Computer Architecture (Practical Class)
Assembly: Array Allocation and Access

Luís Nogueira

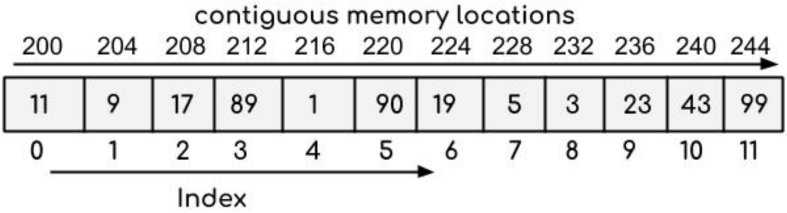
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2022/2023

Arrays

- Contiguously allocated memory region
- All n elements are of the same type
- Each element occupies the number of bytes determined by its data type
- Therefore, the size of the array is given by $n * sizeof(type)$



Array declaration: Examples in C

```c
char array_a[12];
char *array_b[8];
long array_c[6];
long *array_d[5];
int  array_e[] = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60};
```

Array	Element size (bytes)	Total size (bytes)
array_a	1	12
array_b	8	64
array_c	8	48
array_d	8	40
array_e	4	44

Declaring arrays in C and Assembly

Array declaration in C

```c
// uninitialized array
long array_c[6];

// initialized array
int array_e[] = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60};
```

Array declaration in Assembly

```asm
.section .bss          # section BSS (uninitialized variables)

.comm array_c, 48      # space for 6 long (6x8 bytes), name: array_c

.section .data         # section data (initialized variables)

array_e:               # name: array_e
    .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60 # array initialization
```

Declaring strings (arrays of `char`) in C and Assembly

String (array of char) declaration in C

```c
// uninitialized string (array of char)
char str_a[10];

// initialized string (array of char)
char str_b[] = "computer architecture";
```

String (array of char) declaration in Assembly

```
.section .bss         # section BSS (uninitialized variables)

.comm str_a, 10       # space for 10 bytes; variable name: str_a

.section .data        # section data (initialized variables)

str_b:                # name: str_b
    .asciz "computer architecture"  # string initialization
```

Arrays - Accessing values

Using a pointer in C to access a value from an array

```c
int vec[] = {1,2,3,4,5}

// ptr is a pointer to an array of int
int *ptr = vec;

// assign x the value pointed by ptr
int x = *ptr;
```

What is the equivalent Assembly code?

- We have seen that an array is a continuous area in memory, where each element occupies the number of bytes determined by its data type
- We can store memory addresses in registers
- When a register stores a memory address, it is equivalent to a pointer in C

Address computation with *leaq*

leaq Src, Reg

- Computes the effective address of a memory location (first operand) and stores it in a general-purpose register (second operand)
- Has the form of an instruction that reads from memory to a register, but it does not reference memory at all
- Instead of reading from the designated location, the instruction copies the effective address to the destination (a register)
- This instruction is then used to generate pointers for later memory references
- In addition, it can be used to compactly describe common arithmetic operations (more on this on lecture classes)

Important notes

- Since an address is 8 bytes in x86-64, the *lea* instruction has no other variant than *leaq*

Address computation with *leaq*

Consider the following C code

```c
int vec[]={1,2,3,4,5};

int* get_vec_address()
{
  return vec;
}
```

Corresponding Assembly code

```
.section .data
  .global vec

.section .text
  .global get_vec_address

get_vec_address:
  leaq vec(%rip),%rax
  ret
```

Indirect addressing

- When we use the register as a pointer by placing it between parentheses

```
movX (register),destination
```

Important notes

- Byte addressing (addresses are manipulated byte by byte)
- To move the pointer to the next element, we must know the size of the element
- *Must* use the correct `mov` variant (`movq`, `movl`, `movw`, `movb`) according to the number of bytes that you want to copy
- We cannot do `movX (%rbx),(%rax)!` — Why?

Example: Accessing the 10th element of an array of `int`

Return the value of the 10th element

```
# copy the value of the 10th element of array_a to EAX
.section .data

array_a:
        .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60

.section .text

.global tenth_element
tenth_element:
    leaq array_a(%rip), %rdi    # copy the address of array_a to %rdi
    addq $36, %rdi              # offset of 10th element ((10-1)*4=36)
    movl (%rdi),%eax            # copy the value pointed by %rdi to %eax
                                # this will be our return value
    ret
```

Example: Accessing the 5th element of a string (an array of `char`)

Return the value the 5th char

```
# copy the value of the 5th char of str to AL
.section .data

str:
        .asciz "computer architecture"

.section .text

.global fifth_char
fifth_char:
    leaq str(%rip), %rdi    # copy address of str to rdi
    addq $4, %rdi           # offset of fifth element
    movb (%rdi), %al        # copy the char value pointed by %rdi to %al
                            # this will be our return value
    ret
```

Indirect Addressing with Offset

- When we add a constant value (positive or negative) to the address expression that indicates the offset relative to the base address in a register

```
movX offset(register),destination
```

Example:

```
movl 8(%rdi),%eax     # %rdi is a pointer to an array of integers
```

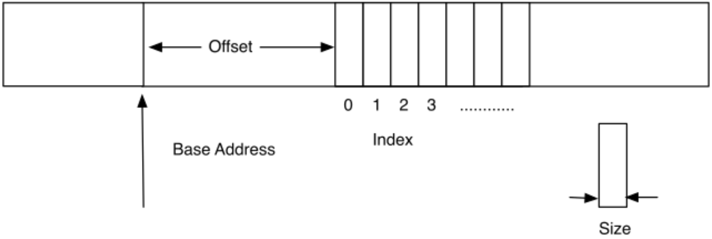
- Places, in the EAX register, the 4-byte value contained in memory, 8 bytes after the location pointed by the RDI register
- It is equivalent to:

```
addq $8, %rdi     # add 8 to the address stored in RDI
movl (%rdi),%eax
subq $8, %rdi     # the value of RDI is restored
```

Indexed memory mode (1/3)

When accessing data in an array, one can use an index system to determine which value will be accessed. For this, we define:

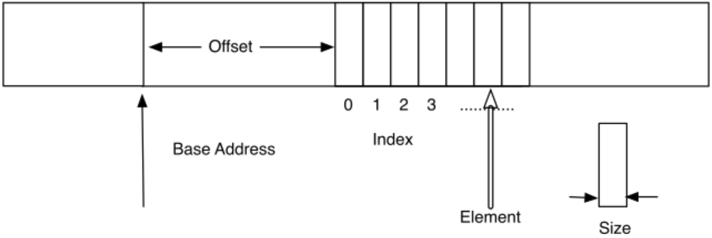
1. `base_address`: a pointer to the start of the memory we want to address
2. `offset`: a displacement value between the beginning of the memory and when we start counting elements
3. `size`: the size of each element
4. `index`: the index of the element we want to access



Indexed memory mode (2/3)

The final address is then calculated by the following expression:

`base_address+offset+index*size`



- This is what the *indexed memory mode* does `offset (base_address, index, size)`

- `base_address`: A base address (register)
- `offset`: An offset value to add to the base address (literal)
- `index`: An index to determine which data element to select (register)
- `size`: The size of the data element (literal with value 1, 2, 4, or 8)

Notes:

- If any of the values are zero, they can be omitted (but the commas are still required as placeholders).
- If we omit the `base_address`, the `offset` can be an absolute address

Indexed Memory Mode Example

```
.section .data

# declare integer array named 'array_a'
array_a:
    .int 10, 15, 20, 25, 30, 35

.section .text

example:
    # address of array_a in %rdi
    leaq array_a(%rip), %rdi

    # index to access: 3rd element
    movq $2, %rcx

    # copy array_a[2] to %eax
    movl (%rdi, %rcx, 4), %eax
    ret
```

Practice

What is the Assembly code that implements the expression given?

Assume:

- an array `int array_e[10]`, with its initial address in `%rdi`
- the value of i (an integer) was previously copied to `%rcx`

Expression	Type	Assembly
$\%rax \leftarrow$ array_e	`int *`	
$\%eax \leftarrow$ array_e[0]	`int`	
$\%eax \leftarrow$ array_e[i]	`int`	
$\%rax \leftarrow$ &array_e[i]	`int *`	
$\%eax \leftarrow$ *(array_e+i-3)	`int`	

Practice

What is the Assembly code that implements the expression given?

Assume:

- an array `int array_e[10]`, with its initial address in `%rdi`
- the value of i (an integer) was previously copied to `%rcx`

Expression	Type	Assembly
$\%rax \leftarrow$ array_e	`int *`	movq %rdi, %rax
$\%eax \leftarrow$ array_e[0]	`int`	movl (%rdi), %eax
$\%eax \leftarrow$ array_e[i]	`int`	movl (%rdi, %rcx, 4), %eax
$\%rax \leftarrow$ &array_e[i]	`int *`	leaq (%rdi, %rcx, 4), %rax
$\%eax \leftarrow$ *(array_e+i-3)	`int`	movl -12(%rdi, %rcx, 4), %eax

Example: Increment by one all the elements of an array of ints

Increment by one all the elements of an array of ints

```
.section .data

    .global vec
vec:
    .int 1,2,3,4,5,6,7,8,9,10


.equ SIZE,10

.section .text

    .global inc_by_one
inc_by_one:
    leaq vec(%rip), %rdi     # store the vec address in %rdi
    movq $SIZE, %rcx         # store the number of elements of vec in %rcx
    cmpq $0, %rcx            # validate size of vec
    jle end

inc_loop:
    incl (%rdi)              # increment value pointed by %rdi
    addq $4, %rdi            # point to next int
    loop inc_loop
end:
    ret
```

Example: Count the number of chars in a string

Count the number of chars in a string

```
.section .data

str:
        .asciz "computer architecture 101"

.section .text

        .global str_count
str_count:
        leaq str(%rip), %rdi     # get address of string
        movl $0, %eax            # counter = 0
str_loop:
        movb (%rdi), %cl         # get char pointed by %rdi
        cmpb $0, %cl             # end of string?
        je end_str_loop
        incl %eax                # increase counter
        incq %rdi                # point to next char
        jmp str_loop
end_str_loop:
        ret
```

Practice: `str_copy()` in Assembly

- Implement a function `int str_copy()` that copies a string from `ptr1` to `ptr2` (two global pointers to char). Test it by calling your function from a C program.
- The function should return the number of chars copied
- Consider that the variables are declared in assembly and initialized in C.

str_cpy.h

```c
#ifndef _STR_CPY_ // avoid duplicate definitions
#define _STR_CPY_

// maximum number of chars in a string
#define MAX_CHAR 20

// function implemented in Assembly
int str_copy();

// pointers declared in Assembly
extern char *ptr1, *ptr2;

#endif
```

main.c

```c
#include <stdio.h>     // for printf()
#include "str_copy.h" // definition of MAX_CHAR, ptr1, ptr2, str_copy()

int main (void){
    char str1[MAX_CHAR] = "abcdef";
    char str2[MAX_CHAR];     // important: str2 must have space for str1
    int n_chars;

    // assign address of strings to global pointers, defined in assembly
    ptr1 = str1;
    ptr2 = str2;

    // call function str_copy(), implemented in Assembly
    n_chars = str_copy();

    // output results
    printf("Copied %d chars\n",n_chars);
    printf(" str1 = %s\n", str1);
    printf(" str2 = %s\n", str2);

    return 0;
}
```

str_copy.s

```
.section .bss
.global ptr1, ptr2
  .comm ptr1,8          # declare pointer (8 bytes)
  .comm ptr2,8          # declare pointer (8 bytes)

.section .text
.global str_copy
str_copy:
    movq  ptr1(%rip), %rsi      # we want the value of the pointer
    movq  ptr2(%rip), %rdi      # not its address
    movl  $0, %eax              # counter = 0

str_loop:
    movb (%rsi), %cl      # copy char from str1 (pointed by %rsi) to %cl
    movb %cl, (%rdi)      # copy char in %cl to str2 (pointed by %rdi)
    cmpb $0, %cl          # check if this is the end of the string
    jz str_loop_end       # jump if it is the end
    incl %eax             # counter ++
    incq %rsi             # move to the next char in str1
    incq %rdi             # move to the next char in str2
    jmp str_loop          # next iteration

str_loop_end:             # return value (counter) in %eax
    ret
```