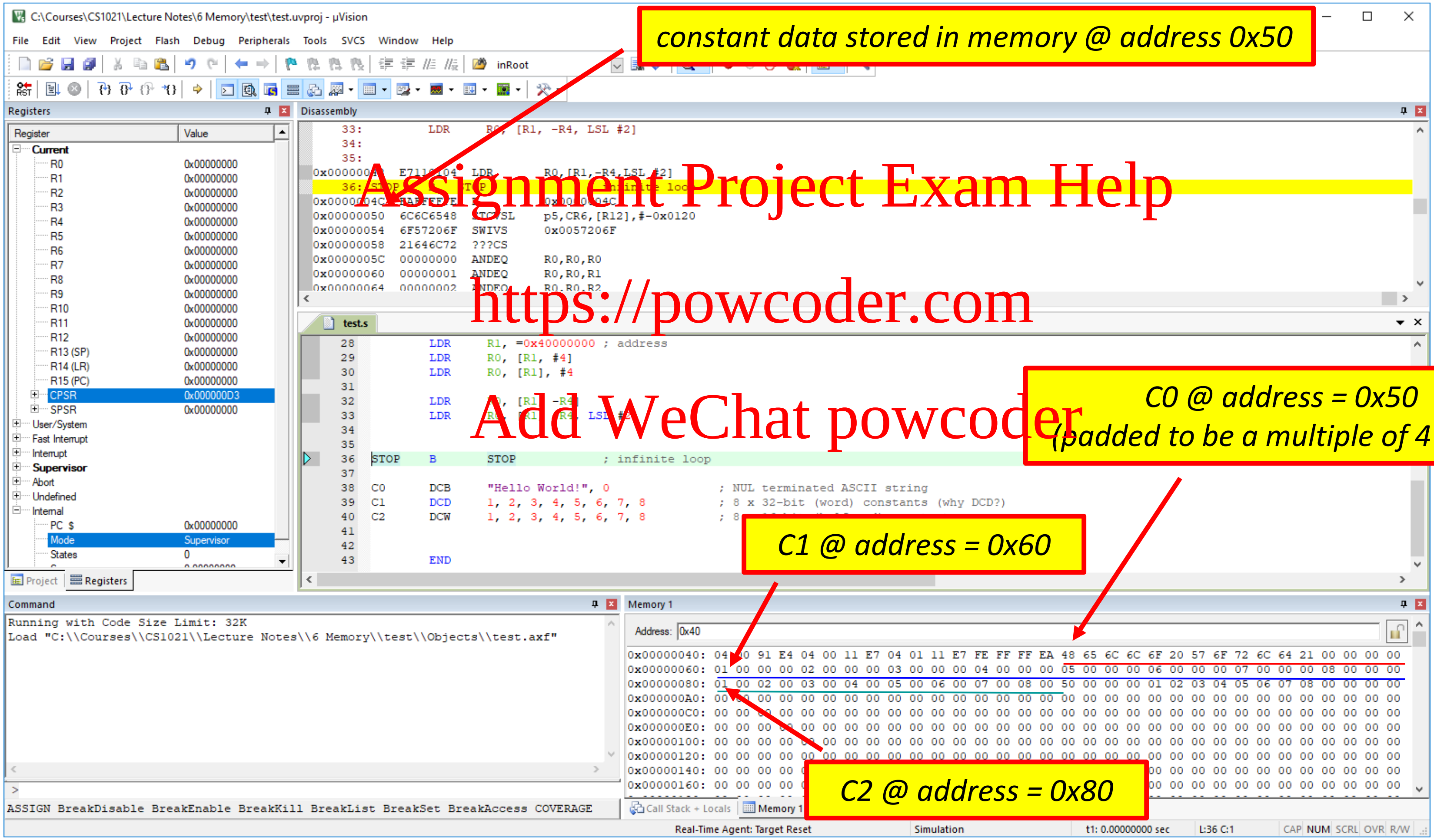
points to

```
        LDR     R0, =C0                 ; R0 -> "Hello World!"
```

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

Memory

DCB, DCW and DCW Assembler Directives …



C:\Courses\CS1021\Lecture Notes\6 Memory\test\test.uvproj - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
inRoot
Registers
Disassembly
test.s
Command
Memory 1
Project
Registers
Call Stack + Locals
Memory 1

constant data stored in memory @ address 0x50

C0 @ address = 0x50 (padded to be a multiple of 4 bytes)

C1 @ address = 0x60

C2 @ address = 0x80

Register	Value
Current	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000000
CPSR	0x000000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	
PC $	0x00000000
Mode	Supervisor
States	0

```
    33:          LDR      R0, [R1, -R4, LSL #2]
    34:
    35:
0x00000048  E7110104  LDR      R0,[R1,-R4,LSL #2]
    36: STOP    B      STOP   ; infinite loop
0x0000004C  EAFFFFFE  B        0x0000004C
0x00000050  6C6C6548  STCVSL   p5,CR6,[R12],#-0x0120
0x00000054  6F57206F  SWIVS    0x0057206F
0x00000058  21646C72  ???CS
0x0000005C  00000000  ANDEQ    R0,R0,R0
0x00000060  00000001  ANDEQ    R0,R0,R1
0x00000064  00000002  ANDEQ    R0,R0,R2
```

```
            LDR     R1, =0x40000000 ; address
            LDR     R0, [R1, #4]
            LDR     R0, [R1], #4

            LDR     R0, [R1, -R4]
            LDR     R0, [R1, R4, LSL #2]


STOP        B       STOP                ; infinite loop

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?)
C2          DCW     1, 2, 3, 4, 5, 6, 7, 8              ; 8


            END
```

```
Running with Code Size Limit: 32K
Load "C:\\Courses\\CS1021\\Lecture Notes\\6 Memory\\test\\Objects\\test.axf"
```

```
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE
```

Address: 0x40

```
0x00000040: 04 00 91 E4 04 00 11 E7 04 01 11 E7 FE FF FF EA 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 00 00 00 00
0x00000060: 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00 06 00 00 00 07 00 00 00 08 00 00 00
0x00000080: 01 00 02 00 03 00 04 00 05 00 06 00 07 00 08 00 50 00 00 00 01 02 03 04 05 06 07 08 00 00 00 00
0x000000A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000140: 00 00 00 00                                                 00 00 00 00 00 00 00 00 00 00 00 00
0x00000160: 00 00 00 00                                                 00 00 00 00 00 00 00 00 00 00 00 00
```

Real-Time Agent: Target Reset   Simulation   t1: 0.00000000 sec   L:36 C:1   CAP NUM SCRL OVR R/W

MEMORY

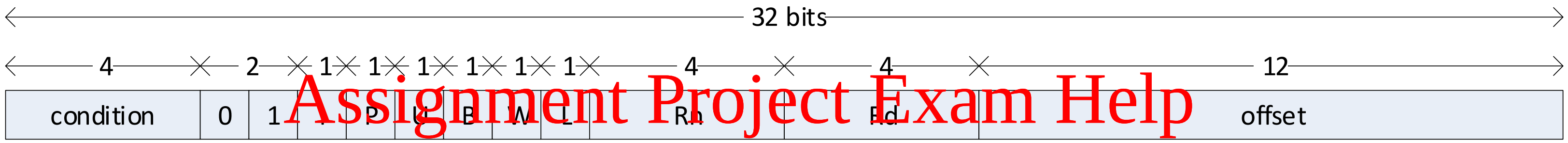
Additional features of LDR / STR instructions

- 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



32 bits

4	2		1	1	1	1	1	1	4	4	12
condition	0	1	I	P	U	B	W	L	Rn	Rd	offset

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 <u>after</u> transfer
  - if 1 (pre indexing), add offset to base register <u>before</u> 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

MEMORY

Examples

immediate offset

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

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

R0 = MEM[R1 + 4]

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

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

R0 = MEM[R1]
R1 = R1 + 4 ; post increment address register

can specify a +ve or -ve offset

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

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

R1 = R1 - 4 ; pre increment address register
R0 = MEM[R1]

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

MEMORY

Examples…

register offset

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

  R0 = 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 = 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 ; pre increment address register
  R0 = MEM[R1]

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

Memory

Examples…

scaled register offset

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

  R0 = 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 = MEM[R1]

  R1 = R1 + R4*4

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

can specify a +ve or -ve offset

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

  R1 = R1 - R4*4

  R0 = 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

Memory

Example 1: string copy

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

points to

```
        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
```

post increment R1

post increment R2

MEMORY

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
L0      LDRB    R0, [R2], #1            ; load next ch of string AND R2 = R2 + 1
        CMP     R0, #0                  ; 0?
        BNE     L0                      ; next ch
        SUB     R2, R2, #2              ; R2 -> last ch of string (excluding terminating 0)
```

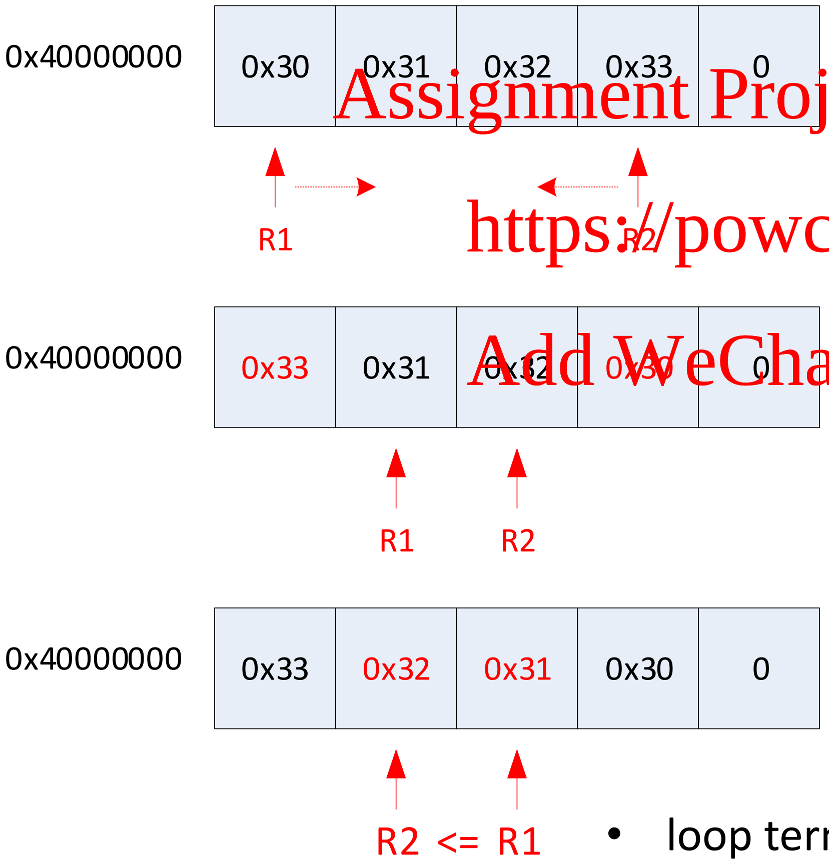
post increment R1

decrement R2 by 2 as R2 was one past NUL terminator

MEMORY

Example 3: reverse string …

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



0x40000000	0x30	0x31	0x32	0x33	0

0x40000000	0x33	0x31	0x32	0x30	0

0x40000000	0x33	0x32	0x31	0x30	0

- loop terminating condition

MEMORY

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

MEMORY

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]

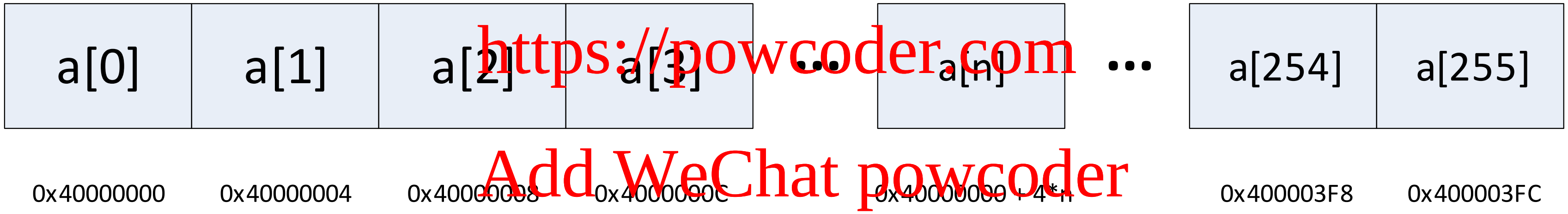
a[0] stored @ MEM[0x40000000] // increasing by 4…
a[1] stored @ MEM[0x40000004] // because each integer
a[2] stored @ MEM[0x40000008] // occupies 4 bytes of memory
…
a[n] stored @ MEM[0x40000000 + 4*n] //
…
a[255] stored @ MEM[0x400003FC] //

- array **a** occupies 256 x 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



a[0]	a[1]	a[2]	a[3]	…	a[n]	…	a[254]	a[255]
0x40000000	0x40000004	0x40000008	0x4000000C		0x40000000 + 4*n		0x400003F8	0x400003FC

- 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];

MEMORY

Example 4: Array Access…

constant offset

scaled register offset

- 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)
```

scaled register offset

MEMORY

What has not been covered?

LDRH	load halfword
STRH	store halfword
LDRSB	load byte with sign extend
LDRSH	load halfword with sign extend