# Informatics 2C Computer Systems - Lab 2

## 1 Introduction

The objective of this lab is to explore some of the useful features of the MIPS assembly language.

### 1.1 Arrays

MIPS assembly language does not have a special notion of an array like higher level programming languages. Arrays, and in fact all data structures, are just blocks of memory in MIPS which need to be allocated explicitly. In this lab, we will be looking at arrays as a representative data structure.

An array is a homogeneous data structure with all elements of the same type and size. The elements are laid out in a contiguous block of memory. The base address of the array corresponds to the address of the first element. In MIPS, an array can be allocated statically in the data segment in the following manner:

```
.data
# <array name/label>:  <.type> <initial values/number of elements>
arr1:  .word 1,2,3
arr2:  .space 40
```

In the example above `arr1` is an array of type `word` containing 3 elements initialized to `1,2,3`. Note that sufficient space (12 bytes) is automatically allocated by MARS. `arr2` is an array of 40 bytes uninitialized. You can find the complete list of `.type` directives in the appendix of the Patterson and Hennessy course textbook.

### 1.2 Jumps and Branches

As discussed in class, `jump` (unconditional) and `branch` (conditional) instructions in MIPS are employed to alter the flow control in a program by changing the PC value. The usage of `jump` and `branch` instructions is as follows:

```
.text
...
# <jump> <label>
j jmplabel
...
jmplabel:
...
```

```
.text
...
# <branch> <condition> <label>
li $s0 , 1
li $t1 , 1
beq $t1 , $s0 , brtakenlabel
...
brtakenlabel:
..
```

### 1.3 Methods

Just like higher-level programming languages, modular code design is considered good practice in MIPS assembly language. It enables code reuse, reducing code length considerably and aids debugging via proper structuring. In MIPS assembly language, methods are represented as code segments with a label. The method call and return can be implemented through `jal` and `jr` instructions (types of jump instructions) as demonstrated below.

```
.text
...
# push required reg values on stack ($t0 in this case)
addi $sp , $sp , -4
sw $t0 , 0($sp)
jal foo # method call
...
...
foo:
...
jr $ra # return
# pop reg values from stack ($t0 in this case)
lw $t0 , 0($sp)
addi $sp , $sp , 4
```

As discussed in class, the MIPS register convention on method call is:

- Method parameters: $a0-$a4
- Return values: $v0,$v1
- Registers preserved across call boundaries: $s0-$s7
- Registers not preserved across call boundaries: $t0-$t9,$ra

A stack in memory can be employed to save the register value not preserved automatically. Care must be taken to push all useful, un-preserved register values to the stack **before** a method call. This technique can also be employed for nested method calls. $ra register is overwritten upon a nested method call. The return address for the earlier method call must, therefore, be saved on the stack prior to the next method call.

## 2 MIPS Assembly Examples

**memread.s** This is a small assembly program that demonstrates the use of the MIPS features discussed earlier. Download the source file provided and open it in MARS.

The array, `inp`, is created and initialized in the data segment. The array is processed using a method `readarr`. Method call and return mechanism is achieved using `jal` and `jr` instructions. `readarr` reads `inp` from memory and prints it out. This is performed element-by-element using a loop.

## 3 Task

Using memread.s as the starting point, create a MIPS assembly program that reads from one array in memory and copies it to another array. The functionality is expressed as C language code below.

```
int out[16];
int i;
for(i=0 ; i<16 ; i++) {
      out[i] = inp[i];
}
```

Use the following steps to help guide you in creating the MIPS assembly program:

- Add a second array, `out`, of the same type and size as `inp`.
- Add a method `initarr`, which initializes all elements of the array `out` with zeroes at the beginning of program execution.

- Modify the method `copyarr`, such that it copies each element of `inp` to the corresponding elements of `out`.