Computer Systems and Networks

ECPE 170 – University of the Pacific

MIPS Assembly
(Functions)

Program Counter

Program Counter

- Instructions are stored in memory sequentially
- Each MIPS32 instruction occupies 4 bytes
- How does the processor know from where to fetch the next instruction?
  - A special 32-bit register called Program Counter (`PC`) holds the address of the next instruction

Program Counter in Action

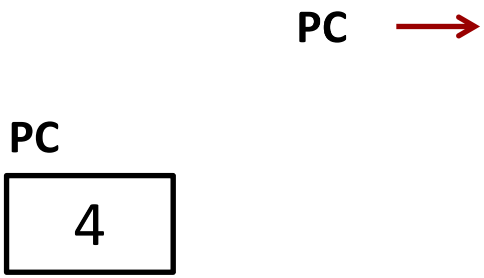
What is the C code for this MIPS Assembly?

```
t0 = 0;
t1 = 2;

while(t0<t1)
{
    t1++;
}

//exit
```

Address	Instruction
4	`addi $t0, $zero, 0`
8	`addi $t1, $zero, 2`
12	`bge $t0, $t1, <label to addr 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

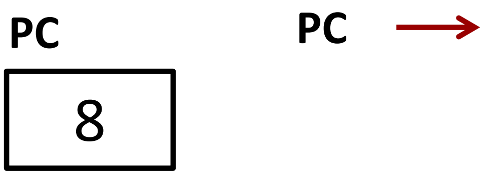
Program Counter in Action

Instructions are stored in memory and each occupy 4 bytes



Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

Program Counter in Action



PC

8

PC

Instructions are stored in memory and each occupy 4 bytes

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

Program Counter in Action

PC

12

Instructions are stored in memory and each occupy 4 bytes

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

PC →

Program Counter in Action

Instructions are stored in memory and each occupy 4 bytes

PC

16

PC →

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

Program Counter in Action

PC

20

Instructions are stored in memory and each occupy 4 bytes

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr 24>`
16	`addi $t0, $t0, 1`
PC → 20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

Program Counter in Action

PC

12

PC →

Instructions are stored in memory and each occupy 4 bytes

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

Program Counter in Action

PC

16

Instructions are stored in memory and each occupy 4 bytes

Address	Instruction
4	addi $t0,$zero,0
8	addi $t1,$zero, 2
12	bge $t0, $t1, <label to addr 24>
16	addi $t0, $t0, 1
20	j <label to addr. 12>
24	li $v0, 10
28	syscall

PC → 16

Program Counter in Action

PC

20

Instructions are stored in memory and each occupy 4 bytes

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

PC →

Program Counter in Action

PC

12

PC →

Instructions are stored in memory and each occupy 4 bytes

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

Program Counter in Action

PC

24

Instructions are stored in memory and each occupy 4 bytes

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

PC → 24

MIPS Functions

Function Requirements?

- **What happens when we call a function?**
  1. Place function arguments in standard location where function can find them
  2. Save current program location to return to later (the “Program Counter” register)
  3. Jump to the function location
  4. Function runs using provided arguments
  5. Function produces output (return value) and saves it in standard location
  6. Jump to original program location (return)
     1. *Technically, +1 instruction*

Function Requirements

- **Can a function change local variables of its calling function?**
- No! The function operates in its own “bubble”
- **What happens if the function changes $s0 which was also used by the calling function?**
- Problem! Your function has corrupted the calling function

Functions in Assembly

In assembly, **you** must do all the background work for functions that the compiler did automatically in a higher level language

Functions still allow for **code re-use** (good!), but they're more complicated than in C or C++

Registers

Name	Use
`$zero`	Constant value: ZERO
`$s0-$s7`	Local variables (Convention: These are *saved* if a function needs to re-use them)
`$t0-$t9`	Temporary results (Convention: These are *not saved* if a function needs to re-use them)
`$a0-$a3`	Arguments to pass to function (max of 4)
`$v0-$v1`	Return value to obtain from function (max of 2)
`$ra`	Return address of function
`$sp`	Stack pointer (current top of stack)

New!

More Jumps

- Jump and Link
  (side effect: `$ra` stores address of next instruction)

```
jal <destination>
```

Use this to *call* a function!

- Jump Register
  (destination address is stored in <reg1>

```
jr <reg1>
```

Use this to *return from* a function!

Basic Functions in MIPS

1. Program saves the context (registers) of calling function (caller)

2. Program saves the arguments in registers (`$a0 - $a3`)

3. Program calls the callee via jump-and-link instruction

```
jal <function label>
```

`jal` saves the address of the next instruction in return address reg., `$ra`

Program Counter (PC) points to the callee's location.
Callee saves return values in regs `$v0-$v1`

4. Callee returns via jump register instruction

```
jr <register name> #usually $ra
```

`jr` sets PC to `$ra`. PC continues there onwards

Problem 1 : Write Code

```
#include <stdio.h>

int function(int a);

int main()
{
  int x=5;
  int y;

  y = function(x);

  printf("y=%i\n", y);

  return 0;
}

int function(int a)
{
  return 3*a+5;
}
```

- Place arguments in `$a0-$a3`
- Place return values in `$v0-$v1`
- Return address saved automatically in `$ra`
- Ignore the stack for this example. (Thus, the function will destroy registers used by calling function)

P1

```
# Simple routine to demo functions
# NOT using a stack in this example.
# Thus, the function does not preserve values
# of calling function!

# ------------------------------------------------------------------

        .text

        .globl  main
main:
        # Register assignments
        # $s0 = x
        # $s1 = y

        # Initialize registers
        lw      $s0, x          # Reg $s0 = x
        lw      $s1, y          # Reg $s1 = y

        # Call function
        move    $a0, $s0        # Argument 1: x ($s0)
        jal     fun             # Save current PC in $ra, and jump to fun
        move    $s1,$v0         # Return value saved in $v0. This is y ($s1)

        # Print msg1
        li      $v0, 4          # print_string syscall code = 4
        la      $a0, msg1
        syscall

        # Print result (y)
        li      $v0,1           # print_int syscall code = 1
        move    $a0, $s1        # Load integer to print in $a0
        syscall

        # Print newline
        li      $v0,4           # print_string syscall code = 4
        la      $a0, lf
        syscall

        # Exit
        li      $v0,10          # exit
        syscall
```

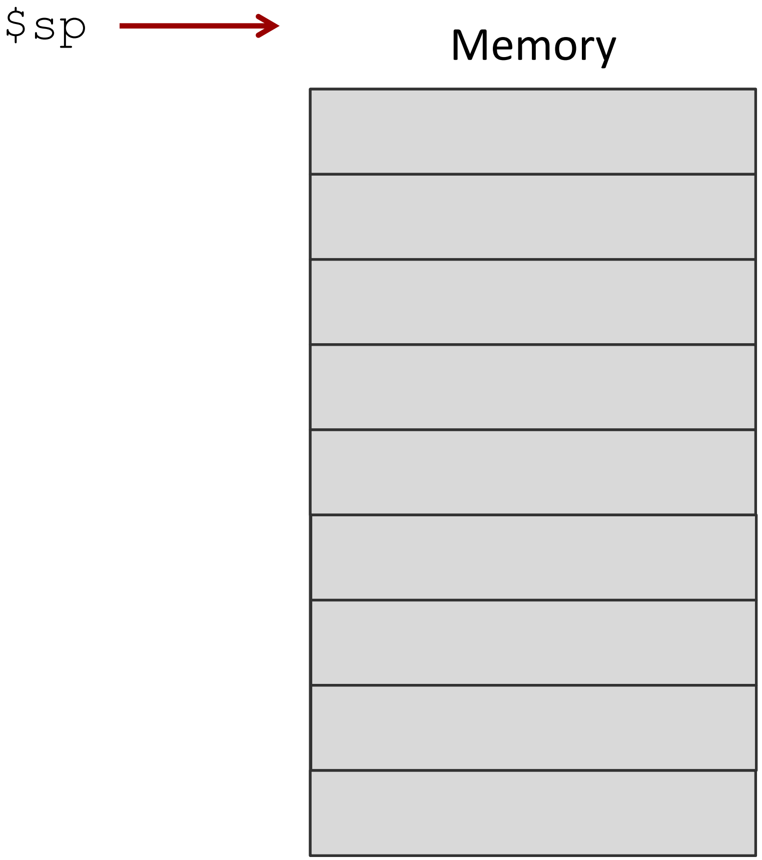
```
# ------------------------------------------------------------------

        # FUNCTION: int fun(int a)
        # Arguments are stored in $a0
        # Return value is stored in $v0
        # Return address is stored in $ra (put there by jal instruction)
        # Typical function operation is:

fun:    # Do the function math
        li $s0, 3
        mul $s1,$s0,$a0# s1 = 3*$a0  (i.e. 3*a)
        addi $s1,$s1,5          # 3*a+5

        # Save the return value in $v0
        move $v0,$s1

        # Return from function
        jr $ra                  # Jump to addr stored in $ra

# ------------------------------------------------------------------

        # Start .data segment (data!)
        .data
x:      .word 5
y:      .word 0
msg1:   .asciiz "y="
lf:     .asciiz"\n"
```

Preserving Registers

- **What if we don’t want to destroy registers used by the calling function?**
  - Perhaps $s0-$s7 are in use with important data…
  - Or $ra holds the return address of a previous call…
- Need to save those registers somewhere while our function runs (like memory!)
- A **stack** is a good structure for this

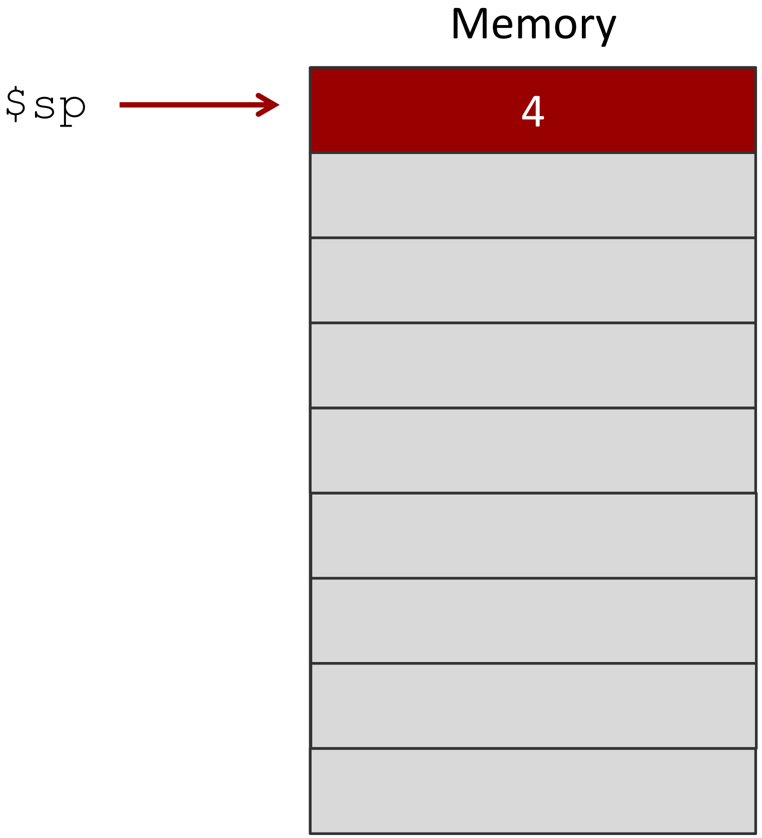
The Stack

- Stack is a data structure stored in memory
- $sp ("Stack Pointer") points to top of stack
  - But stack grows **down** in memory!
- Example
  - Push 4 to stack
  - Push 5 to stack
  - Pop (5 from stack)
  - Pop (4 from stack)



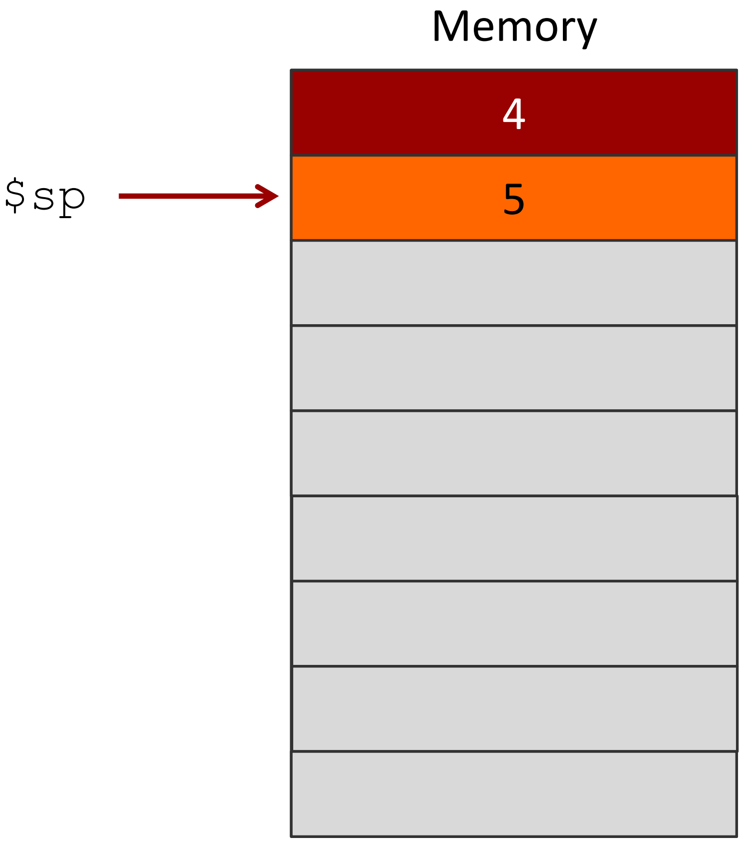
The Stack

- Stack is a data structure stored in memory
- `$sp` ("Stack Pointer") points to top of stack
  - But stack grows **down** in memory!
- Example
  - Push 4 to stack
  - Push 5 to stack
  - Pop (5 from stack)
  - Pop (4 from stack)



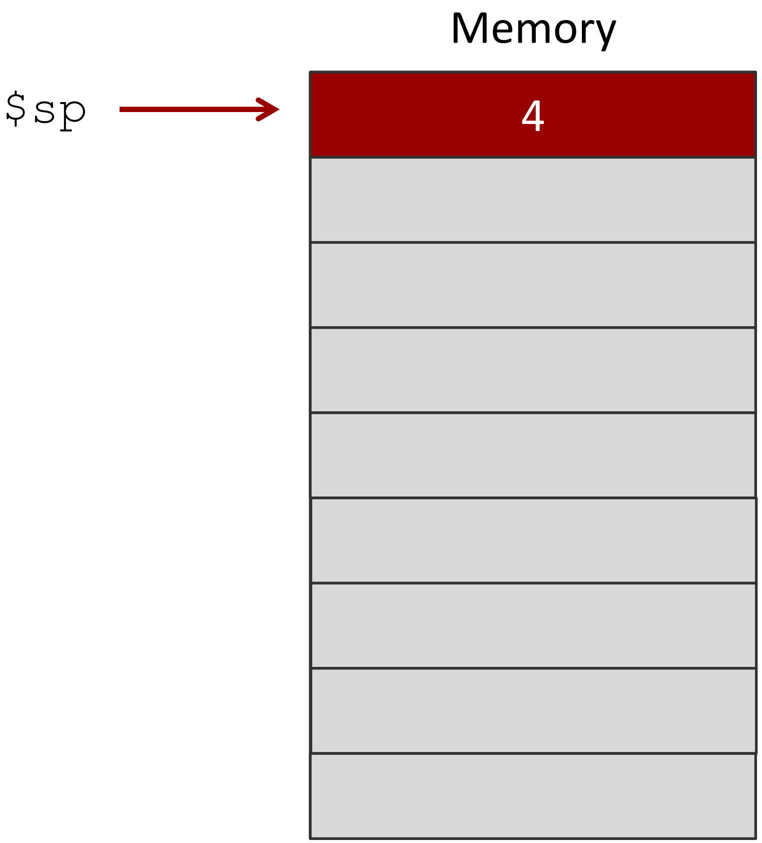
The Stack

- Stack is a data structure stored in memory
- `$sp` ("Stack Pointer") points to top of stack
  - But stack grows **down** in memory!
- Example
  - Push 4 to stack
  - Push 5 to stack
  - Pop (5 from stack)
  - Pop (4 from stack)



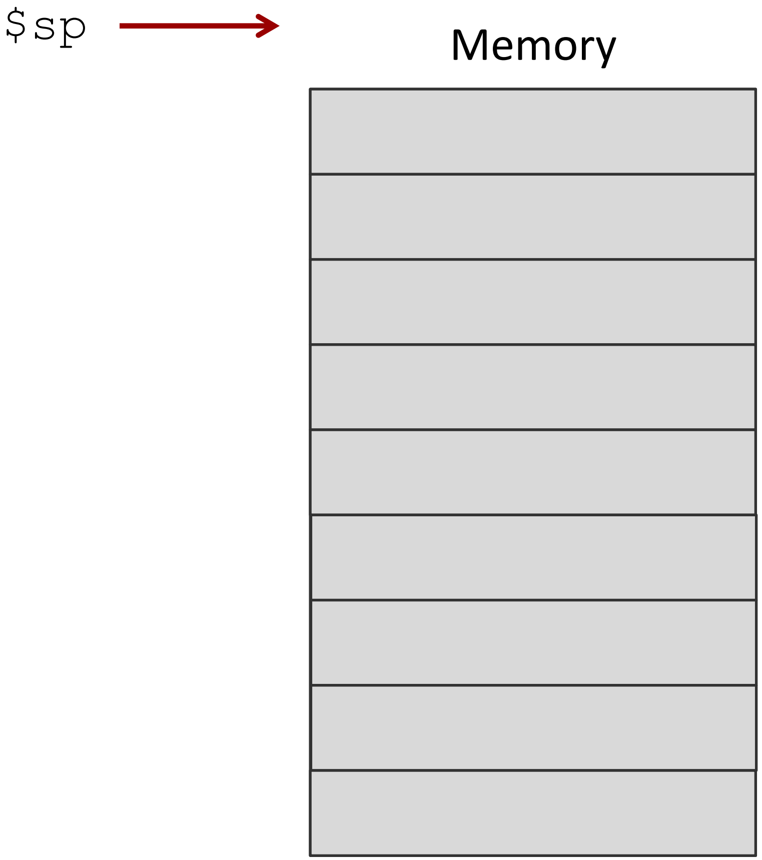
The Stack

- Stack is a data structure stored in memory
- `$sp` ("Stack Pointer") points to top of stack
  - But stack grows **down** in memory!
- Example
  - Push 4 to stack
  - Push 5 to stack
  - Pop (5 from stack)
  - Pop (4 from stack)



The Stack

- Stack is a data structure stored in memory
- `$sp` ("Stack Pointer") points to top of stack
  - But stack grows **down** in memory!
- Example
  - Add 4 to stack
  - Add 5 to stack
  - Pop
  - Pop



Problem 2 : Stack Code

- Using $sp, write the set of commands for pushing and popping the register $s0

```
# Beginning of function

# Push onto stack
addi $sp,$sp,-4     # Adjust stack pointer
sw $s0,0($sp)       # Save $s0

# Function code. Put return values in $v0,$v1

# Restore saved register values from stack
# in opposite order. This is POP'ing from stack
lw $s0,0($sp)       # Restore $s0
addi $sp,$sp,4      # Adjust stack pointer
```

P2

Stack Responsibilities: Caller

- What must a **caller** do with the stack prior to a function call? *(Less common for our programs)*
- Must use the stack if
  - It wants to store temporary registers (`$t0-$t9`) or its argument registers (`$a0-$a3`) onto the stack. This is done before calling another function
  - It wants to pass arguments via stack. *(Not necessary for our programs, we will use the `$a` registers)*
- After function returns, the caller should pop the stack

Stack Responsibilities: Callee

- What must a **callee** do with the stack? ***(required for our programs)***
  1. Push `$s` registers onto the stack, so that it does not overwrite the caller’s data
  2. Push `$ra` onto the stack because a callee may call another function, overwriting the return address.
  3. Do function stuff
  4. Pop `$ra` from the stack
  5. Pop `$s` registers from the stack

Stack Example

- **How would we modify Problem 1 to use a stack?**

```
# Simple routine to demo functions
# NOT using a stack in this example.
# Thus, the function does not preserve values
# of calling function!

# ------------------------------------------------------------------

        .text

        .globl  main
main:
        # Register assignments
        # $s0 = x
        # $s1 = y

        # Initialize registers
        lw      $s0, x          # Reg $s0 = x
        lw      $s1, y          # Reg $s1 = y

        # Call function
        move    $a0, $s0        # Argument 1: x ($s0)
        jal     fun             # Save current PC in $ra, and jump to fun
        move    $s1,$v0         # Return value saved in $v0. This is y ($s1)

        # Print msg1
        li      $v0, 4          # print_string syscall code = 4
        la      $a0, msg1
        syscall

        # Print result (y)
        li      $v0,1           # print_int syscall code = 1
        move    $a0, $s1        # Load integer to print in $a0
        syscall

        # Print newline
        li      $v0,4           # print_string syscall code = 4
        la      $a0, lf
        syscall

        # Exit
        li      $v0,10          # exit
        syscall
```

```
# ------------------------------------------------------------------

        # FUNCTION: int fun(int a)
        # Arguments are stored in $a0
        # Return value is stored in $v0
        # Return address is stored in $ra (put there by jal instruction)
        # Typical function operation is:

fun:    # This function overwrites $s0 and $s1
        # We should save those on the stack
        # This is PUSH’ing onto the stack
        addi $sp,$sp,-4# Adjust stack pointer
        sw $s0,0($sp)           # Save $s0
        addi $sp,$sp,-4# Adjust stack pointer
        sw $s1,0($sp)           # Save $s1

        # Do the function math
        li $s0, 3
        mul $s1,$s0,$a0# s1 = 3*$a0  (i.e. 3*a)
        addi $s1,$s1,5          # 3*a+5

        # Save the return value in $v0
        move $v0,$s1

        # Restore saved register values from stack in opposite order
        # This is POP’ing from stack
        lw $s1,0($sp)           # Restore $s1
        addi $sp,$sp,4          # Adjust stack pointer
        lw $s0,0($sp)           # Restore $s0
        addi $sp,$sp,4          # Adjust stack pointer

        # Return from function
        jr $ra                  # Jump to addr stored in $ra

# ------------------------------------------------------------------

        # Start .data segment (data!)
        .data
x:      .word 5
y:      .word 0
msg1:   .asciiz "y="
lf:      .asciiz"\n"
```

Problem 3 : Write Code

```
int array[] = {2, 3, 4, 5, 6};

int main() {
   int num, position;
   scanf("%d",&num);
   position = search(array, num, 5);
   printf("The position is:
      %d",position);
}

int search(int *array, int num, int size)
{
  int position = -1;
  for(int i=0;i<size;i++)
  if(array[i]==num) {
     position=i;
     break;
  }
  return position;
}
```

Map:

$s0: num
$s1: position
$a0: array address
$a1: num
$a2: size
$v0: return value

P3