Computer Organization and Design (ECE 219)
Winter Semester 2020-2021

Laboratory 2

Objectives of the workshop

- Stack usage
- Functions

Stack

During the execution of a program, a part of the memory is reserved and allocated for the specific process. This piece of memory is divided into certain sectors, each of which stores a specific type of data. As can be seen in figure 1, the sectors where the memory is divided are the following:

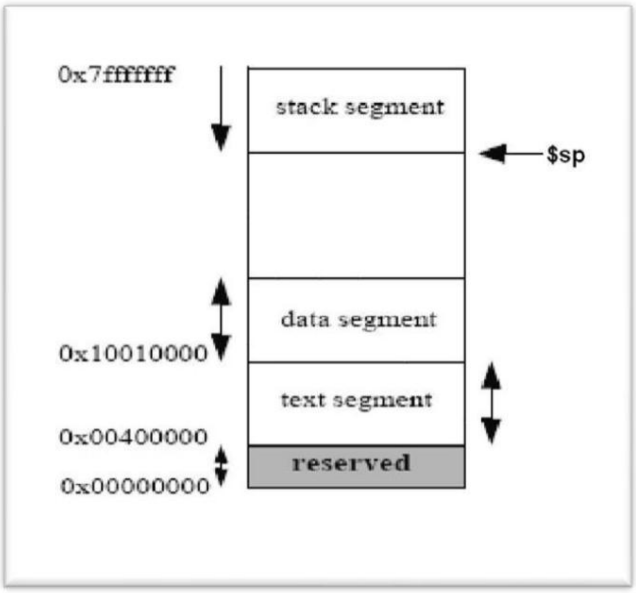


Figure 1 (memory segments)

Reserved: Reserved by the operating system area of memory where your program cannot access.

Text Segment: Area of memory where program commands are stored. These are machine language commands. When the code you have written does not consist of so-called pseudo-commands but of real commands then for each command you have written, the corresponding system command in binary format has been stored in this part of the memory.

Data Segment: Area of memory where program data is stored. All variables, arrays, and generally what you have declared in the .data area of your code is stored in this piece of memory. If we analyze more we will see that it is divided into 2 pieces. The static data segment where the static data is stored (those declared in .data) and the dynamic data segment where it is the extra space dynamically allocated by the operating system.

Stack Segment: Area of memory used by the stack. The boundaries of this area are defined by the $sp pointer which you can change at runtime. On the contrary, the boundaries of the data segment are determined by the operating system and to allocate additional memory you must call the corresponding system function (malloc command in C).

Empty segment: A segment between the data segment and the stack segment which, although it does not contain data, is reserved and allocated by the operating system to the specific process. If during execution the stack area or data area needs to be increased then they will take extra space from this chunk.

Using the stack

First of all when we refer to a stack we mean a LIFO (Last In First Out) list. This means that we can each time retrieve the elements in the reverse order from which we saved them (first we retrieve the last elements entered in the list). From Figure 1 you see that to increase the space occupied by the stack we need to move the $sp pointer down. To do this we must **reduce** its value by as many places as the extra bytes we want the stack to increase. Because as a rule we save the contents of registers on the stack, which have a size of 4 bytes, the extra space on the stack must be ensured to be a multiple of 4 without this being binding.

Why use the stack?

During the execution of a program the registers are all visible from any point of the code, even from the various functions. But this can create a problem when running a program. When a particular register is used for writing inside a function (let's say A) and at the same time contains data necessary for another function (let's say B) which at some point of its execution called A then the data of this register is lost for the function B.

To better understand the problem that is created, let's look at the following example:

```
Func1:
```

```
…

li $s0.50            # command #1: In the register $s0 data # necessary for Func1
                     is stored. Let the value be 50.
…

jal       Func2              # call Func2. Execution jumps to Func2:

…

move $a0, $s0 # Problem. The data in $s0 is not that of # command
                 #1. The function expected to find 50.
…

Func2:               #function func2

…

addi $s0, $zero,100      #The value 100 is stored in the register $s0.
                       #Any data previously inside is lost.
…

jr$ra
```

In the above example we see the problem created with the $s0 register. Both func1 and func2 use this register. The problem is created in func1 which after calling func2 lost the contents it had in $s0. One could argue that in such cases by being careful about which registers
to use we can avoid such unpleasant side effects. But what about when you're asked to implement a function that another developer will later use? Or also what if you need to implement large code where function calls are very frequent?

How to solve the problem using a stack.

The process to store something on the stack is as follows:

1. Decrement the $sp index by as many places as the number of registers you want to store. Each register corresponds to 4 bytes. Suppose that space is reserved for N words. Decrease by N*4 bytes.
2. Store with the command sw the contents of the registers you will use in the memory locations from where $sp points up to plus N*4 locations.

3. Use registers normally inside the function.
4. Load with the lw command the initial contents of the registers respectively as you stored them in step 2
5. Increment $sp by N*4 positions so that it returns to the position it was before step 1.
6. The function returns.

Example of using the stack

Below is the implementation of a function that saves what is needed on the stack.

Example:

```
func1:
addi $sp,$sp,-12            #allocate positions for three words on the stack.
sw $ra,0($sp)               #push the contents of $ra onto the stack
sw $t1,4($sp)               #push the contents of $t1 onto the stack
sw $t2,8($sp)               #push the contents of $t2 onto the stack

 …

addi $t1, $0.10 #we change without restrictions the contents of the 2
move $t2, $t1 #registers we pushed

 …

jal func2                   #we can call others without a problem
            # functions
 …

jal func1                   #we can even call her.
                            #function(recursively) without any problems
 …

lw $ra, 0($sp) #pop the contents of $ra off the stack
lw $t1,4($sp) #pop the contents of $t1 off the stack
lw $t2,8($sp) #pop the contents of $t2 off the stack
addi $sp, $sp,12#reset the stack pointer
jr$ra
```

In the MIPS processor there are a large number of registers, most of which have a specific use. Registers $s0-$s7 and $t0-$t9 are called general-purpose and are most often used primarily for inter-register operations. You can see from table 1 which is also in the Instructions Set how you should treat each register according to the group it belongs to.

Register	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Used for retum values from function calls.
v1	3	
a0	4	Used to pass arguments to procedures and functions.
a1	5	
a2	6	
a3	7	
t0	8	Temporary (Caller-saved, need not be saved by called procedure)
t1	9	
t2	10	
t3	11	
t4	12	
t5	13	
t6	14	
t7	15	
s0	16	Saved temporary (Callee-saved, called procedure must save and restore)
s1	17	
s2	18	
s3	19	
s4	20	
s5	21	
s6	22	
s7	23	
t8	24	Temporary (Caller-saved, need not be saved by called procedure)
t9	25	
k0	26	Reserved for OS kernel
k1	27	
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address for function calls.

Table 1

Registers $t0 to $t9 are called Temporary or Caller-saved. This tells us that a function is not bound to keep the contents of registers the same when return. If the Caller, which can be another function or main, wants to have the contents of these registers and after calling one of the functions it has to save them on the stack itself.

Registers $s0 $s7 are called Saved Temporary or Callee-Saved. This means that a function must keep the contents of these registers constant with what they had before. To do this it must first save their contents on the stack and restore them when finished. This is exactly what happens in the example.

For the next workshop you have to implement the following exercises. During the workshop you will be tested orally on the codes you will deliver.

Exercise 1 Symmetrical matrix (5 units)

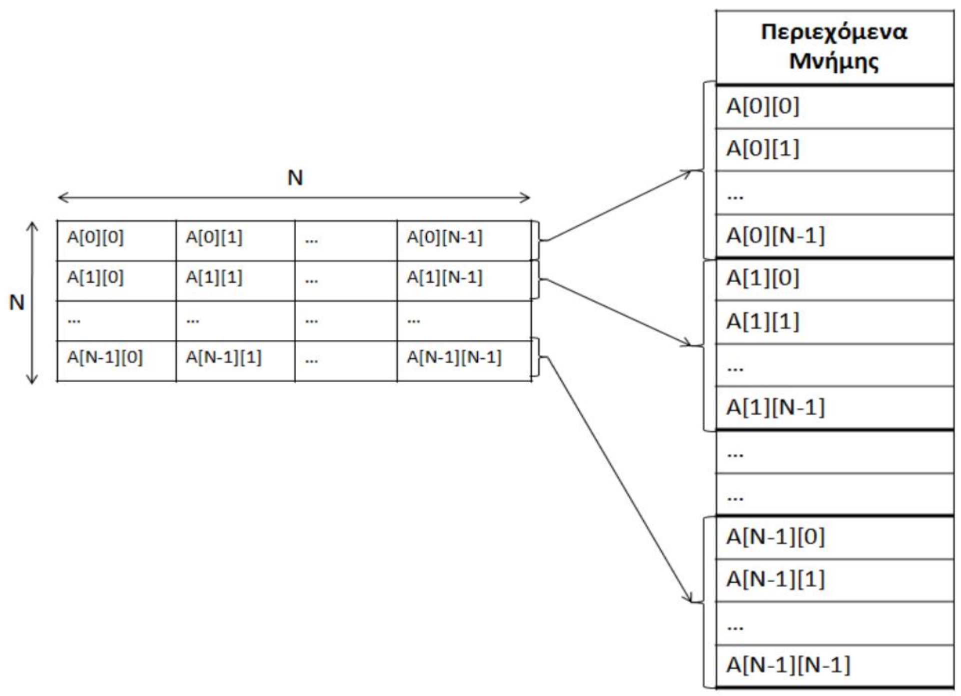
Develop a function *int IsSymmetric(int *A[], int N)* that will test whether the square matrix *A[N][N]* is symmetric. main will first ask the user for the size *N* of the square matrix *NxN* and then the elements to be stored in *N*

2 positions of this array, by row. The function will return 1 if the array is symmetric, otherwise it will return 0.

Suggestion:

One-dimensional arrays are stored in memory so that element A[i] is immediately after element A[i-1] of the array. Storing a two-dimensional array A is done by storing row i-1 immediately before row i.

Example of a two-dimensional array in memory (NxN):



Example run:

```
Please give the size N of the square matrix: 2
Please give the elements row-wise: 1

2
2
1
Issymetric: 1
-- program is finished running --
```

```
Please give the size N of the square matrix: 3
Please give the elements row-wise: 1

2
3
4
5
6
7
8
9
Issymetric: 0
-- program is finished running --
```

Comments:
1) In red is the input of the user.
2) **Do not print** the last line. MARS prints it by default.

Exercise 2 – Substring (4 units)

To implement the function *int substring(const char *str, const char *substr)* which will takes as arguments the string *str,* and the string *substr.* The function will search for the maximum part of *substr* (always starting from the beginning of substring) found in the string str and will return the maximum length of *substr* found in *str.*

Example:
str: **“abriaaklrivcariver”**
substr:“river”
The function will return 5.
Part of *substr* present in *str* is ri, riv, and river. But the maximum piece of *substr* that exists in *str* is the river, so the function returns 5.

The user will give from the console the initial string, and the substring we are looking for, and as an output it will output the maximum length of substring found.

Execution examples:

```
Please give string: aafgbba
Please give substring: fab
The max substring has length: 1
-- program is finished running --

Please give string: kalmacc
Please give substring: vbd
The max substring has length: 0
-- program is finished running --

Please give string: alloavcllof
Please give substring: llof
The max substring has length: 4
 -- program is finished running --
```

Comments:
1) In red is the input of the user.
2) **Do not print** the last line. MARS prints it by default.

Exercise 3 – Binding Registers (2 credits)

```
int x, y,
z; int *p = &x;

x = 10;
*p = 12;
y = x+1;
z = x-3;
```

Suppose you are developing the part of a MIPS compiler that implements the register allocator part of MIPS. The register allocator determines at each time which variables will be placed in each of the existing registers of a processor. The register allocator prefers to place in registers variables that will be used very often in the near future.

Let the variable x be such a case. Would you place the x variable in a register in the code above? What is the problem (if any)?

Hint: The answer has to do with the pointer.