**Department of Electrical Engineering**

|  |  |
| --- | --- |
| **Faculty Member:Taufeeq-Ur-Rehman** | **Dated: 22 March 2018** |
|  |  |
| **Course/Section:BESE/7B** | **Semester: 4TH** |
|  |  |

**Computer Organization and**

**Achitecture (EE321)**

**Lab #\_6\_\_**

**Group#13**

**CPU Simulator**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Reg. no.** | **Report Marks / 10** | **Viva Marks / 5** | **Total/15** |
| **Abdul Manan** | **108480** |  |  |  |
| **Sheheryar Iqbal** | **175826** |  |  |  |
| **Sohaib Azhar** | **129831** |  |  |  |
|  |  |  |  |  |

**Lab # CPU Simulator**

**Objectives:** At the end of this lab you should be able to:

* Create instructions to move data to CPU registers
* Create instructions to compare values in CPU registers
* Push data to the stack, pop data from the stack
* Create instructions to jump to address locations
* Add values in CPU registers
* Demonstrate compare instruction’s effect on the status register flags
* Use direct and indirect addressing modes
* Create iterative loop of instructions
* Display text on console using IO instruction
* Create a sub-routine
* Call the sub-routine and return from the sub-routine

**Lab Instructions**

* This lab activity comprises two parts, namely Lab tasks, and Post-Lab Viva session.
* Each group to upload completed lab report on LMS for grading.
* The students will start lab task and demonstrate each lab task separately for step-wise evaluation
* There are related questions in this activity give complete answers. Also provide complete code and results.

Introduction

The CPU simulator can run programs manually created. This requires four stages:

1) Create a CPU program

2) Enter CPU instructions in the program

3) Run the program

4) Observe and control simulations

Below is the description of these four main stages:

**1. Creating a CPU program that will contain the instructions**

Enter the program’s name in the **Program Name** text box, e.g. MyProgram. Enter a number in the **Base Address** text box (suggest you use 0 in this case). Next click the **ADD** button. The name of the program will appear in the PROGRAM LIST frame (see Fig. 2).

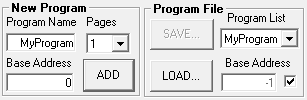


Fig. 1 – New program frame

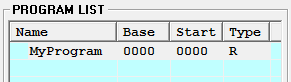


Fig 2 – Program list frame

**2. Adding CPU instructions in the program**



In the **PROGRAM INSTRUCTIONS** frame the **ADD NEW…** button is enabled. Click this button to view the CPU instructions you can select for your program created above (see Fig. 4 below).

Fig. 3 – Program instructions frame

**Selecting CPU instructions to manually enter in the program:**

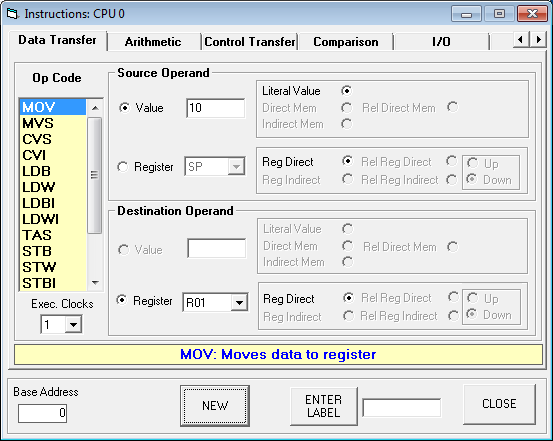
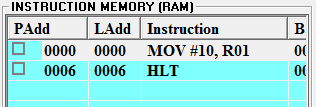


Fig. 4 – Selecting CPU instructions

In the **Instructions** window shown above, you can select any of the available CPU instructions for your program. Instructions that are related are categorized into different groups. You select a group by clicking on a tab. Select the desired instruction from the list under the **Op Code**. If the selected instruction requires operand(s) then the available types will be enabled under the **Source Operand** and the **Destination Operand** frames. Next, click the **NEW** button to add the instruction. You can add multiple instructions without closing this window.

The instruction will appear in CPU program memory area as shown in Fig. 5 below. The CPU simulator requires that the last instruction in the program should be the **HLT** instruction (this instruction stops the simulation from running). This is included in Fig. 5. The program is now ready to run.

**CPU instructions in program:**



Each entry in the **INSTRUCTION MEMORY** view includes the following information:

**PAdd** (Physical Address), **LAdd** (Logical Address) and the instruction. Other information is also available but this is not relevant at this stage.

Fig. 5 – CPU program memory

**Editing the program instructions:**

Once the program instructions are entered they can be edited. To do this, select the desired instruction and use one of the editing functions (**EDIT**, **DELETE**, **MOVE UP**, **MOVE DOWN**) shown in Fig. 6 to edit the selected instruction. Use the **INSERT ABOVE…** and the **INSERT BELOW…** buttons to insert a new instruction above or below the selected instruction.

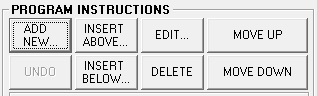
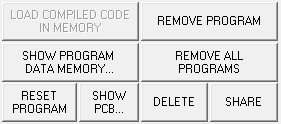


Fig. 6 – Program edit functions

**Removing the program:**



A CPU program can be removed by clicking the **REMOVE PROGRAM** button shown in Fig. 7. Once the program is removed it is no longer available and is lost. However you can save it so that you can re-load it at a later time (see Fig. 12).

Fig. 7 – Program removal

**3. Running the program**

A CPU program can run in two different ways: 1) instruction by instruction, 2) automatically in one go. To run a desired instruction by itself, first select it as in Fig. 5 and double-click it. Alternatively, you can click the **STEP** button shown in Fig. 8 (make sure the **by inst.** option is selected). Use the **STOP** button to stop the running program. Use the slider control to speed up or slow down the running program.

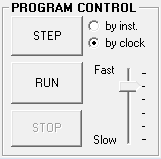


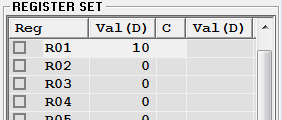
Fig. 8 – Program control

**4. Observing and controlling the simulations**

An instruction which writes or reads a register accesses the **REGISTER SET** frame and the accessed register is highlighted. This frame shows the registers and their values. Click on the **Val** (value) column to change the values from decimal (**D**) to hex (**H**) format and vice versa.

**Note**: To manually alter a register’s value, first select it then enter the new value in the **Reg Value** field and click on the **CHANGE** button (not shown in Fig. 9).

**Observing/altering the CPU registers:**

  
 Fig. 9 – Register set view

**Observing/altering the program data:**

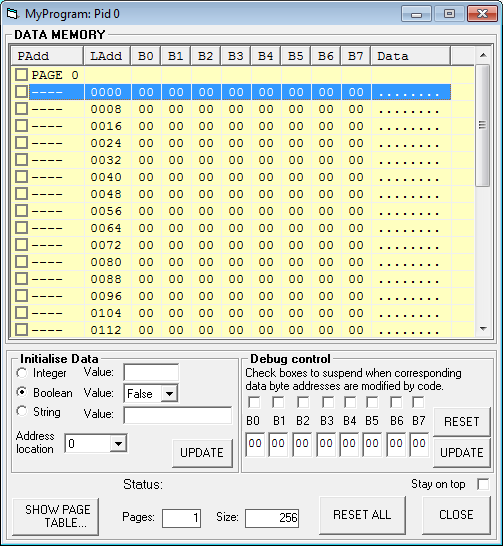


Fig. 10 – Data memory page

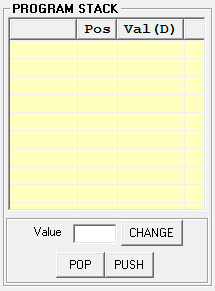
The CPU instructions that access that part of the memory containing data can write or read the data accessed. This information is available in the memory pages window shown in Fig. 10 above. You can display this window by clicking the **SHOW PROGRAM DATA MEMORY…** button shown in Fig. 7 above. In this window you can also edit the contents of the data.

The **LAdd** column shows the starting address of each line in the display. Each line of the display represents 8 bytes of information, so the **LAdd** numbers are incremented by 8 for each line down the display. Columns **B0** through to **B7** contain bytes 0 to 7. The **Data** column shows the displayable characters corresponding to the 8 bytes (hence 8 dots). Those bytes that correspond to non-displayable characters are shown as dots. The data bytes are displayed in hex format only.

To change the values of any bytes, first select the line(s) containing the bytes. Then use the information in the **Initialize Data** frame to modify the values of the bytes in the selected line(s) as **Integer**, **Boolean** or **String** formats. You need to click the **UPDATE** button to make the change.

Check the **Stay on top** check box to make sure the window always stays on top of other windows while still allowing access to the windows below it.

**Observing/altering the program stack:**



Programs make use of the **PROGRAM STACK** for temporarily storing important information such as subroutine return addresses and subroutine parameters as well as other relevant information. There are instructions that can push (**PSH**) data on top of this stack and that can pop (**POP**) data from top of the stack to a register.

You can manually push and pop data by clicking the **PUSH** and **POP** buttons. You can also modify a stack entry by selecting it, entering the new value in the Value text box and clicking the **CHANGE** button.

Fig. 11 – Program stack frame

**Special CPU registers view**

This view shows the set of CPU registers, which have pre‐defined specialist functions:

**PC:** Program Counter contains the address of the next instruction to be executed.

**IR:** Instruction Register contains the instruction currently being executed.

**SR:** Status Register contains information pertaining to the result of the last executed instruction.

**SP:** Stack Pointer register points to the value maintained at the top of the program stack

**BR:** Base Register contains current base address.

**MAR:** Memory Address Register contains the memory address currently being accessed.

**Status bits: OV: Overflow; Z: Zero; N: Negative**

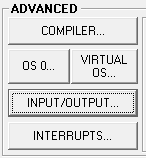
**Saving and loading the CPU programs:**



To save a program select it from the pull down list and click the **SAVE…** button. To load a saved program click the **LOAD…** button.

Fig. 12 – Saving and loading programs

**Observing the displayed information:**



Programs are able to display information on and accept data from the simulated console. The **OUT** instruction is used to display information and the **IN** instruction is used to accept input from the console. To show the console click the **INPUT/OUTPUT…** button shown in Fig. 13. The console window is shown in Fig. 14 below.

Fig. 13 – Console button

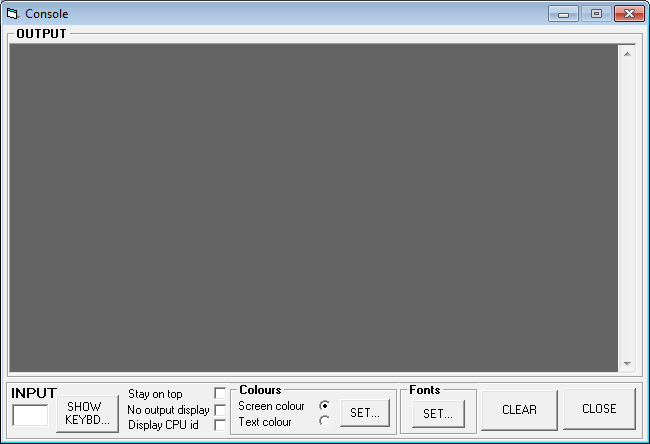


Fig. 14 – Console window

Check the **Stay on top** check box to make sure the window always stays on top of other windows while still allowing access to the windows below it. Click on the **SHOW KEYBD…** to show a small virtual keyboard used to input data (see Fig. 15 below).

Check the **Lower Case** check box to input lower case characters. This keyboard is a subset of a standard keyboard. You can also supply input to a program by typing in the **INPUT** text box shown in Fig. 14 above.



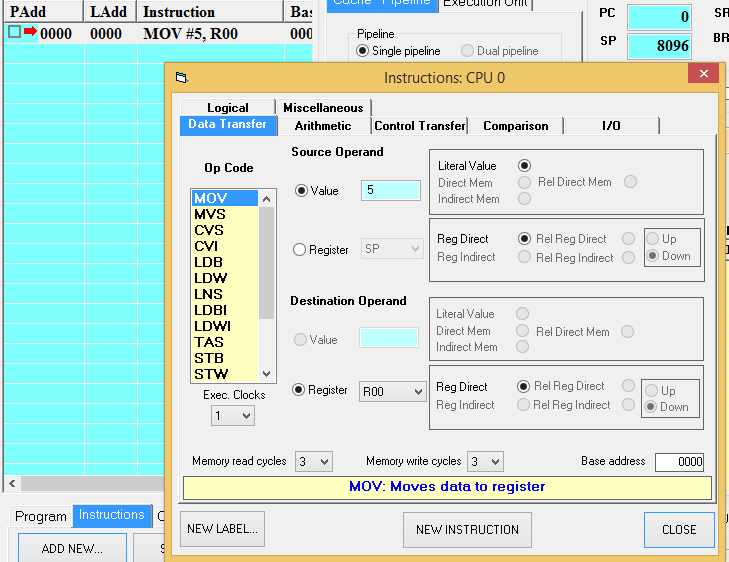
Fig. 15 – Virtual keyboard

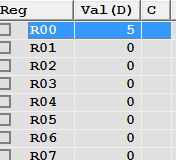
Lab Exercise 1

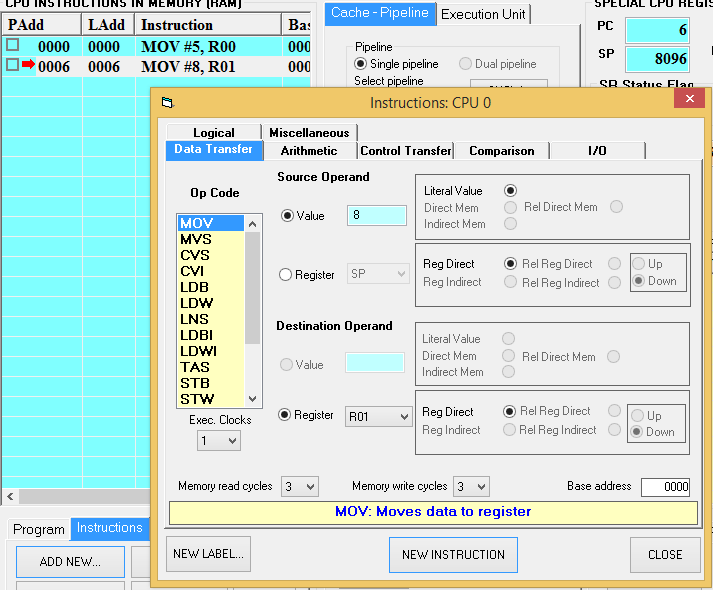
You now need to enter instructions into the CPU simulator. You do this by first creating a new program and then clicking on the ADD NEW… button in the Edit Program tab. This will display the Instructions: CPU0 window. Use this window to enter the instructions. You’ll find a list of useful CPU simulator instructions and examples at the end of the above document. You can also edit the instructions by using the edit commands available in the Edit Program tab.

**Please record your work in the boxes as you go along.**

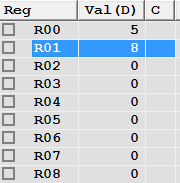
1. Create an instruction that moves number 5 to register R00.



1. Execute the above instruction by simply double clicking on it in the **CPU INSTRUCTIONS IN MEMORY (RAM)** view.  
   
2. Create an instruction that moves number 8 to register R01.

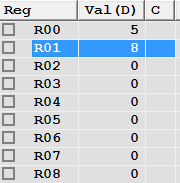


1. Execute it.

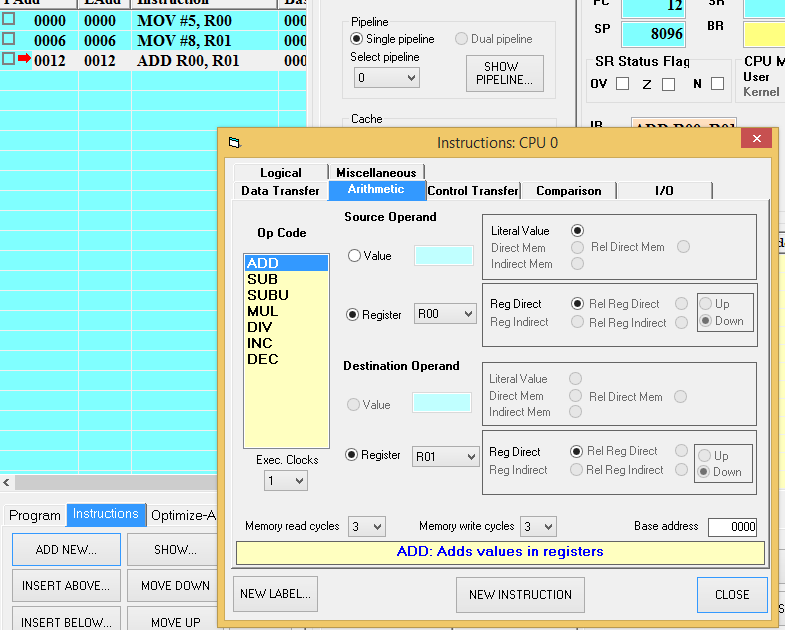


1. Observe the contents of R00 and R01 in the **Register Set** view.

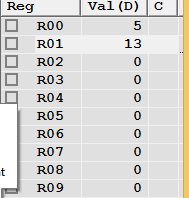
R00 contain value 5 in decimal and 8 in R01



1. Create an instruction that adds the contents of R00 and R01.

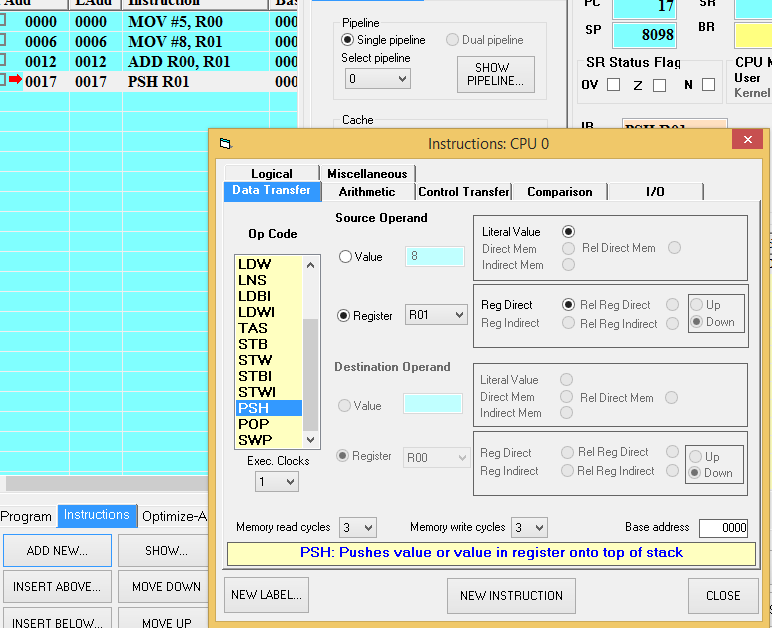


1. Execute it.
2. Write down which register the result is stored in.

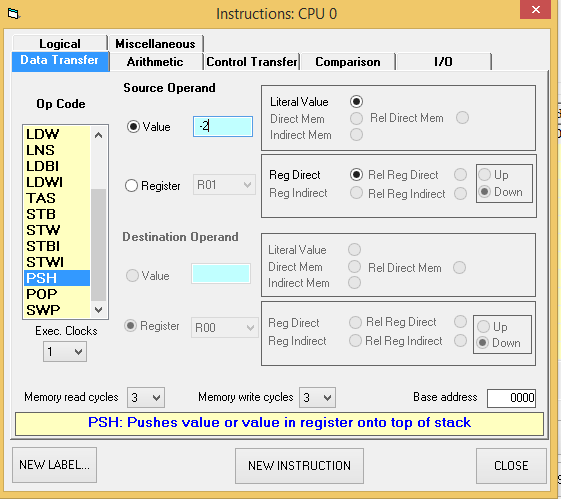


The result is stored in R01

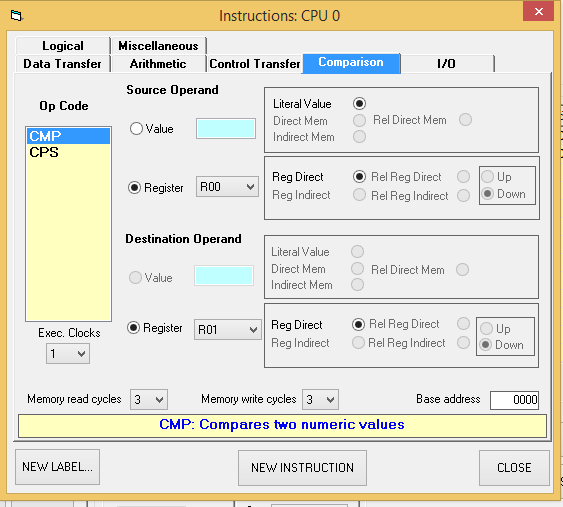
1. Create an instruction that pushes the above result to the top of the program stack, and then execute it. Observe the result in **PROGRAM STACK (RAM)** view.



1. Create an instruction to push number -2 on top of the stack and execute it.

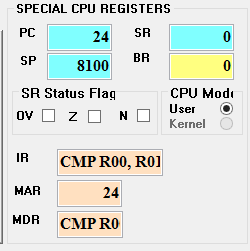


1. Create an instruction to compare the values in registers R00 and R01.

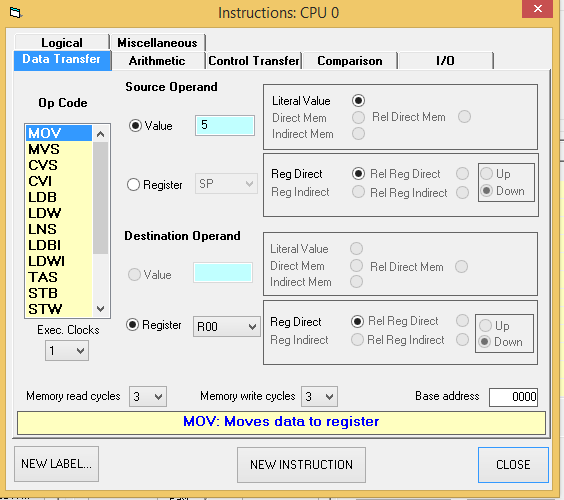


1. Record the status (i.e. set or reset) of the **Z**/**N** flags of the status register. If the box is checked then means set otherwise means not set.

The box is unchecked its means not set.

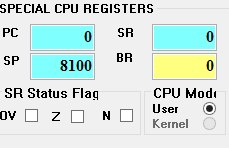


1. Create an instruction to unconditionally jump to the first instruction.

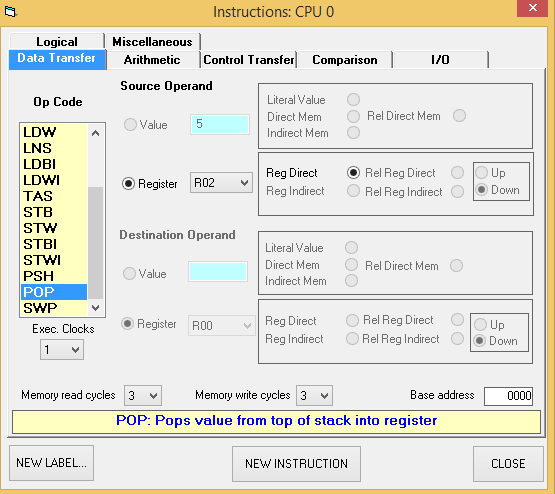


1. Observe the value in the PC register. What address is it pointing to? Explain.

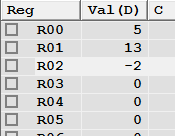
PC register is pointing to 0000



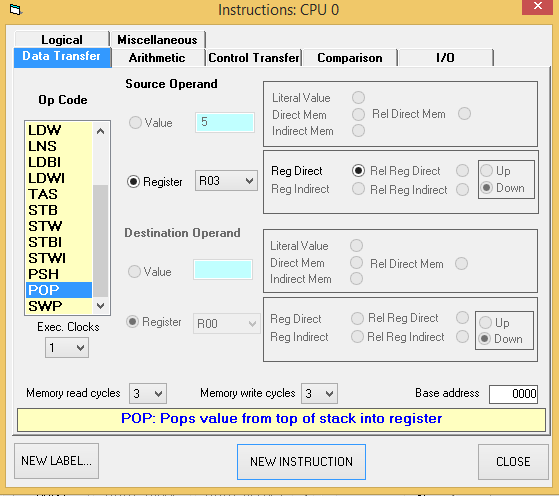
1. Create an instruction to pop the value on top of the program stack into register R02.



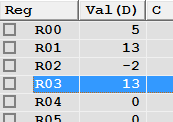
1. Execute it.



1. Create an instruction to pop the value on top of the program stack into register R03.

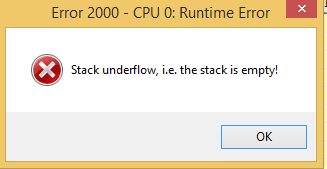


1. Execute it.

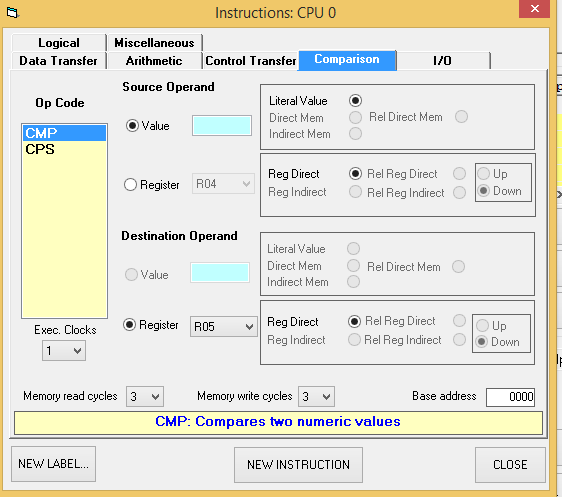


1. Execute the last instruction again. What happened? Briefly explain.

This error will be popped as stack is empty.



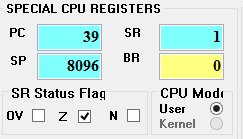
1. Create a compare instruction that compares values in registers R04 and R05.



1. Manually insert two equal values in registers R04 and R05.



1. Execute the compare instruction in step 22 above.



1. Which of the status flags **Z/N** is set? Why?

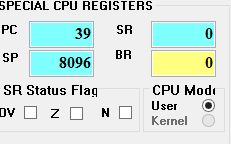
The z flag is set because there is no difference in value of RO4 and R05 as Z flag shows zero

1. Manually insert a value in register R05 greater than that in register R04.



1. Execute the compare instruction in step 22 above.

1. Which of the status flags **Z/N** is set? Why?

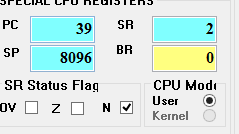


No one is set because the difference is not zero or negative

1. Manually insert a value in register R04 greater than that in register R05.

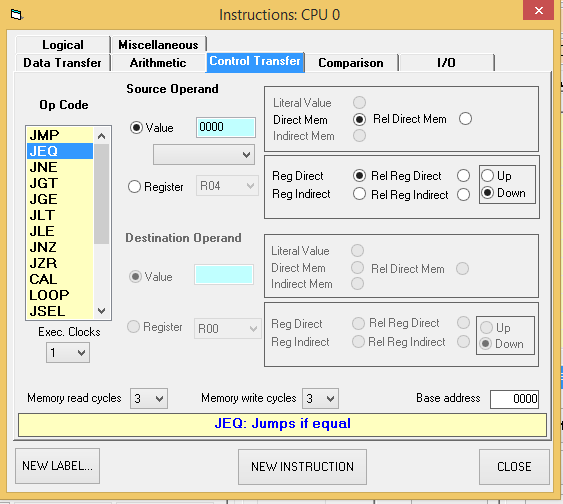


1. Execute the compare instruction in step 22 above.
2. Which of the status flags **Z/N** is set? Why?



As there difference is negative so N flag is checked

1. Create an instruction that will conditionally jump to the first instruction if the values in registers R04 and R05 are equal (**Note**: You will need to execute the compare instruction first before you execute the jump instruction if you change values in R04 and R05)



As then values are not equal so it will not jump

1. Test this instruction by manually putting values in registers R04 and R05 and then first executing the compare instruction and then executing the jump instruction (**i.e.** You will need to execute the compare instruction first before you execute the jump instruction every time you change values in R04 and R05)



