Computer Architecture (Practical Class)
Assembly: Arrays, Strings and Loops

Luís Nogueira Raquel Faria

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

{lmn,arf}@isep.ipp.pt

2020/2021

Arrays - Review (1/5)

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 $n * sizeof(type)$

Arrays - Review (2/5)

Array declaration: Examples in C

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

- What is the size of each element?
- What is the total size of the array?

Arrays - Review (3/5)

Array declaration: Examples in C

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

- What is the size of each element?
- What is the total size of the array?

Array	Element size	Total size
array_a	1	12
array_b	4	32
array_c	8	48
array_d	4	20
array_e	4	44

Declaring arrays in C and Assembly

Array declaration in C

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

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

Array declaration in Assembly

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

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

```
// 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 identifier

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

.section .data         # section identifier

str_b:                 # variable name
    .asciz "computer architecture"
```

Arrays - Accessing values (1/5)

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

Arrays - Accessing values (2/5)

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 (`movl`, `movw`, `movb`) according to the number of bytes that you want to copy
- We cannot do `movl (%ebx),(%eax)` — Why?

Arrays - Accessing values (3/5)

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

Return the value of the 10th element (`http://codepad.org/vb74YeOs`)

```
# 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       # declare function as global
tenth_element:              # function start
# prologue
    pushl %ebp              # save previous stack frame pointer
    movl %esp, %ebp         # the stack frame pointer for our function

# body
    movl $array_a, %edx     # copy the address of array_a to %edx
    addl $36, %edx          # move to 10th element ((10-1)*4=36)
    movl (%edx),%eax        # copy the value pointed by %edx to %eax
                            # this will be our return value
# epilogue
    movl %ebp, %esp         # restore the stack pointer ("clear" the stack)
    popl %ebp               # restore the stack frame pointer
    ret                     # return from the function
```

Arrays - Accessing values (4/5)

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

Return the value the 5th char (`http://codepad.org/jy5lYOqP`)

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

str:
        .asciz "computer architecture"

.section .text

.global fifth_char
fifth_char:
# prologue
    pushl %ebp              # save previous stack frame pointer
    movl %esp, %ebp         # the stack frame pointer for our function

# body
    movl $str, %edx         # copy address of str to edx
    addl $4, %edx           # move to the fifth element
    movb (%edx),%al         # copy the char value pointed by edx to al

# epilogue
    movl %ebp, %esp         # restore the stack pointer ("clear" the stack)
    popl %ebp               # restore the stack frame pointer
    ret                     # return from the function
```

Arrays - Accessing values (5/5)

Indirect Addressing with Offset

- When we add a constant value (positive or negative) to the address stored in a register

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

Example:

```
movl 4(%edx),%eax     # edx is a pointer to an array of integers
```

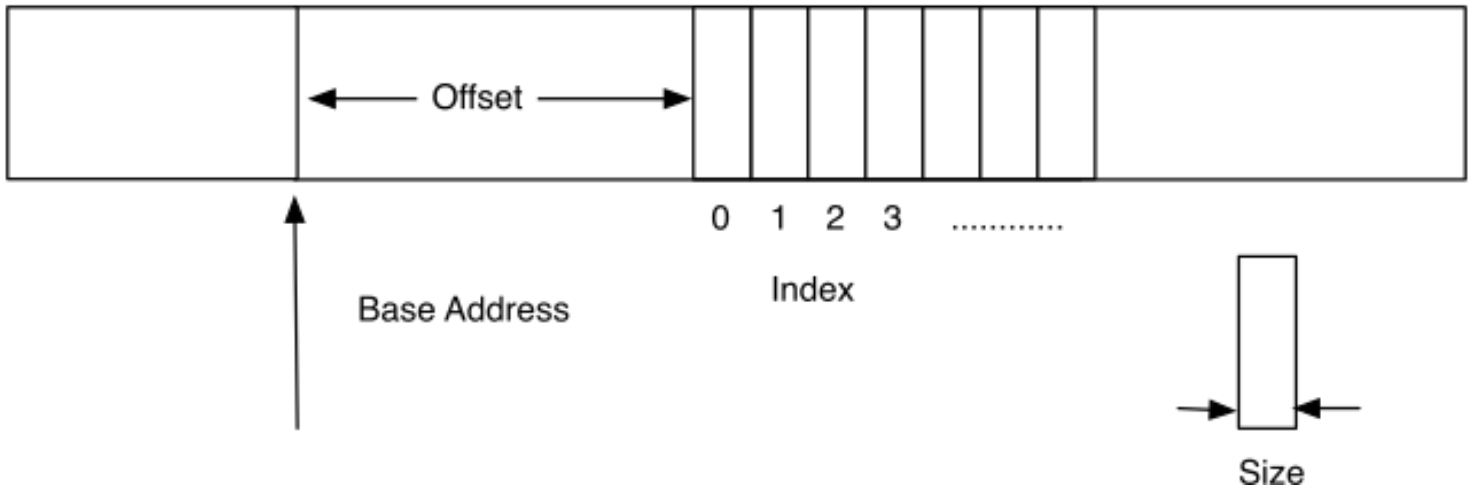
- Places the value contained in the memory location 4 bytes after the location pointed to by the EDX register in the EAX register
- It is equivalent to:

```
addl $4, %edx
movl (%edx),%eax
subl $4, %edx     # the value of EDX is not modified
```

Indexed memory mode (1/3)

When accessing data in an array, you can use an index system to determine which value you are accessing. For this, we define:

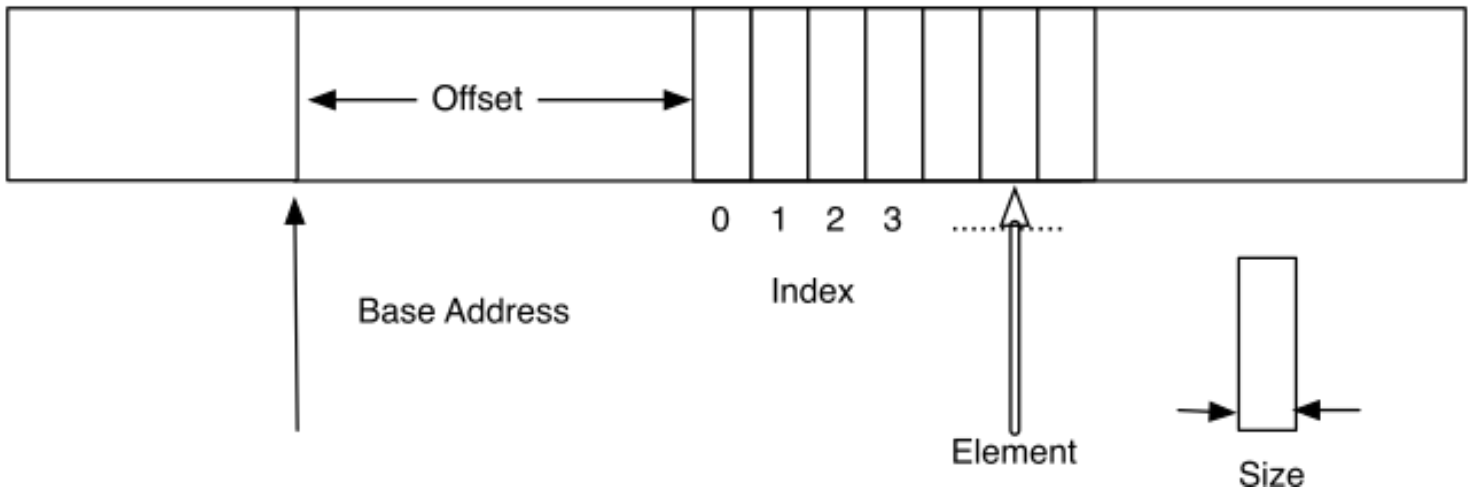
1. `base_address`: a pointer to the start of the memory we want to address
2. `offset`: a displacement value between the beginning of the memory and when we start counting elements
3. `size`: the size of each element
4. `index`: the index of the element we want to access



Indexed memory mode (2/3)

To calculate the address, calculate the expression:

```
base_address+offset+index*size
```



- This is what the *indexed memory mode* does

Indexed memory mode (3/3)

`offset (base_address, index, size)`

- `base_address`: A base address (register)
- `offset`: An offset address 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 identifier
.section .data

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

# section identifier
.section .text

# function implementation
function_example:
        ...

        # index to access: 3rd element
        movl $2, %ecx

        # move value to %eax
        # note: base_address is omitted
        movl array_a(, %ecx, 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 `edx`
- the value of *i* (an integer) was previously copied to `ecx`

Expression	Type	Assembly
eax ← array_e	`int *`	
eax ← array_e[0]	`int`	
eax ← array_e[i]	`int`	
eax ← *(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 `edx`
- the value of *i* (an integer) was previously copied to `ecx`

Expression	Type	Assembly
eax ← array_e	`int *`	`movl %edx,%eax`
eax ← array_e[0]	`int`	`movl (%edx),%eax`
eax ← array_e[i]	`int`	`movl array_e(, %ecx, 4), %eax`
eax ← *(array_e+i-3)	`int`	`movl -12(%edx,%ecx,4),%eax`

The while loop in Assembly

While loop example

```
# function that implements a while(i < limit) loop
# using conditional jumps
function_example:
  ...

  # function body
  movl limit, %eax    # limit value in %eax
  movl $0, %edx       # iterating index in %edx (i=0)

  my_loop:            # loop start
    cmpl %eax, %edx   # compare index with limit value
    jge end_my_loop   # jump if i >= limit

    ...               # loop body

    addl n, %edx      # increments index (i+=n)
    jmp my_loop

  end_my_loop:

  ...

  ret
```

Example: Iterate through a string

Iterate through a string

```
.section .data
str:
        .asciz "computer architecture"

.section .text
.global iterate_string # declare function as global
iterate_string:        # function start
# prologue
  ...

# body
  movl  $str, %edx     # address of string in %edx (notice the $)
string_loop:
  movb  (%edx),%cl     # copy char pointed by %edx to %cl
  cmpb  $0,%cl         # check if char is zero (end of string)
  je    end_loop       # no more chars in string

  ...                  # do something with char...

  incl   %edx          # moves pointer to next 1 byte
  jmp    string_loop   # jumps to next iteration

end_loop:              # we reached the end of the string...

# epilogue
  ...
```

Example: Count the number of chars in a string

Count the number of chars in a string (`http://codepad.org/xXe07szh`)

```
.section .data          # section identifier (data)
str:                    # declare string
    .asciz "computer architecture"

.section .text          # section identifier (text)
str_count:              # function start
# prologue
    pushl %ebp          # save previous stack frame pointer
    movl %esp, %ebp     # the stack frame pointer for our function
# body
    movl  $str,%edx     # copy str address to %edx (notice the $)
    movl  $0,%eax       # counter = 0
cnt_loop:
    movb (%edx),%cl     # copy char from str1 (pointed by %edx) to %cl
    cmpb $0,%cl         # check if this is the end of the string
    jz cnt_loop_end     # jump if it is the end
    incl %eax           # counter ++
    incl %edx           # move to the next char in str1
    jmp cnt_loop        # next iteration
cnt_loop_end:
    # note: return value (counter) in %eax
# epilogue
    movl %ebp, %esp     # restore the stack pointer ("clear" the stack)
    popl %ebp           # restore the stack frame pointer
    ret                 # return from the function
```

The `loop`, `loope`, `loopz`, `loopne`, and `loopnz` instructions

- `loop` instructions provide iteration control and combine loop index management with conditional branching
- Prior to using the loop instruction, load the `%ecx` register with an unsigned iteration count
- Then, add the `loop` instruction at the end of a series of instructions to be iterated
- The loop instrucyion automatically decrements `%ecx` and jumps to the label if `%ecx` is different from 0

Important notes:

- The `loop` instructions test the flags, but do not change them
- What will happen if the `%ecx` register is zero before the first call to any `loop` instruction?

The loop, loope, loopz, loopne, and loopnz instructions

loop instruction example

```
    ...
    movl $100, %ecx
my_loop:
    ...
    loop my_loop
    ...
```

- loop automatically decrements %ecx and jumps to the label if %ecx is different from 0
- loope/loopz: decrements %ecx and jumps to the label if %ecx is different from 0, *and the flag ZF is active*
- loopne/loopnz: decrements %ecx and jumps to the label if %ecx is different from 0, *and the flag ZF is* ***not*** *active*

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.

Practice: str_copy() in Assembly - C source (1/2)

str_cpy.h (http://codepad.org/fSwYx1fu)

```
#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
```

Practice: str_copy() in Assembly - C source (2/2)

main.c (http://codepad.org/kGOcWc48)

```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;
}
```

Practice: str_copy() in Assembly - Assembly source (1/2)

str_copy.s (http://codepad.org/ymqsdnL3)

```
.section .bss                  # section identifier (bss)

.comm    ptr1,4                # declare pointer (4 bytes)
.comm    ptr2,4                # declare pointer (4 bytes)
.global ptr1                   # declare ptr1 global so that it can be used in C
.global ptr2                   # declare ptr2 global so that it can be used in C

.section .text                 # section identifier (text)

.global str_copy               # declare function as global so that it can be used in C
str_copy:                      # function start
# prologue
    pushl %ebp                 # save previous stack frame pointer
    movl %esp, %ebp            # the stack frame pointer for our function
    push %esi                  # stores current value of %esi in stack
    push %edi                  # stores current value of %esi in stack
# body
    movl  ptr1,%esi            # copy str1 address to %esi
    movl  ptr2,%edi            # copy str2 address to %esi
    movl  $0,%eax              # counter = 0
```

Practice: str_copy() in Assembly - Assembly source (2/2)

str_copy.s (http://codepad.org/ymqsdnL3)

```
str_loop:
    movb (%esi),%cl      # copy char from str1 (pointed by %esi) to %cl
    movb %cl,(%edi)      # copy char in %cl to str2 (pointed by address in %edi)
    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 ++
    incl %esi            # move to the next char in str1
    incl %edi            # move to the next char in str2
    jmp str_loop         # next iteration

str_loop_end:            # loop end; note: return value (counter) in %eax
# epilogue
    pop %edi             # restore %edi
    pop %esi             # restore %esi
    movl %ebp, %esp      # restore the stack pointer ("clear" the stack)
    popl %ebp            # restore the stack frame pointer
    ret                  # return from the function
```