# COMP 3221

# Microprocessors and Embedded Systems

## Lecture 12: Memory Access - II

http://www.cse.unsw.edu.au/~cs3221

August, 2003

Saeid Nooshabadi

Saeid@unsw.edu.au

## Overview

- Word/ Halfword/ Byte Addressing
- Byte ordering
- Signed Load Instructions
- Instruction Support for Characters

## Review: Assembly Operands: Memory

- C variables map onto registers; what about large data structures like arrays?
- 1 of 5 components of a computer: memory contains such data structures
- But ARM arithmetic instructions only operate on registers, never directly on memory.
- Data transfer instructions transfer data between registers and memory:
  - Memory to register
  - Register to memory

## Review: Data Transfer: Memory ←→ Reg

- Example: `ldr a1, [v1, #8]`
- Example: `ldr a1, [v1, v2]`

Similar instructions For STR

- Example: `ldr a1, [v1,#12]!`

Pre Indexed Load: Subsequently, `v1` is updates by computed sum of `v1` and 12, ( `v1 ← v1 + 12`).

- Example: `ldr a1, [v1, v2]!`

Pre Indexed Load: Subsequently, `v1` is updates by computed sum of `v1` and 12, ( `v1 ← v1 + v2`).

- Example: `ldr a1, [v1],#12`

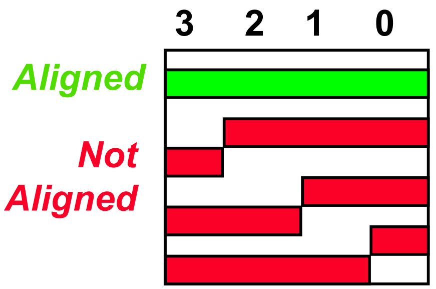
Post Indexed Load: Subsequently, `v1` is updates by computed sum of `v1` and 12, ( `v1 ← v1 + 12`).

- Example: `ldr a1, [v1], v2`

Post Indexed Load: Subsequently, `v1` is updates by computed sum of `v1` and 12, ( `v1 ← v1 + v2`).

## Review: Memory Alignment

- ARM requires that all words start at addresses that are multiples of 4 bytes



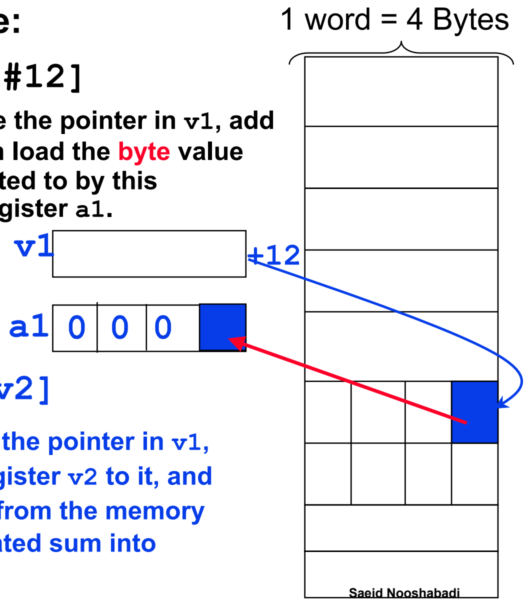
- Called Alignment: objects must fall on address that is multiple of their size.
- Some machines like Intel allow non-aligned accesses

## Data Transfer: More Mem to Reg Variants (#1/2)

- Load Byte Example:

```
ldrb a1, [v1,#12]
```

This instruction will take the pointer in `v1`, add 12 bytes to it, and then load the byte value from the memory pointed to by this calculated sum into register `a1`.



- Load Byte Example:

```
ldrb a1, [v1, v2]
```

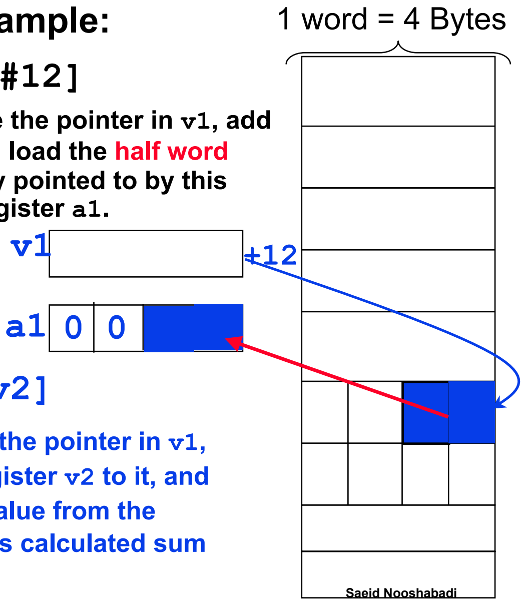
This instruction will take the pointer in `v1`, add an index offset in register `v2` to it, and then load the byte value from the memory pointed to by this calculated sum into register `a1`.

## Data Transfer: More Mem to Reg Variants (#2/2)

- Load Half Word Example:

```
ldrh a1, [v1,#12]
```

This instruction will take the pointer in `v1`, add 12 bytes to it, and then load the half word value from the memory pointed to by this calculated sum into register `a1`.



- Load Byte Example:

```
ldrh a1, [v1, v2]
```

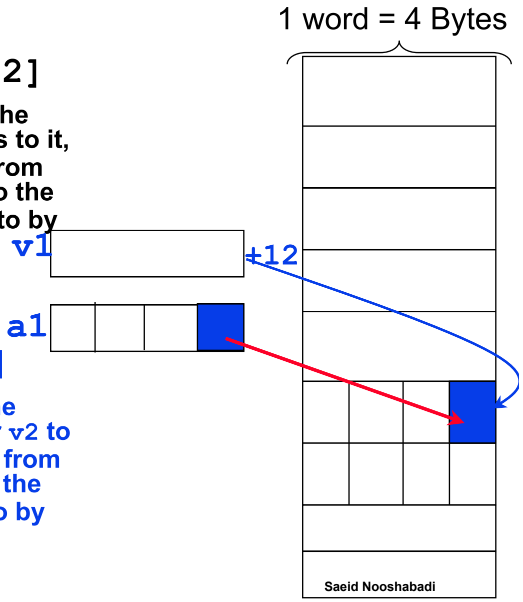
This instruction will take the pointer in `v1`, add an index offset in register `v2` to it, and then load the half word value from the memory pointed to by this calculated sum into register `a1`.

## Data Transfer: More Reg to Mem Variants (#1/2)

- Store Byte Example:

```
strb a1, [v1,#12]
```

This instruction will take the pointer in `v1`, add 12 bytes to it, and then store the value from lsb Byte of register `a1` into the memory address pointed to by the calculated sum.



- Store Byte Example:

```
strb a1,[v1, v2]
```

This instruction will take the pointer in `v1`, adds register `v2` to it, and then store the value from lsb Byte of register `a1` into the memory address pointed to by the calculated sum.

## Data Transfer: More Reg to Mem Variants (#2/2)

° Store Half Word Example:

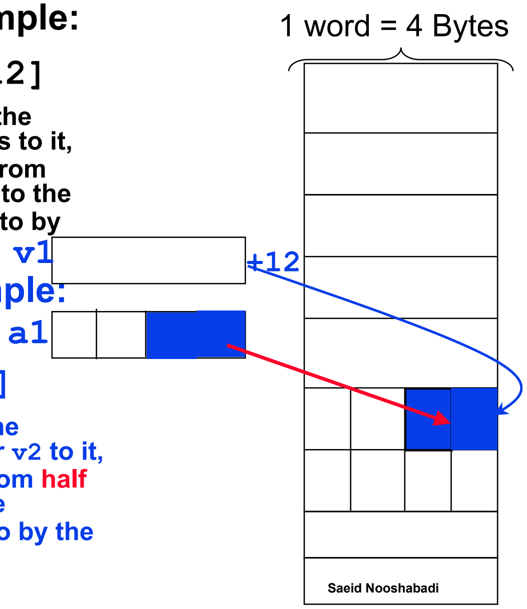
`strh a1, [v1,#12]`

This instruction will take the pointer in `v1`, add 12 bytes to it, and then store the value from half word of register `a1` into the memory address pointed to by the calculated sum.

° Store Half Word Example:

`strh a1,[v1, v2]`

This instruction will take the pointer in `v1`, adds register `v2` to it, and then store the value from half word of register `a1` into the memory address pointed to by the calculated sum.



## Compilation with Memory (Byte Addressing)

° What offset in `ldr` to select `my_Array[8]` (defined as `Char`) in C?

° 1x8=8 to select `my_Array[8]`: byte

° Compile by hand using registers:

```
    g = h + my_Array[8];
```

- g: `v1`, h: `v2`, `v3`:base address of `my_Array`

° 1st transfer from memory to register:

```
ldrb v1, [v3,#8] ; v1 gets my_Array[8]
```

- Add 8 to `r3` to select `my_Array[8]`, put into `v1`

° Next add it to h and place in g

```
add v1,v2,v1     ; v1 = h+ my_Array[8]
```

## Compilation with Memory (half word Addressing)

° What offset in `ldr` to select `my_Array[8]` (defined as halfword) in C?

° 2x8=16 to select `my_Array[8]`: byte

° Compile by hand using registers:

```
    g = h + my_Array[8];
```

- g: `v1`, h: `v2`, `v3`:base address of `my_Array`

° 1st transfer from memory to register:

```
ldrh v1, [v3, #16] ; v1 gets my_Array[8]
```

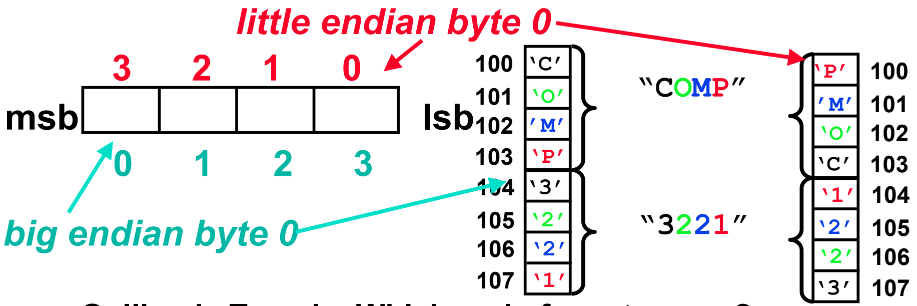
- Add 16 to `r3` to select `my_Array[8]`, put into `v1`

° Next add it to h and place in g

```
add v1,v2,v1     ; v1 = h+ my_Array[8]
```

## More Notes about Memory: Word

° How are bytes numbered in a word?



•Gulliver's Travels: Which end of egg to open?

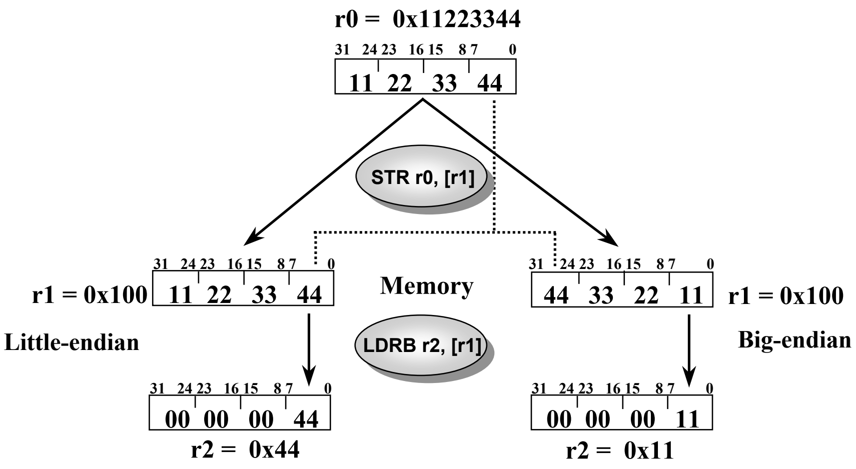
Cohen, D. "On holy wars and a plea for peace (data transmission)." *Computer*, vol.14, (no.10), Oct. 1981. p.48-54.

•Little Endian address of least significant byte: Intel 80x86, DEC Alpha,

•Big Endian address of most significant byte HP PA, IBM/Motorola PowerPC, SGI, Sparc

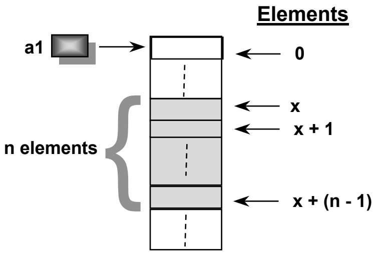
•ARM is Little Endian by default, However it can be made Big Endian by configuration.

# Endianess Example



# Code Example

- Write a segment of code that add together elements x to x+(n-1) of an array, where the element x = 0 is the first element of the array.
- Each element of the array is word sized (ie. 32 bits).
- The segment should use post-indexed addressing.
- At the start of your segments, you should assume that:
  - a1 points to the start of the array.
  - a2 = x
  - a3 = n



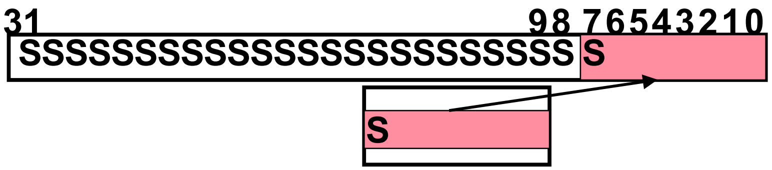
# Code Example: Sample Solution

```
add a1, a1, a2, lsl #2      ; Set a1 to address
                            ; of element x
add a3, a1, a3, lsl #2      ; Set a3 to address
                            ; of element x +(n-1)
mov a2, #0                  ; Initialise
                            ;accumulator
Loop:
ldr a4, [a1], #4            ; Access element and
                            ; move to next
add a2, a2, a4              ; Add contents to
                            ; counter
cmp a1, a3                  ; Have we reached
                            ; element x+n?
blt loop                    ; If not - repeat
                            ; for next element
                            ; on exit sum
                            ; contained in a2
```

# Sign Extension and Load Byte & Load Half Word

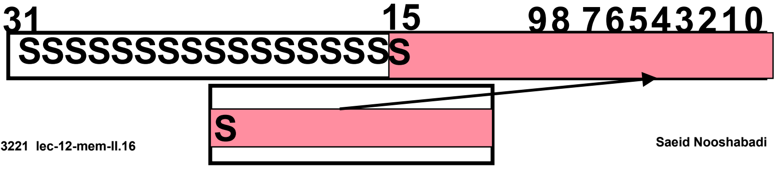
- ARM instruction (`ldrsb`) automatically extends "sign" of byte for load byte.

`ldrsb a1, [v1,#12]` `ldrsb a1, [v1,v2]`



- ARM instruction (`ldrsh`) automatically extends "sign" of half word for load half word.

`ldrsh a1, [v1,#12]` `ldrsh a1, [v1,v2]`



## Instruction Support for Characters

- ARM (and most other instruction sets) include instructions to operate on bytes:
  - move byte (`ldrb`) loads a byte from memory/reg, placing it in rightmost 8 bits of a register, or vice versa
- Declares byte variables in C as “`char`”
- Assume `x`, `y` are declared `char`. `x` in memory at `[v1,#4]`and `y` at `[v1,#0]`.
  What is ARM code for `x = y;` ?

```
ldrb a1, [v1,#0]

strb a1, [v1,#4]   ; transfer  y to x
```

## Strings in C: Example

- String simply an array of char

```
void strcpy (char x[], char y[]){
int i = 0; /* declare,initialize i*/

while ((x[i] = y[i]) != ’\0’) /* 0 */
 i = i + 1; /* copy and test byte */
}
```

- function

i, addr. of `x[0]`, addr. of `y[0]`: `v1, a1, a2` , func ret addr. `:lr`

```
strcpy:
     mov v1, #-1              ; i = -1
 L1: add v1, v1, #1           ; i =i + 1
     ldrb a3, [a2,v1]         ; a1= y[i]
     strb a3, [a1,v1]         ; x[i]=y[i]
     cmp a3, #0
     bne L1                   ; y[i]!=0
                              ;goto L1
     mov pc, lr               ; return
```

## Strings in C: Example using pointers

- String simply an array of char

```
void strcpy2 (char *px, char *py){

while ((*px++ = *py++) != ’\0’) /* 0 */
         ; /* copy and test byte */
}
```

- function

addr. of `x[0]`, addr. of `y[0]`: `v2, v3` func ret addr.`:lr`

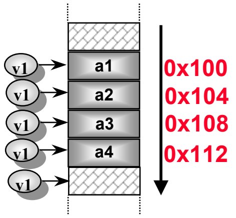
```
strcpy:
 L1: ldrb a1, [v3],#1      ;a1= *py, py = py +1
     strb a1, [v2],#1      ;*px = *py, px = px +1
     cmp a1, #0
     bne L1                ; py!=0 goto L1
     mov pc, lr            ; return
```

- ideally compiler optimizes code for you

## Block Copy Transfer (#1/5)

- Consider the following code:

```
str a1, [v1],#4
str a2, [v1],#4
str a3, [v1],#4
str a4, [v1],#4
```

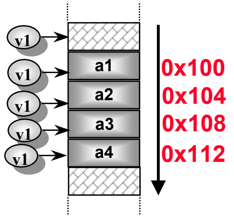


Replace this with
`stmia v1!, {a1-a4}`

STMIA : SORE MULTIPLE INCREMENT AFTER

- Consider the following code:

```
str a1, [v1, #4]!
str a2, [v1, #4]!
str a3, [v1, #4]!
str a4, [v1, #4]!
```



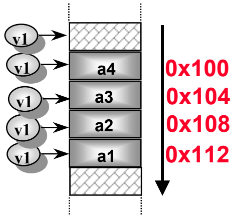
Replace this with
`stmib v1!, {a1-a4}`

STMIB : SORE MULTIPLE INCREMENT BEFORE

# Block Copy Transfer (#2/5)

° Consider the following code:

```
str a1, [v1],#-4
str a2, [v1],#-4
str a3, [v1],#-4
str a4, [v1],#-4
```



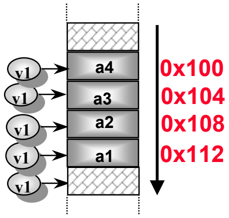
Replace this with

**stmda v1!, {a1-a4}**

**STMDA : SORE MULTIPLE DECREMENT AFTER**

° Consider the following code:

```
str a1, [v1, #-4]!
str a2, [v1, #-4]!
str a3, [v1, #-4]!
str a4, [v1, #-4]!
```



Replace this with

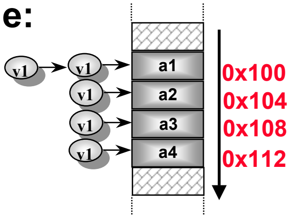
**stmdb v1!, {a1-a4}**

**STMDB : SORE MULTIPLE DECREMENT BEFORE**

# Block Copy Transfer (#3/5)

° Consider the following code:

```
str a1, [v1]
str a2, [v1,#4]
str a3, [v1,#8]
str a4, [v1,#12]
```



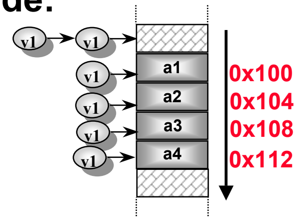
Replace this with

**stmia v1, {a1-a4}**

**STMIA : SORE MULTIPLE INCREMENT AFTER**

° Consider the following code:

```
str a1, [v1, #4]
str a2, [v1, #8]
str a3, [v1, #12]
str a4, [v1, #16]
```



Replace this with

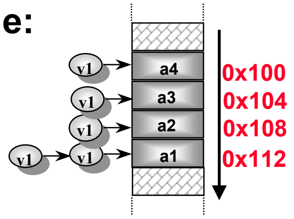
**stmib v1, {a1-a4}**

**STMIB : SORE MULTIPLE INCREMENT BEFORE**

# Block Copy Transfer (#4/5)

° Consider the following code:

```
str a1, [v1]
str a2, [v1,#-4]
str a3, [v1,#-8]
str a4, [v1,#-12]
```



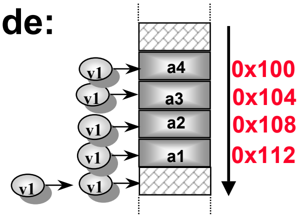
Replace this with

**stmda v1, {a1-a4}**

**STMDA : SORE MULTIPLE DECREMENT AFTER**

° Consider the following code:

```
str a2, [v1,#-4]
str a3, [v1,#-8]
str a4, [v1,#-12]
str a1, [v1,#16]
```



Replace this with

**stmdb v1, {a1-a4,}**

**STMDB : SORE MULTIPLE DECREMENT BEFORE**

# Block Data Transfer (#5/5)

° Similarly we have

- LDMIA : Load Multiple Increment After
- LDMIB : Load Multiple Increment Before
- LDMDA : Load Multiple Decrement After
- LDMDB : Load Multiple Decrement Before

**For details See Chapter 3, page 61 – 62**
**Steve Furber: ARM System On-Chip; 2nd Ed,**
**Addison-Wesley, 2000, ISBN: 0-201-67519-6.**

## COMP3221 Reading Materials (Week #4)

- Week #4: Steve Furber: ARM System On-Chip; 2nd Ed, Addison-Wesley, 2000, ISBN: 0-201-67519-6. We use chapters 3 and 5
- ARM Architecture Reference Manual –On CD ROM

## “And in Conclusion…” (#1/2)

- In ARM Assembly Language:
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster
- Memory is byte-addressable, but `ldr` and `str` access one word at a time.
- Access byte and halfword using `ldrb`, `ldrh`,`ldrsb` and `ldrsh`
- A pointer (used by `ldr` and `str`) is just a memory address, so we can add to it or subtract from it (using offset).

## “And in Conclusion…”(#2/2)

- New Instructions:

```
ldr, str
ldrb, strb
ldrh, strh
ldrsb, ldrsh
```