

CDT204 -Computer Architecture

Lab Assignment 1

Prepared by Husni Khanfar

Objectives

- Practice MIPS assembly instructions
- Be familiar with PCSPIM
- Converting assembly programs from equivalent C code

Introduction

SPIM is a simulator used to run the programs written for MIPS architecture. In our lab, we use a special version of SPIM called PCSPIM, which is a Windows versions of SPIM. PCSPIM can now be downloaded from your BlackBoard account.

Simulator

MIPS assembly programs are saved as .asm or .s files.

To run your assembly program: run PCSPIM simulator, open your assembly file, then you need to select one of two modes to run your program:

- 1- Run the assembly program once: To do so: select the main menu item: `Simulator`, then select from the appeared pop-up menu: `Go (F5)`.
In this option, all the instructions are executed one from the first instruction labeled by `__start` to the Exit code.
- 2- Run the instructions individually in a step by step manner, hence, the observer can see how each instruction updates the processor and the memory state. To do so: select the main menu item: `Simulator`, then select from the pop-up menu `Single Step (F10)`. You need to press `F10` for executing every instruction

GUI of PCSPIM

The GUI of PCSPIM has four main windows, the first window shows the value of the registers. The second windows displays the instructions of the program with their op-codes. The third window shows the values of the data memory and the last window displays some comments.

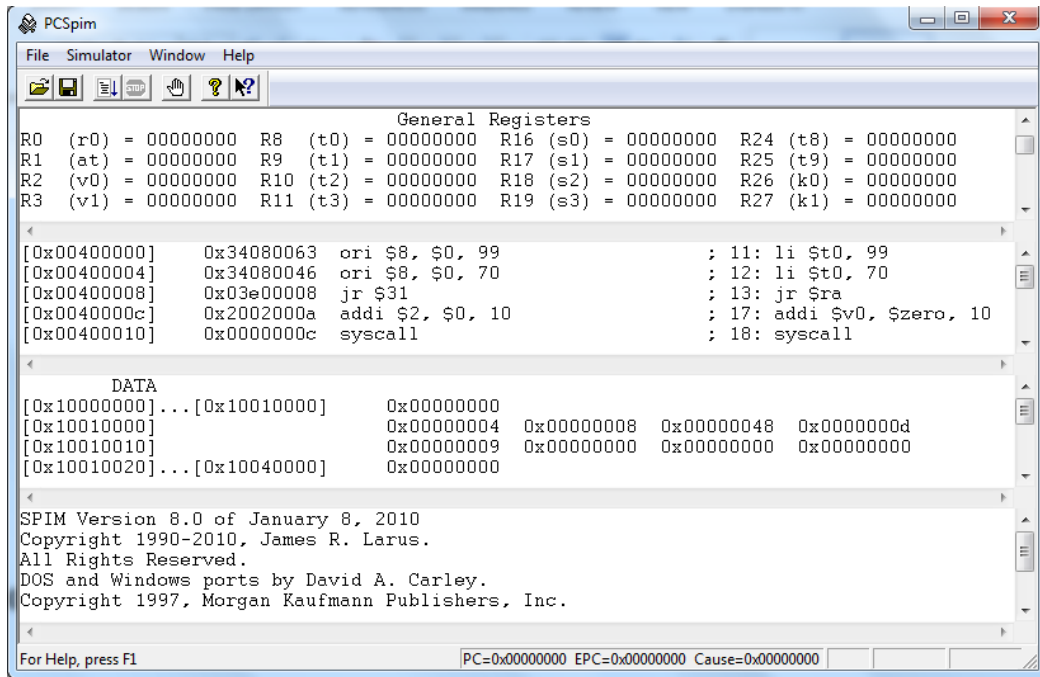


FIGURE 1

The Registers

Most MIPS instructions requires one or more operands. Normally, the data, which the microprocessor processes and manipulates, are stored in an external memory (RAM). Rather than fetching at every execution the values of the operands from the external memory, the microprocessor has a special internal set of locations where the values of its operands are stored temporarily. These internal locations are called: "Registers".

MIPS architecture has 32 registers, which each is 32-bit and they are numbered from 0 to 31:

\$0, \$1, \$2 ... \$30, \$31

The registers in MIPS architecture can be referred also by a special name. The following list shows the corresponding name of some registers in MIPS architecture:

- \$2 - \$3 → \$v0 - \$v1: Result Registers
- \$4 - \$7 → \$a0 - \$a3: Argument Registers
- \$8 - \$15 → \$t0 - \$t7: Temporary Registers
- \$16 - \$23 → \$s0 - \$s7: Saved Registers
- \$24 - \$25 → \$t8 - \$t9: Temporary Registers

General Program Structure

Figure 2 shows the general program structure. You can use this general structure template in writing all of your assembly program. Each section part in figure 2 is illustrated by one of the following sections:

.text and .data

Usually, every assembly program consists mainly of two kind of statements:

- The code segment, which consists of the instructions of the program.
- The data segment, where the data is stored

Both of these two segments must not be merged together. The directives (.text and .data) let the compiler distinguish between the code segment and the data segment respectively.

The Comments

To write a comment in MIPS assembly file, write a sharp sign (#) then everything from the sharp-sign to the end of the line is not compiled.

__start

The execution always starts at the instruction labeled by: __start (two underscores). The label __start must also have been declared as a global directive by: .globl __start.

Exit Code

There is a one way to halt a MIPS program, calling exit system service. If you forget to "call exit" your program, the program does not stop at the last instruction and it goes on through memory. To prevent this, the O.S. must be informed when the program stops and needs removing from the memory.

When MIPS program reaches to the last statement, it must inform the O.S. that it needs to be removed from the memory by making a "call exit". Figure 2 shows the Exit Code, which must be written at the end of each program code.

Basic Instructions: li, move, add, addi

Li: Load Immediate



```
li rd, [immediate value]
```

li instruction is used to load an immediate value into a destination register. As an example in the following line the value 0xBE is stored in \$v0:

```
li $v0, 0xBE
```

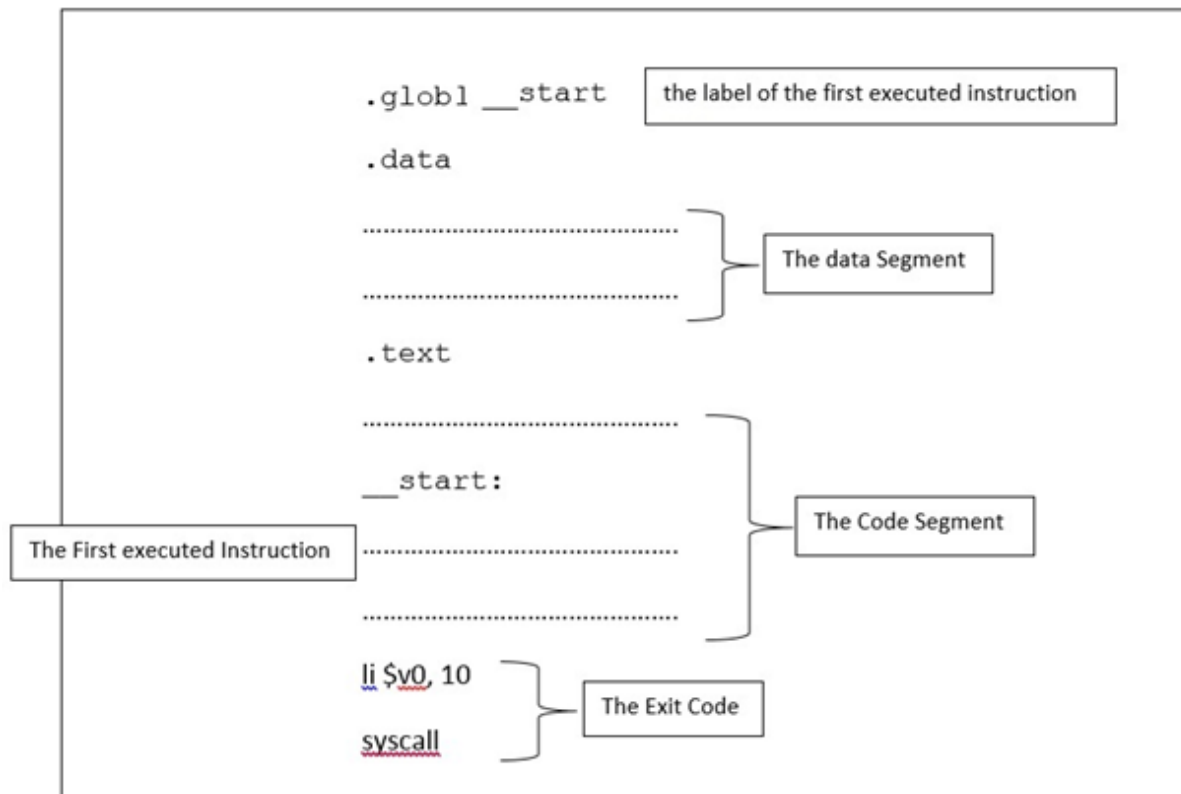


FIGURE 2

MOVE: Move the contents between Registers

`move` instruction has two operands, which both refer to registers. The function of `move` instruction is copying the contents of the second register (operand) to the first register:

`move rd, rs`

For instance, we need to write the following instruction to move the contents of `$t1` to `$v0`:

`move $v0, $t1`

ADD:

`ADD` instruction has three operands and its function is adding the contents of the second register to the third register and then store the result in the third register:

`add rd, rs, rt`

`rd=rs+rt`

ADDI:

`ADDI` works as `ADD`, adding the contents of the second register to the third and stores the results in the first operand. The difference is that the third operand is an immediate value rather than being a register:

`addi $rd, $rs, IMM`

`$rd=$rs+IMM`



Check Your Understanding 1

```
.globl __start
.data
.text
__start:

                                $t1 = .....0..... $t2 = .....0..... $t3= .....0..... $t4 = .....0.....

li $t1, 8

                                $t1 = 8..... $t2 = 0..... $t3= 0..... $t4 = 0.....

li $t2, 9

                                $t1 = 8..... $t2 = 9..... $t3= 0..... $t4 = 0.....

move $t3, $t1

                                $t1 = 8..... $t2 = 9..... $t3= 8..... $t4 = 0.....

add $t4, $t2, $t3

                                $t1 = 8..... $t2 = 9..... $t3= 8..... $t4 = 17.....

addi $t4, $t1, 4

                                $t1 = 8..... $t2 = 9..... $t3= 8..... $t4 = 12.....

add $t2, $t2, $t1

                                $t1 = 8..... $t2 = 17..... $t3= 8..... $t4 = 12.....

li $v0, 10
syscall
```

PROGRAM1.S

- Read Program1.s and calculate the values of the registers \$t1, \$t2, \$t3 and \$t4 after each instruction.
- Run the above program. You can follow the following steps:
 - 1- Create a new folder in the desktop. Nominate it: "Computer Architecture Lab"
 - 2- In "Computer Architecture Lab" folder, create the subfolder: Lab 1
 - 3- Edit the above program by notepad, then save it in lab1 folder by the name: program1.s
Hint: be aware of the extension of the file: it must be .s or .asm and not .txt
 - 4- Run PCSPIM, then open program1.s

- 5- Use F10 to execute individually the instructions. After each instruction, record the value of the registers: \$t1, \$t2, \$t3 and \$t4.
- 6- Finally: Compare your calculations with your observations



Self-Study 1

- 1- Read from any reference the description of the MIPS instructions: "MULT" and "SUB"
- 2- Write the program that does the following :
 - a. $\$t1 = 4, \$t2 = 2, \$t3 = 9$
 - b. $\$t4 = \$t1 * \$t2$
 - c. $\$t5 = \$t3 - \$t1$
- 3- Name your new assembly file: program2.s

addi \$t1, \$zero, 4

addi \$t2, \$zero, 2

addi \$t3, \$zero, 9

mult \$t1, \$t2

mflo \$t4

sub \$t5, \$t3, \$t1

System Calls

Many hardware parts compose and work together to form a Computer System. The O.S. at the middle of this system acts as a conductor in an Orchestra. Although the microprocessor is in the core or the heart of any computer system, but it cannot access or control other hardware peripherals like screen, keyboard, mouse, etc. Usually, the user applications want to communicate with other hardware parts to perform individual tasks (e.g. displaying a message on the screen, reading an integer from the keyboard). Because the operating system (O.S.) is the side that is in charge of controlling the different parts in the system, then the user application should forward its request to the O.S. for performing the intended task. The user application cannot communicate directly with the hardware part.

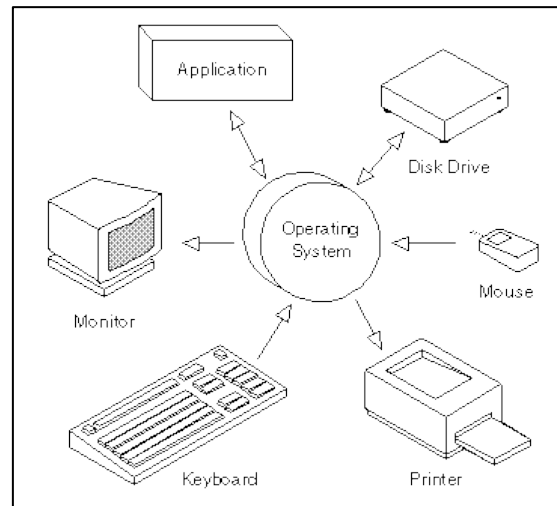


FIGURE 3

In the computer system, there are many hardware parts can serve the applications, each part can respond to many types of requests. If we take the screen as an example, then it can respond to the following types of requests: printing an integer on the screen, printing a string on the screen, move the cursor to new place, read the coordinates of the cursor, change the color of the text, change the color of the background, etc.

The **Service** refers to a type of request served by the O.S.

The instruction: `syscall` is used in the applications to send services to the O.S.

The O.S. usually provides hundreds of Services. Therefore every service has a unique **Call Code**. The call code must be available in `$V0` when `syscall` is executed in order to let the O.S. realizes the intended Service.

It is worth noting that we work with few number of services because we run our MIPS programs in a simulator and not in realistic MIPS platform.

Table 1 illustrates what are the service that we apply in our lab, what the registers associate with each service, and what is the Call Code for each service.

Description	Service	Call Code (\$V0)	Argument	Result
Displaying an integer on screen	<code>print_int</code>	1	<code>\$A0=integer</code>	
Displaying a string on screen	<code>print_string</code>	4	<code>\$A0=string</code>	
Reading an integer from Keyboard	<code>read_int</code>	5		<code>\$V0</code>
Reading a string from the keyboard	<code>read_string</code>	8	<code>\$A0=buffer</code> <code>\$A1=length</code>	
Halt the program	<code>exit</code>	10		

TABLE 1

Example: for using the service `print_int`, to display an integer on the screen, then the following steps should take place:

- 1- store 1 in `$V0`: `li $V0,1`
- 2- Store the value of the integer, which we need to display, in `$A0`. If the intended number is: 7. Then we must write: `li $A0, 7`
- 3- Execute `syscall` instruction.



Check Your Understanding 2

```
.globl __start
.data
.text
__start:

li $v0, 5
syscall
move $t0,$v0

li $v0, 5
syscall
move $t1,$v0

add $t2,$t0,$t1

move $a0,$t2
li $v0, 1
syscall

li $v0, 10
syscall
```

PROGRAM3.S

Program3.s has mainly five parts, read each part, try to understand it and then describe it on the right blanks:

- 1- Set syscall to 5 (read int), call system, receive result integer in \$v0 register and move integer into \$t0 register
.....
- 2- Set syscall to 5 (read int), call system, receive result integer in \$v0 register and move integer into \$t1 register
.....
- 3- Add \$t1 and \$t0 register values and place result into \$t2
.....
- 4- Place \$t2 into \$a0 argument register and set syscall to 1 (print int), call system to print integer in register \$a0
.....
- 5- Load immediate value 10 into \$v0 register (exit system call) and call system to end execution of program
.....

Run Program3.s in PCSPIM and see the results.

Conditional Branches

<code>beq \$t0,\$t1,target</code>	<code># branch to target if \$t0 = \$t1</code>
<code>blt \$t0,\$t1,target</code>	<code># branch to target if \$t0 < \$t1</code>
<code>ble \$t0,\$t1,target</code>	<code># branch to target if \$t0 <= \$t1</code>
<code>bgt \$t0,\$t1,target</code>	<code># branch to target if \$t0 > \$t1</code>
<code>bge \$t0,\$t1,target</code>	<code># branch to target if \$t0 >= \$t1</code>
<code>bne \$t0,\$t1,target</code>	<code># branch to target if \$t0 <> \$t1</code>

Example:

```
li $t5, 8
li $t6, 2
blt $t6,$t5, L1
addi $t1, 4
l1: move $t5,$t7
```

`blt` jumps to `L1` because its first operand (`$t6`) is less than its second operand (`$t5`).

When is `addi $t1, 4` executed? When `$t6` is equal or greater than `$t5`



Check your understanding 3

In `Program4.s`, do the followings:

- 1- Track the instructions
- 2- Fill the blanks by the values of `$t1`, `$t3` and `$t6` respectively.
- 3- Write the equivalent C code in the right of each block
- 4- Implement `Program4.s`. Run your instructions individually by `F10`, at every "nop" instruction stop and check whether your calculations agree well with your observations.

```
.globl __start
```

```
.data
```

```
.text
```

```
__start:
```

```
li $t1, 8
```

```
li $t2, 9
```

```
blt $t1,$t2, L1
```

```
addi $t1, $t1, 2
```

```
L1: addi $t1, $t1, 1
```

```
nop
```

\$t1=8 ;

\$t2 = 9

if (\$t1 < \$t2) goto L1;

\$t1=\$t1+2;

L1: \$t1=\$t1+1;

~~\$t1~~ = ⁹

```
li $t7, 2
```

```
li $t3, 2
```

```
bne $t7,$t3, L2
```

```
addi $t3, $t3, 2
```

```
L2: addi $t3, $t3, 5
```

```
nop
```

\$t7 = 2;

\$t3 = 2;

if(\$t7 != \$t3) goto L2;

\$t3 = \$t3 + 2;

L2: \$t3 = \$t3 + 5;

~~\$t3~~ = ⁹

```
li $t6, 4
```

```
li $t5, 1
```

```
bge $t6,$t5, L3
```

```
add $t6, $t6, $t5
```

```
addi $t6, $t6, 3
```

```
L3: addi $t6, $t6, 5
```

```
nop
```

\$t6 = 4;

\$t5 = 1;

if(\$t6 >= \$t5) goto L3;

\$t6 = \$t6 + \$t5

\$t6 = \$t6 + 3

L3: \$t6 = \$t6 + 5

~~\$t6~~ = ¹³

PROGRAM4.S



Check your Understanding 4

Write a simple program that does the following steps:

- 1- read the first input
- 2- read the second input
- 3- If the first input is greater than the second input, store 1 in \$t1
- 4- If the first input is less than the second input, store 2 in \$t1
- 5- If they are equal, store 3 in \$t1
- 6- Display the value of \$t1

```
.globl __start
```

```
.data
```

```
.text
```

```
__start:
```

```
li $v0, 5
```

```
syscall
```

```
move $s0, $v0
```

```
li $v0, 5
```

```
syscall
```

```
move $s1, $v0
```

```
ble $s0, $s1, l1
```

```
li $t1, 1
```

```
l1:
```

```
beq $s0, $s1, l2
```

```
li $t1, 2
```

```
l2:
```

```
li $t1, 3
```

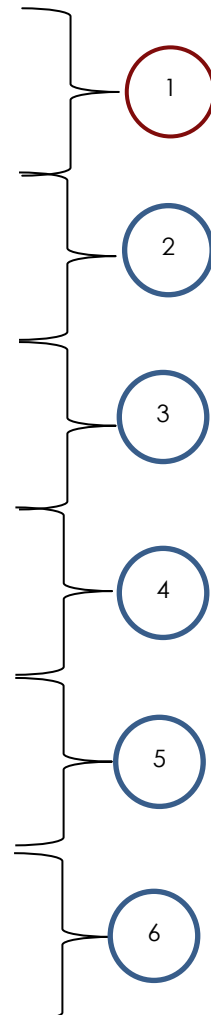
```
li $v0, 1
```

```
move $a0, $t1
```

```
syscall
```

```
li $v0, 10
```

```
syscall
```



PROGRAM5.S

- 7- After writing your program, check the correctness of your code by implementing it.

PC and JUMP Instructions

When the microprocessor is executing any instruction, PC register stores the address of the next instruction (The one which is immediately after the instruction that is being executed). If the value of the PC register is changed, then the flow of the executions is changed. Jump instructions change the flow of the executions by changing the value of PC register.

Labels

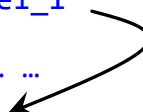
Jump instructions load a new value (address) into PC register. Since the programmer doesn't know the address of each instruction during writing his code, then rather than working with addresses, the programmer describes the address of each instruction by its label.

The name of the label is a user-defined. Each label is followed by a colon (:). The label represents the address of its immediately after instruction.

J : Jump

J instruction jumps the flow of the execution to the instruction labelled by "Label_1".

```
J Label_1
... ..
Label_1:
li $t0, 1
move $t1, $v0
```

A curved arrow originates from the text 'J Label_1' and points to the 'Label_1:' line in the code block below.

Label_1 represents the address of the instructions:

```
li $t0,1
```



Check you Understanding 6

Write a program that does the following: reads an integer number from the keyboard, if the value of this input is more than 10 and less than 50 then display on the screen: "1". Otherwise, do not display anything

Hint: after reading the value, if it is less than 10 or greater than 50, then go to the label END_1.

```
.globl __start
```

```
.data
```

```
.text
```

```
__start:
```

```
li $v0, 5
```

```
syscall
```

```
addi $t0, $zero, 10
```

```
addi $t1, $zero, 50
```

```
blt $v0, $t0, END_1
```

```
bge $v0, $t1, END_1
```

```
addi $t2, $zero, 1
```

```
move $a0, $t2
```

```
li $v0, 1
```

```
syscall
```

```
END_1: li $v0, 10
```

```
syscall
```

Read the Integer

If the input is less than 10,
then Jump to END

If the input is greater than
50, then Jump to END

Display 1 on the screen

PROGRAM6.S

JR : Jump Register

JR jumps to an address stored in the register Rsc.

JR Rsc

.....

JAR: Jump and Link

JAR instruction stores the address of the next instruction (the one immediately after the jump) in the return address (\$ra; \$31) register. This allows a subroutine to return to the main body routine after completion.

JAR Label

Procedures Called by JAR and JR

In C language, when a function is called, there are two important steps:

- 1- In the first step, the flow of the execution jumps to the first statement in the function rather than to the next immediate statement.. This is shown in figure 4 by the green line.
- 2- In the second step, which is shown by the blue line, a jump occurs from the last statement in the function CircleArea to the immediate next statement of the CircleArea callee.

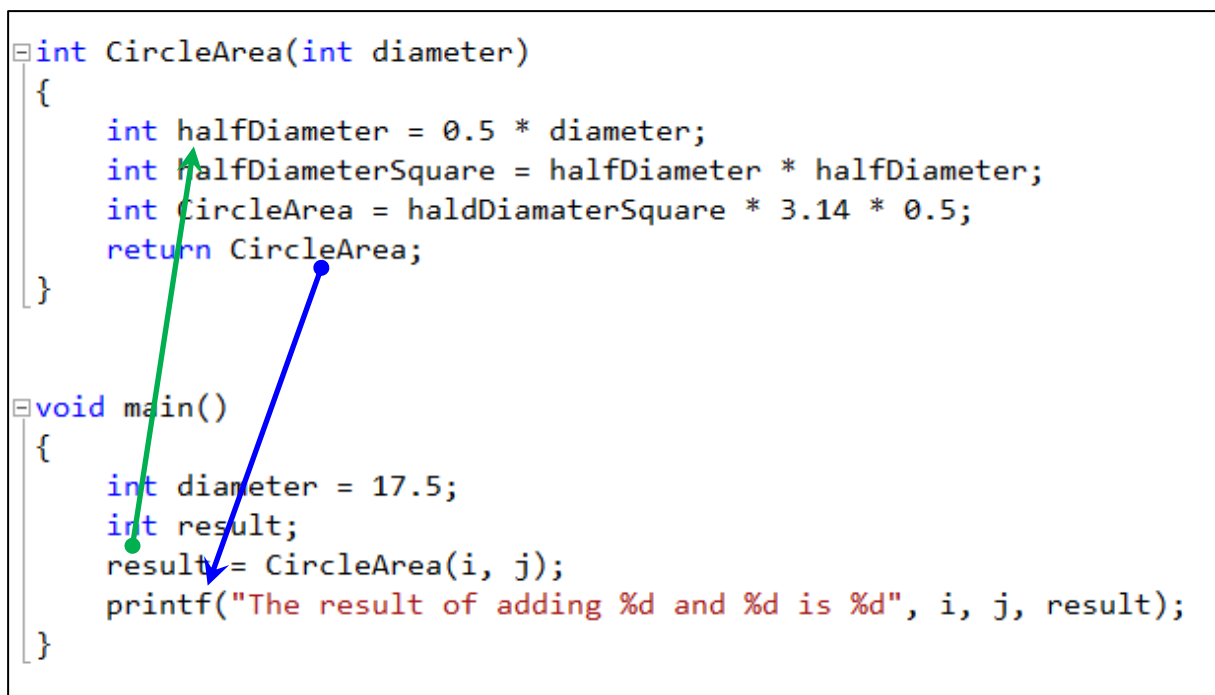


FIGURE 4

JAL and JR cooperate together to build a procedure-style structure. JAL saves the address of its immediate next instruction in \$ra register and load the address of its label in PC register. JR works as the blue line. It jumps to the address stored in \$ra, so it returns to the immediate next instruction of JAL instruction.

```

#This procedure adds the contents of the registers:
#$t1, $t2, $t3 and $t4
ADD4Registers:
add $t1, $t1, $t2
add $t1, $t1, $t3
add $t1, $t1, $t4
jr $ra
__start
.....
.....
JAL ADD4Registers
nop

```

FIGURE 5



Check your Understanding 6

Assume that you need to read 3 integers from the keyboard and store these values in the registers: \$t0, \$t2 & \$t4 respectively. Write a program achieve this mission.

Hint: Rather than writing read_int service three times, use read_int service to write a procedure reading integers from the keyboard, then use this procedure in reading the inputs from the keyboard.

Implement Program 7 and check whether your code works properly or not.

```

.globl __start

.data

.text

Read_Input:
    li $v0, 5
    .....
    syscall
    .....
    jr $ra
    .....

__start:
    jal Read_Input
    .....
    move $t0, $v0
    .....

    jal Read_input
    .....
    move $t2, $v0
    .....

    jal Read_Input
    .....
    move $t4, $v0
    .....

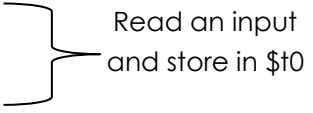
nop      #check manually the values of $t0,$t2 & $t4

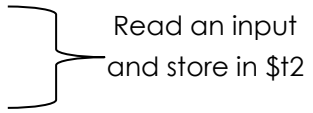
END:

li $v0, 10

syscall

```





Program7.s

Data Segment

The data are declared by many types such as `.byte` and `.word`.

The format for data declarations:

name: storage_type value(s)

- create storage for variable of specified type with given name and specified value
- value(s) usually gives initial value(s);
- Note: labels always followed by colon (:)

Load Instructions

The load instructions (`lw` & `lb`) copy the contents of a memory location into one of the registers.

`lw` copies a word (4 bytes) at a source RAM location to a destination register.

`lw register_destination, RAM_source`

`lb` copies byte at source RAM location to low-order byte of destination register

`lb register_destination, RAM_source`

Store Instructions

`sw` stores a word in source register into a RAM destination:

`sw register_source, RAM_destination`

`sb` stores byte (low-order) in source register into RAM destination:

`sb register_source, RAM_destination`



Check your understanding 7

Assume that we have a list of purchases (banana: 70 SEK, Apple: 50 SEK, Orange: 40 SEK, Bed: 1000 SEK, Chair: 600 SEK, Desk: 1000 SEK). Write a program that stores each item with its price in the data memory. In addition, create two word memory locations: Total Fruit and Total Furniture.

The function of this program is to sum the prices of the fruit items in "Total Fruit" and sum the prices of the furniture items in "Total Furniture".

Hint: Store the prices which are less than 255 by `.byte` type and store the prices which are more than 255 in a `.word` type.

Program8.s is the name of this file. Fill the blanks in the next page.

Arrays

Because we have limited number of registers (~32), it is impractical to use the registers for long-term storage of the array data. Sometimes, the programmer needs to work with long arrays, e.g. array of names, array of purchases, array of aggregated data, etc. The arrays store a lot of data. Thus, the arrays are stored in the Data Segment.

Every array has three fundamental operations. Table 2 compares how these three operations are done in C language and by MIPS instructions.

	C Language	MIPS Assembly
Creating an Array	<code>int array[4]</code>	<code>array: .word 200,9,20,5</code>
Getting the data from an array cell	<code>x = array[2]</code>	<code>lw \$t1, array+2</code>
Storing data into an array cell	<code>array[2]=x</code>	<code>sw \$t1, array+2</code>

TABLE 2



Check your Understanding 9

In the previous example, store the six purchases in a one array "purchases", then display on the screen "Total Purchases".

Program9.s helps writing your code, fill the blanks and then implement the source code to see the results in "Total Purchases".

```

.globl __start
.data
Banana: .byte 70
Apple: .byte 50
.....
Orange: .byte 40
.....
Fruits: .byte 0
.....

Bed: .byte 1000
.....
Chair: .byte 600
.....
Desk: .byte 1000
.....
Furniture: .byte 0
.....

.text
__start:
lb $t1, Banana
.....
lb $t2, Apple
.....
lb $t3, Orange
.....
add $t0, $t1, $t2
.....
add $t0, $t0, $t3
.....
sb $t0, Fruits
.....

lb $t1, Bed
.....
lb $t2, Desk
.....
lb $t3, Chair
.....
add $t0, $t1, $t2
.....
add $t0, $t0, $t3
.....
sb $t0, Furniture
.....

#Check the value of Total Fruit and #Total Furniture
nop
li $v0, 10
syscall

```

```

#The price of the Banana
#The price of the Apple
#The price of the Orange
#Total Price of Fruits

#The price of the Bed
#The price of the Chair
#The price of the Desk
#Total Price of Furniture

#Banana → $t1
#Apple → $t2
#Orange → $t3
#$t0 = $t1 + $t2
#$t0 = $t0 + $t3
#$t0 → Total Fruit

#Bed → $t1
#Desk → $t2
#Chair → $t3
#$t0 = $t1 + $t2
#$t0 = $t0 + $t3
#$t0 → Total Furniture

```

PROGRAM8.S

```

.globl __start

.data

#define the array: purchases
    array .word 70, 50, 40, 1000, 600, 1000
.....

.text

__start:  la
    lw $t1, array                #$t1 = purchases[0]
    lw $t2, array+1              #$t2 = purchases[1]
    lw $t3, array+2              #$t3 = purchases[2]
    lw $t4, array+3              #$t4 = purchases[3]
    lw $t5, array+4              #$t5 = purchases[4]
    lw $t6, array+5              #$t6 = purchases[5]
    add $t0, $t1, $t2            #$t0 = $t1 + $t2
    add $t0, $t0, $t3            #$t0 = $t0 + $t3
    add $t0, $t0, $t4            #$t0 = $t0 + $t4
    add $t0, $t0, $t5            #$t0 = $t0 + $t5
    add $t0, $t0, $t6            #$t0 = $t0 + $t6

#Display the Total Purchase on the Screen
    li $v0, 1
    move $a0, $t0
    syscall

li $v0, 10

syscall

```

PROGRAM9.S

Tasks

Task 1: Conditional jump

Read two integers from the keyboard, then display the greater number on the screen.

Hints: write and use two procedures Read_Int and Print_In

The name of the file: Program10.s

Task 2: Loops and Arrays

Implement a loop that finds the largest value in an array called purchases, which is allocated in the data segment and consisting of 6 elements. After finding the largest value, print it on the screen.

The equivalent C code is:

```
int max
max = purchases[0]
for(int i=1; i < 6 ; i++)
    if(purchases[i] > max) max = purchases[i];
print_int(max);
```

The name of the file: Program11.s

Task 3: Fill the array from the Keyboard

In the second task, the array was already allocated on the data memory. Now we need to improve Task 2 to be more practical. Rather than entering the values of the purchases in the compile time, we need to enter all the purchases in the run time (when the program is executed).

The equivalent C code would look something like this:

```
//read the inputs from the keyboard
int i, max, purchases[6];
for (i = 0; i <6; i++)
    purchases[i] = read_int();

//Finding the largest value in the array
max = arr [0];
for (i = 1; i <6; i++)
    if (purchases[i]> max)
        max = arr [i];

//Print the largest number
print_int(max);
```

The name of the file: Program12.s