

Computer organization and architecture

Lesson 7

Memory instructions

RISC architectures are **load/store architectures**.

Data in external memory must be brought into the processor using an instruction.

Instructions operate **exclusively** on registers.

Data stored in memory **must be moved to a register before it can be processed.**

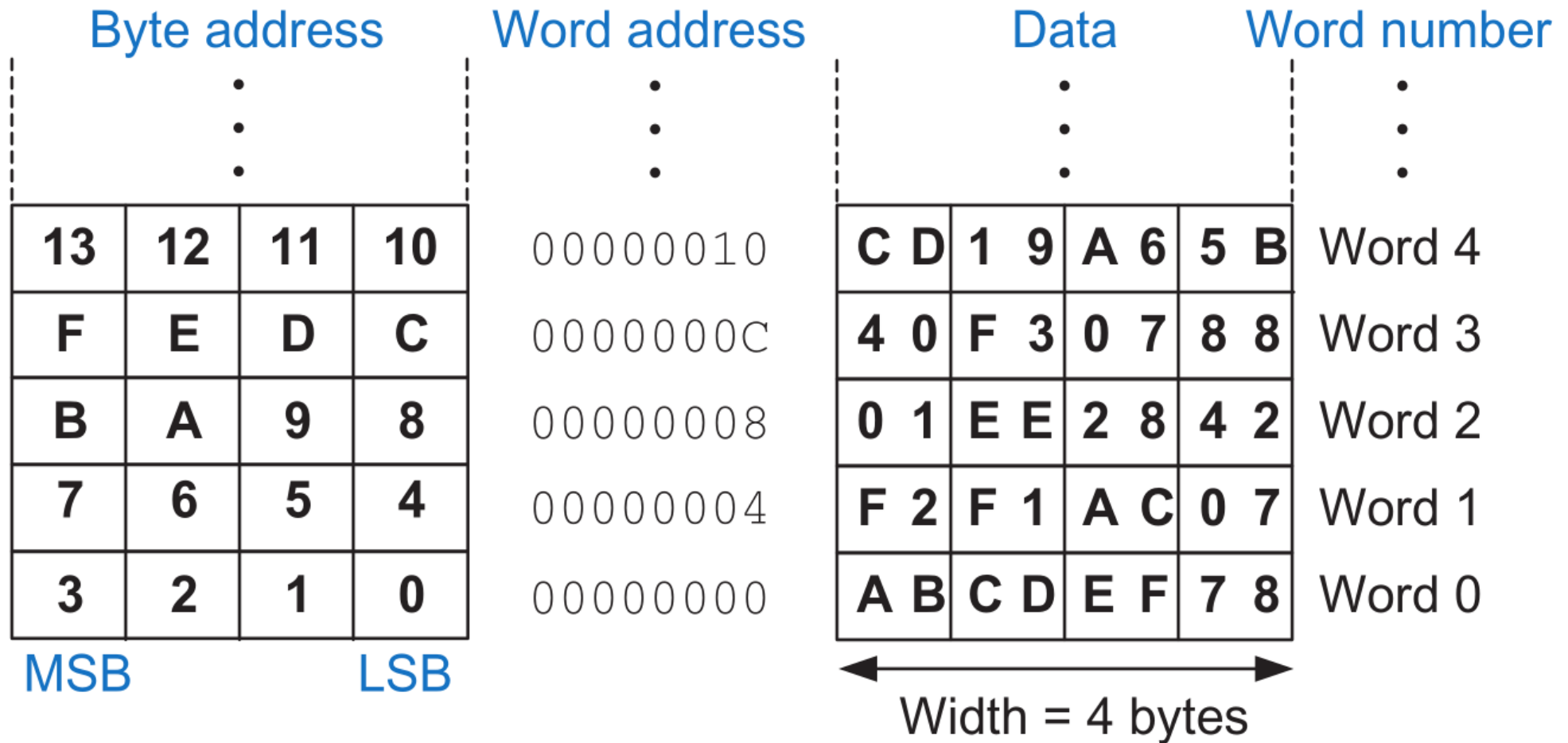
Operations that take a value in memory, multiply it by a coefficient, add it to another register, and then store the result back to memory with only a single instruction do not exist.

Memories are organized as an **array** of **data words**.

The ARM architecture uses 32-bit memory addresses and 32-bit data words.

ARM uses a **byte-addressable** memory.

Each byte in memory has a unique address.



Each word address is a multiple of 4

Reading memory

High-level

```
a = mem[2]; // index or word number = 2
```

ARM assembly

```
; R7 = a  
MOV R5, #0           ; base address = 0  
LDR R7, [R5, #8]      ; R7 <= data at  
                      ; memory address (R5+8)
```

LDR – **load register instruction**, reads a data word from memory into a register

The LDR instruction specifies the memory address using a base register (R5) and an **offset (8)**

Word number 2 is at address 8

A read from the base address (index 0) is a special case that requires no offset in the assembly code.

A memory read from the base address held in R5 is

```
LDR R7, [R5]
```

Writing memory

High-level

```
mem[5] = 42; // index or word number = 5
```

ARM assembly

```
MOV R1, #0           ; base address = 0
MOV R9, #42
STR R9, [R1, #0x14]   ; 42 stored at
                      ; memory address (R1+20)
```

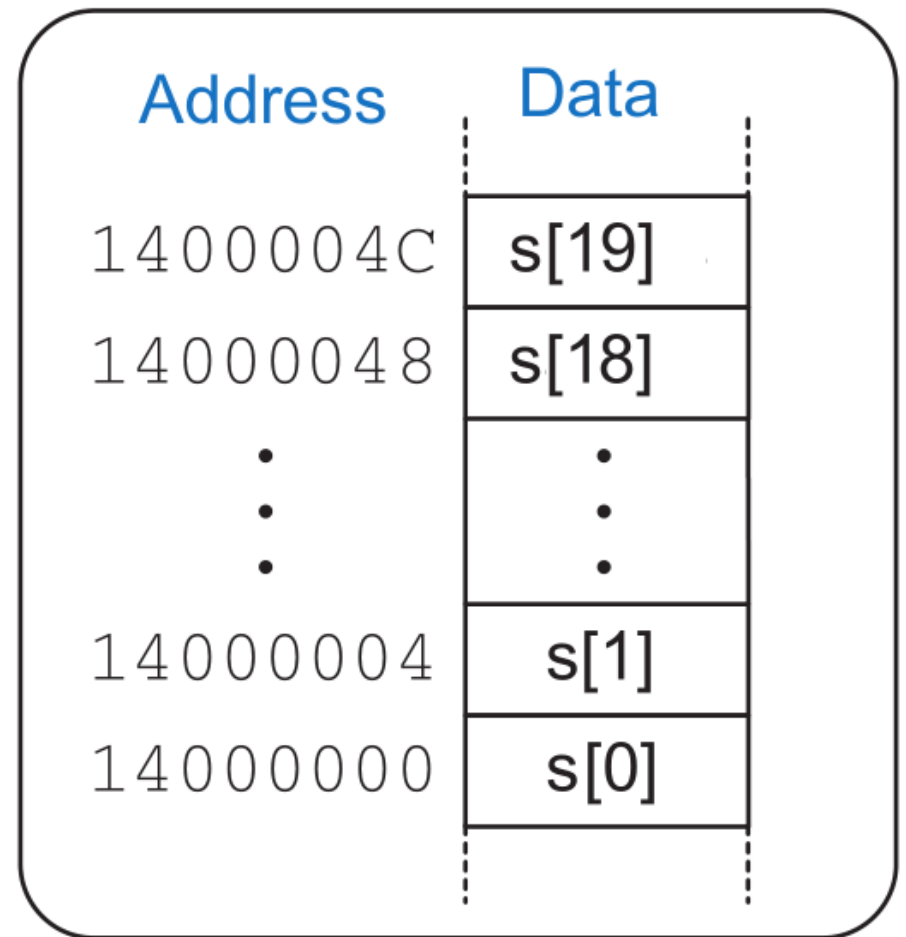
STR – **store register instruction**, writes a data word from a register into memory

Similar data can be grouped together into an **array**.

An array stores its contents at sequential data addresses in memory.

Each array element is identified by a number called its **index**.

The number of elements in the array is called the **length** of the array.



Example: a 20-element array


Main memory

Accessing arrays using a for loop

High-level

```
int i;  
int s[20];  
...  
for (i = 0; i < 20; i = i + 1)  
    s[i] = s[i] + 10;
```

ARM assembly

```
MOV R0, #0x40000000    ; R0 = base address  
MOV R1, #0             ; i = 0  
LOOP    
    CMP R1, #20         ; i < 20?  
    BGE L3             ; if i = 20, exit loop  
    LSL R2, R1, #2      ; R2 = i * 4  
    LDR R3, [R0, R2]    ; R3 = s[i]  
    ADD R3, R3, #10     ; R3 = s[i] + 10  
    STR R3, [R0, R2]    ; s[i] = s[i] + 10  
    ADD R1, R1, #1      ; i = i + 1  
    B LOOP             ; repeat loop  
L3
```


In the previous page, we needed to have the array be located in writable memory.

LPC2104 has 16 kB of on-chip static RAM.

Programming it requires that we know the starting address of RAM, which is 0x40000000.

RAM in LPC2104 has addresses between 0x40000000 and 0x40003FFF

See details in

http://www.nxp.com/documents/data_sheet/LPC2104_2105_2106.pdf

You can run the code from the previous page in Keil and see the content of the memory using
View → Memory Windows

ARM can **scale** (multiply) the index, add it to the base address, and load from memory in a single instruction.

Instead of

```
LSL R2, R1, #2           ; R2 = i * 4  
LDR R3, [R0, R2]         ; R3 = s[i]
```

we can use a single instruction:

```
LDR R3, [R0, R1, LSL #2]
```

Also,

```
STR R3, [R0, R1, LSL #2]
```

In addition to scaling the index register, ARM provides

- offset addressing
- pre-indexed addressing
- post-indexed addressing

ARM indexing modes

Mode	ARM Assembly	Address	Base Register
Offset	LDR R0, [R1, R2]	$R1 + R2$	Unchanged
Pre-index	LDR R0, [R1, R2]!	$R1 + R2$	$R1 = R1 + R2$
Post-index	LDR R0, [R1], R2	R1	$R1 = R1 + R2$

Mode	ARM Assembly	Address	Base Register
Offset	LDR R0, [R1, R2]	$R1 + R2$	Unchanged
Pre-index	LDR R0, [R1, R2]!	$R1 + R2$	$R1 = R1 + R2$
Post-index	LDR R0, [R1], R2	R1	$R1 = R1 + R2$

Offset addressing calculates the address as the base register \pm the offset.

The base register is unchanged.

Pre-indexed addressing calculates the address as the base register \pm the offset and updates the base register to this new address.

Post-indexed addressing calculates the address as the base register only and then, after accessing memory, the base register is updated to the base register \pm the offset.

for loop using
post-indexing

High-level

```
int i;  
int s[20];  
...  
for (i = 0; i < 20; i = i + 1)  
    s[i] = s[i] + 10;
```

ARM assembly

```
MOV R0, #0x40000000 ; R0 = base address  
ADD R1, R0, #80      ; R1 = base address + (20*4)  
LOOP  
  CMP R0, R1          ; reached end of array?  
  BGE L3              ; if yes, exit loop  
  LDR R2, [R0]         ; R2 = s[i]  
  ADD R2, R2, #10      ; R2 = s[i] + 10  
  STR R2, [R0], #4     ; s[i] = s[i] + 10  
                      ; then R0 = R0 + 4  
  B LOOP              ; repeat loop  
L3
```

To access individual bytes in memory:

LDRB – load byte

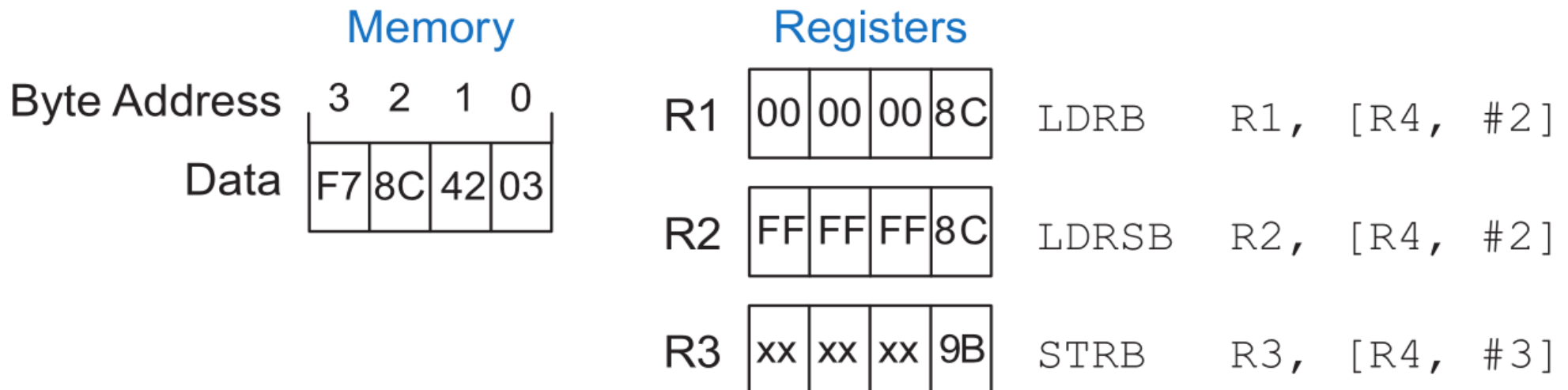
STRB – store byte

LDRSB – load signed byte

To fill the entire 32-bit register, LDRB zero-extends the byte

LDRSB sign-extends the byte

STRB stores the least significant byte of the 32-bit register into the specified byte address in memory.



LDRH, LDRSH, and STRH are similar to, but access 16-bit halfwords.

For halfword loads, the data is placed in the least significant halfword (bits [15:0]) of the register with zeros in the upper 16 bits.

For halfword stores, the data is taken from the least significant halfword.

Assuming the address in register r0 is 0x8000,

LDRH r11, [r0] ; load a halfword into r11

	Memory	Address
r11 before load	0xEE	0x8000
0x12345678	0xFF	0x8001
r11 after load	0x90	0x8002
0x0000FFEE	0xA7	0x8003

LDRSH r11, [r0] ; load signed halfword into r11

	Memory	Address
r11 before load	0xEE	0x8000
0x12345678	0x8C	0x8001
r11 after load	0x90	0x8002
0xFFFF8CEE	0xA7	0x8003

There are no signed stores of halfwords or bytes into memory.

Data stored to memory never needs to be sign extended.

Computers simply treat data as a sequence of bit patterns and must be told how to interpret numbers.

The value 0xEE could be a small, positive number, or it could be an 8-bit, two's complement representation of the number -18.

The LDRSB and LDRSH instructions provide a way for the programmer to tell the machine that we are treating the values read from memory as signed numbers.

```
GLOBAL Reset_Handler
AREA Reset, CODE, READONLY
```

```
ENTRY
```

```
Reset_Handler
```

```
    MOV    R0, #0                ; sum = 0
    MOV    R1, #9                ; number of elements-1
    ADR     R2, arraya           ; load start of array
again
    LDR     R3, [R2, R1, LSL #2] ; load value from memory
    ADD     R0, R0, R3           ; sum += a[i]
    SUBS    R1, R1, #1           ; i = i-1
    BGE     again               ; loop only if i > 0
```

```
stop B stop
```

```
    ALIGN
arraya DCD 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA
    END
```

DCD directive allocates one or more words of memory, aligned on four-byte boundaries

Exercise 7.1

Run the code from the page 8 of this lesson in Keil.

Observe the changes in memory.

Exercise 7.2

Rewrite the code from the page 8 using pre-indexing.

See the example for the same code using post-indexing on the page 13.

Exercise 7.3

Using the STRB command store the number 0xAB at the address #0x40000001, and the number 0xCD at the address #0x40000002.

Find these numbers in memory.

Load the number from the address #0x40000001 to R1 and zero-extend the byte.

Load the number from the address #0x40000002 to R2 and sign-extend the byte.

Exercise 7.4

Using STRH, store the number 0x1234 at the address #0x40000000

Find this number in memory.

Change the code so that this number would be stored at #0x40000001

Which places in memory are occupied by 0x1234

Try the same for the addresses #0x40000002, #0x40000003, #0x40000004, #0x40000005

What interesting phenomenon do you observe?

Exercise 7.5

Store the array [-1, -2, -3, -4, -5] in memory using the DCD directive.

Write an assembly code to calculate the sum

$$-1 - 2 - 3 - 4 - 5 = -15 = 0 \text{ } xFFFFFFF \text{ } 1$$

Find where exactly each element of this array is stored in the memory.

When you add the elements step by step, notice how each element is highlighted in red in Keil.

Identify, each highlighted element.

Exercise 7.6

Consider the following code snippet.

```
int i;  
int a[5];  
for (i=0; i < 5; i=i+1)  
    a[i] = i;
```

Assume R0 holds i and that R1 holds the base address of the array.

Write the code snippet using ARM assembly language.

Then load this array from memory, and put a[0] to R6, a[1] to R7, a[2] to R8, a[3] to R9, a[4] to R10.

Exercise 7.7

Find the maximum value in a list of 32-bit values located in memory.

Assume the values are in two's complement representations.

Your program should have 15 values in the list.