Computer Program Execution

A. Introduction

Objectives

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

- Use the CPU simulator to create basic CPU instructions
- Use the simulator to execute basic CPU instructions
- Use CPU instructions to move data to registers, compare values in registers, push data to the stack, pop data from the stack, jump to address locations and add values held in registers.
- Explain the function of the special CPU registers PC (Program Counter) and SR (Status Register) and status flags OV, N and Z.
- Produce code for simple conditional statements and loops.
- Use direct addressing mode of accessing data in memory
- Use indirect addressing mode of accessing data in memory
- Write a subroutine and call it
- Pass parameters to a subroutine

B. Processor (CPU) Simulators

The computer architecture tutorials are supported by simulators, which are created to help underpin theoretical concepts normally covered during the lectures. The simulators provide visual and animated representation of mechanisms involved and enable the students to observe the hidden inner workings of systems, which would be difficult or impossible to do otherwise. The added advantage of using simulators is that they allow the students to experiment and explore different technological aspects of systems without having to install and configure the real systems.

C. Basic Theory

The programming model of computer architecture defines those low-level architectural components, which include the following

- CPU instruction set
- CPU registers
- Different ways of addressing instructions and data in instructions, i.e. addressing modes such as direct and indirect addressing.

It also defines interactions between the above components. It is this low-level programming model which makes programmed computations possible.

D. Simulator Details

This section includes basic information on the simulator enabling the students to use the simulator. The tutor(s) will be available to help in case of difficulty in using the simulator. The main window is composed of several views that represent different functional parts of the simulated Central Processing Unit. These are shown in Figure 1 below and are composed of

- CPU Instructions memory
- Special CPU registers
- CPU (general purpose) registers
- Program stack
- Program creation and program running features

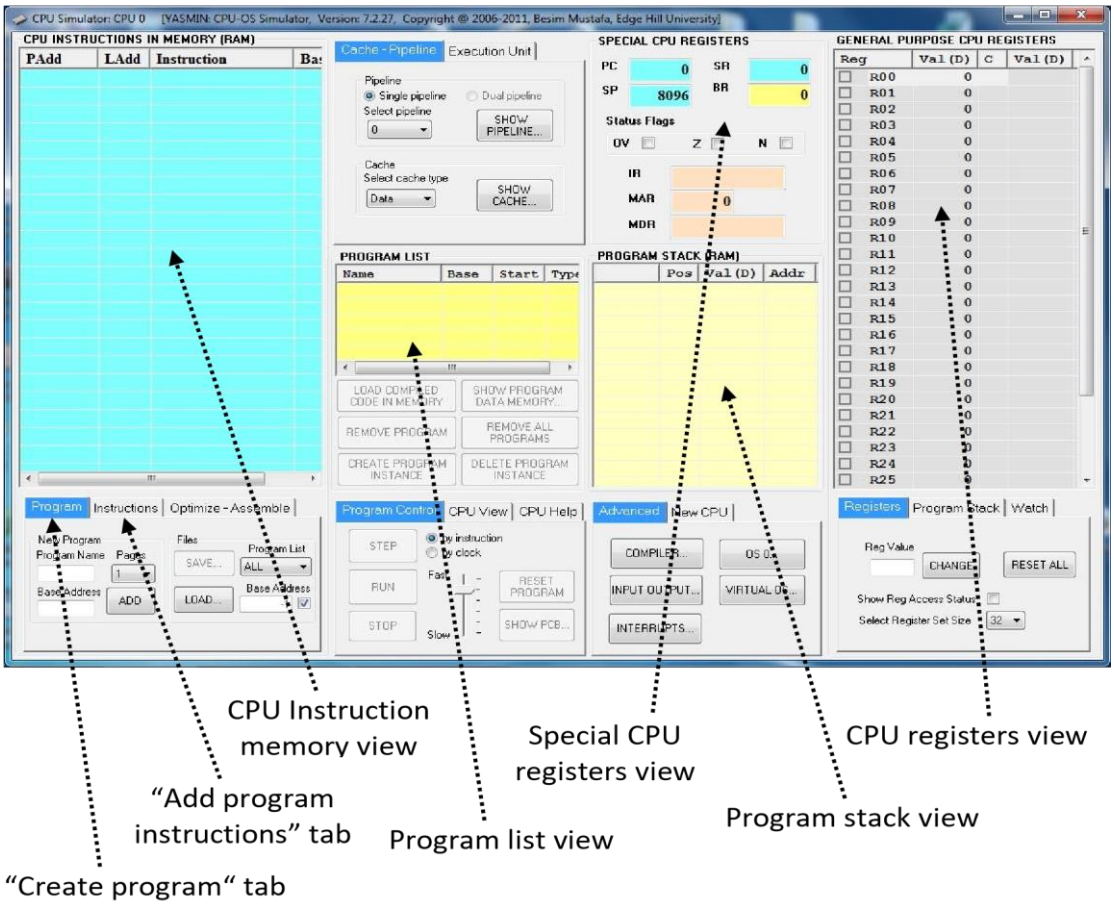

Image 1 – CPU Simulator window

Figure 1. CPU-OS Simulator Window

The parts of the simulator relevant to this lab are described below. Please read this information carefully and try to identify the different parts on the real CPU Simulator window **BEFORE** attempting the following exercises. Use this information in conjunction with the exercises that follow.

1. CPU instruction memory view

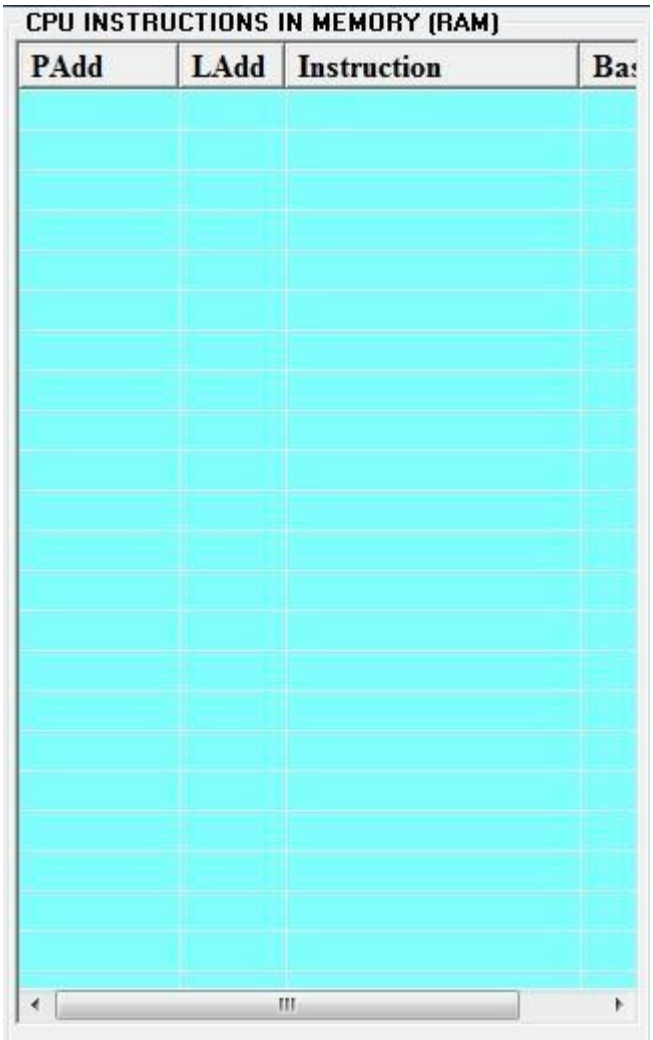

Figure 2. Instruction memory view

This view contains the program instructions. The instructions are displayed as sequences of low-level instruction mnemonics (in assembler-level format) and not as binary code. This is done for clarity and makes code more readable by humans.

Each instruction is associated with two addresses: the physical address (**PAdd**) and the logical address (**LAdd**). This view also displays the base address (**Base**) against each instruction. The sequence of instructions belonging to the same program will have the same base address.

2. Special CPU registers view

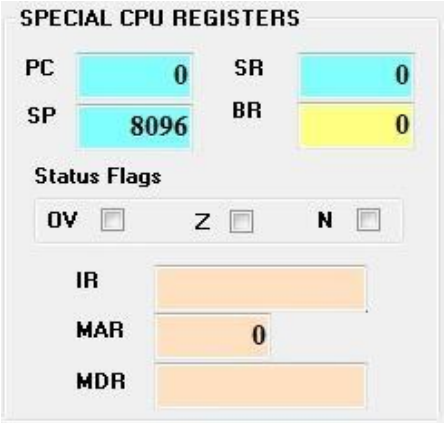

Figure 3. Special CPU registers view

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 (see below).

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

3. CPU registers view

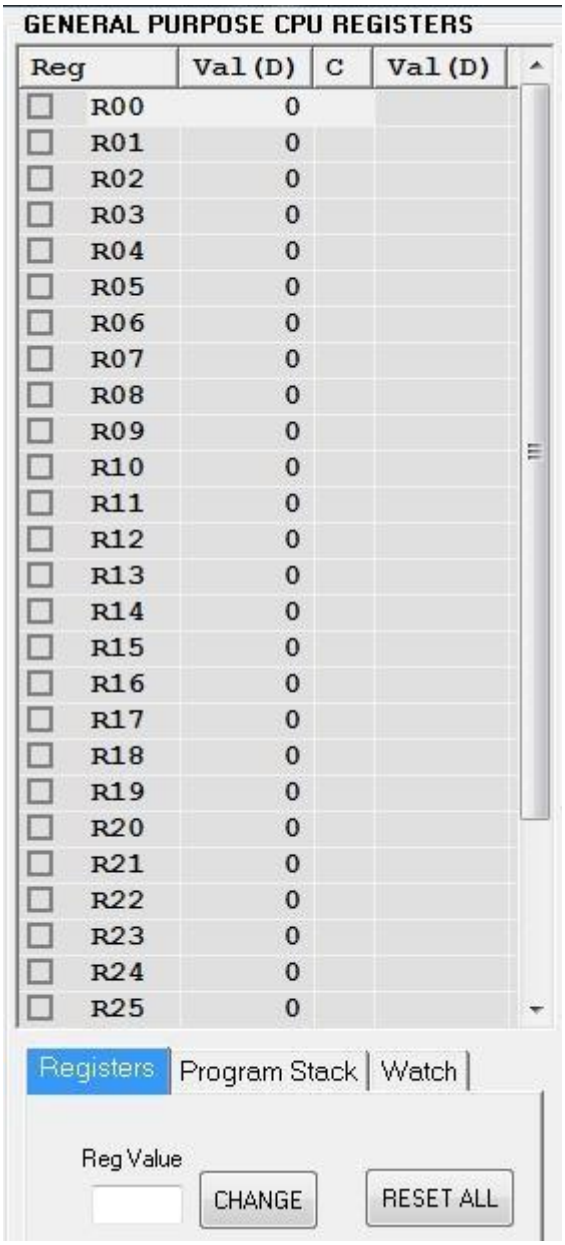

Figure 4. CPU Registers view

The register set view shows the contents of all the general-purpose registers, which are used to maintain temporary values as the program's instructions are executed. Registers are very fast memories that hold temporary values while the CPU executes instructions.

This architecture supports from 8 to 64 registers. These registers are often used to hold values of a program's variables as defined in high-level languages.

Not all architectures have this many registers. Some have more (e.g. 128 register) and some others have less (e.g. 8 registers). In all cases, these registers serve similar purposes.

This view displays each register's name (**Reg**), its current value (**Val**) and some additional values, which are reserved for program debugging. It can also be used to reset the individual register values manually which is often useful for advanced debugging. To manually change a register's content, first select the register then enter the new value in the text box, **Reg Value**, and click on the **CHANGE** button in the **Registers** tab.

4. Program stack view

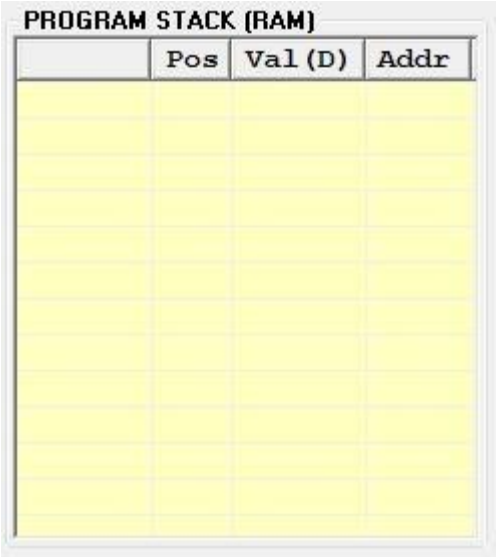

Figure 5. Program stack view

The program stack is another area which maintains temporary values as the instructions are executed. The stack is a LIFO (last-in-first-out) data structure. It is also used for efficient interrupt handling and sub-routine calls. Each program has its own individual stack.

The CPU instructions PSH (push) and POP are used to store values on top of stack and pop values from top of stack respectively.

5. Program list view

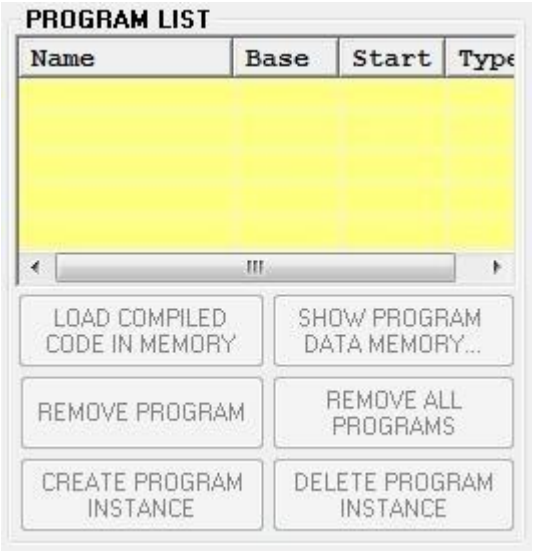

Figure 6. Program List View

Use the **REMOVE PROGRAM** button to remove the selected program from the list; use the **REMOVE ALL PROGRAMS** button to remove all the programs from the list. Note that when a program is removed, its instructions are also removed from the **Instruction Memory View**.

6. Program creation

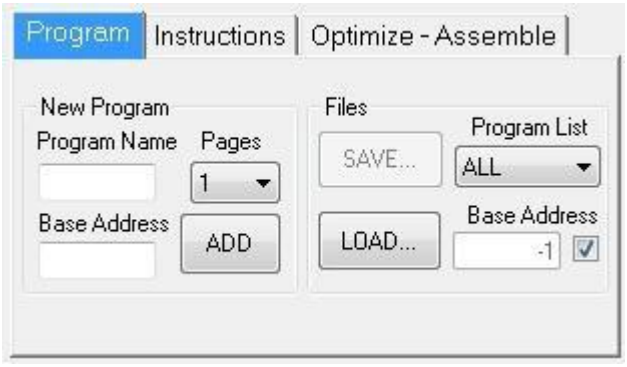

Figure 7. Create program tab

To create a new program enter its name in the **Program Name** box and its base address in the **Base Address** box then click on the **ADD** button. The new program's name will appear in the Program List view (see Figure 6).

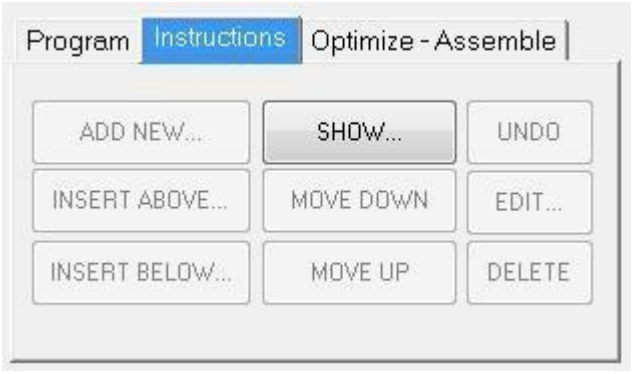

Figure 8. Add program instructions tab

Use **ADD NEW…** button to add a new instruction; use **EDIT…** button to edit the selected instruction; use **MOVE DOWN**/ **MOVE UP** buttons to move the selected instruction down or up; use **INSERT ABOVE…/INSERT BELOW…** buttons to insert a new instruction above or below the selected instruction respectively.

7. Program data memory view

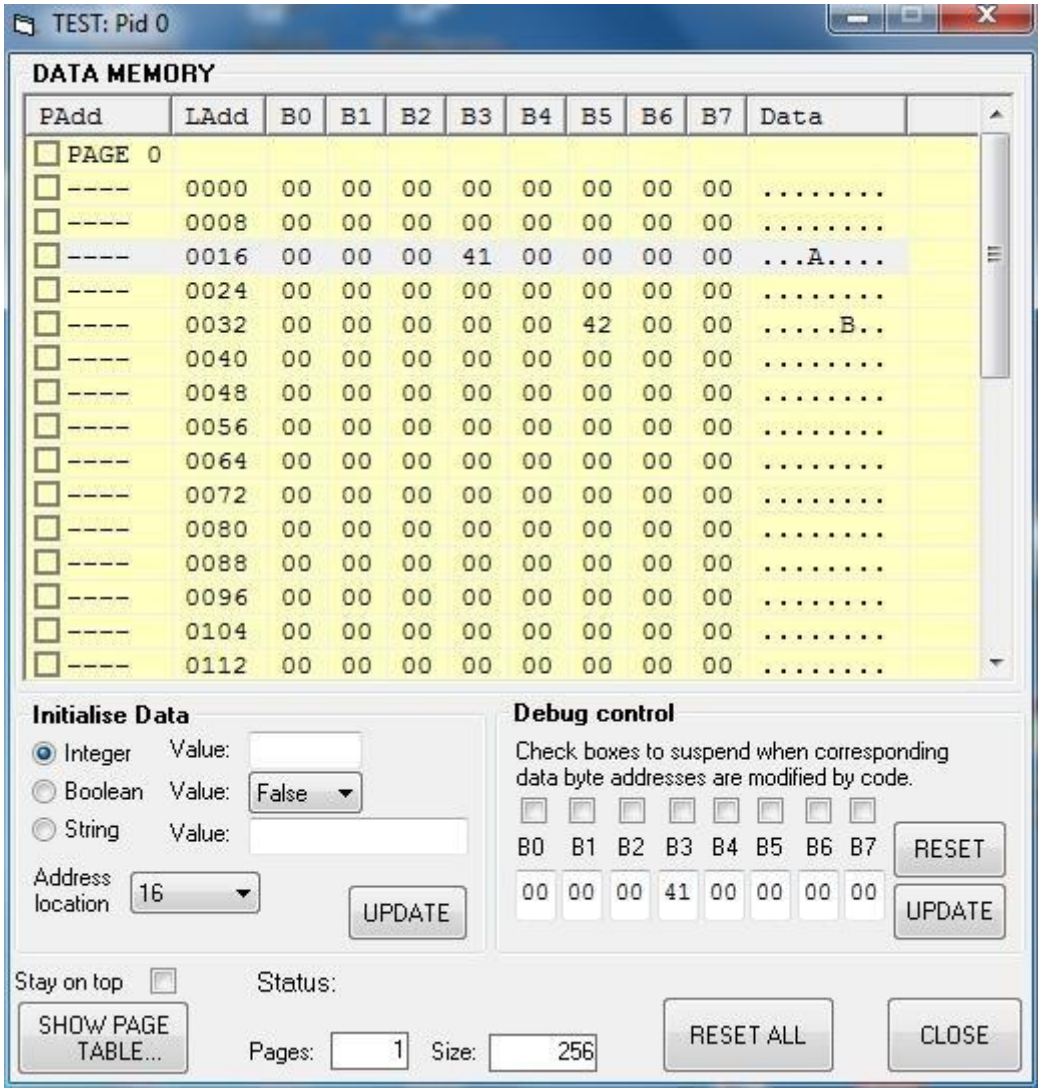

Figure 9. Program data memory view

The CPU instructions that access that part of the memory containing data can write or read the data in addressed locations. This data can be seen in the memory pages window shown in Figure 9 above.

You can display this window by clicking the **SHOW PROGRAM DATA MEMORY…** button in the main CPU window. The **Ladd** (logical address) column shows the starting address of each line in the display. Each line of the display represents 8 bytes of data. Columns **B0** through to **B7** represent bytes 0 to 7 on each line. The **Data** column shows the displayable characters corresponding to the 8 bytes. Those bytes that correspond to non-displayable characters are shown as dots. The data bytes are displayed in hex format only. For example, in Figure 9, there are non-zero data bytes in address locations 19 and 37. These data bytes correspond to displayable characters capital A and B.

To manually 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.

E. Lab Exercises - Investigate and Explore

The lab exercises are a series of activities, which are carried out by the students under basic guidelines. So, how is this tutorial conducted? The students are expected to follow the instructions given in order to identify and locate the required information, to act upon it and make notes of their observations. In order to be able to do these activities you should consult the information in **Section D** above and also frequently refer to the **Appendix** for information on various CPU instructions you will be asked to create and use.

Remember, you need to carefully read and understand the instructions before you attempt each activity.

Enter the instructions you create in order to answer the questions in the blank boxes. Refer to Appendix at the end of this document to find the details on the desired instructions. You are expected to execute the instructions you created on the simulator in order to verify your answers.

START HERE: Creating a program and adding instructions to it

Now, let us start. First you need to place some instructions in the **Instruction Memory View** (see Figure 2)**,** representing the RAM in the real machine, before executing any instructions. To do this, follow the steps below:

In the **Program** tab (see Figure 7), first enter a **Program Name**, and then enter a **Base Address** (this can be any number, but for this exercise use 100). Click on the **ADD** button. A new program name will be entered in the **Program List** view (see Figure 6). By the way, you can use the **SAVE…** button to save instructions in a file. You can also use the **LOAD…** button to load instructions from a file.

You are now ready to enter instructions into the CPU Simulator. You do this by clicking on the **ADD NEW…** button in the **Instructions** tab (see Figure 8). This will display the **Instructions: CPU0** window. You use this window to select and enter the CPU instructions. Further information on this can be found on this module's site. **Appendix** lists the instructions you will need in this exercise.

Now, have a go at the following activities (enter your answers in the text boxes provided**).**

A word of caution: Regularly save your code in a file in case the simulator crashes in which case you can restart the simulator and re-load your file.

Part A: (Refer to the appendix for help with CPU instructions)

1. Create an instruction, which moves number 5 to register R00.

2. Execute the above instruction (You do this by double-clicking on the instruction). Observe the result in the **CPU Registers** view (Figure 4).

3. Create an instruction, which moves number 7 to register R01.

4. Execute it (You do this by double-clicking on the instruction)

5. Observe the contents of R00 and R01 in the **CPU Registers** view (Figure 4).

6. Create an instruction, which adds the contents of R00 and R01.

7. Execute it.

8. Note down which register the result is put in.

9. Create an instruction, which pushes the value in the above register to the top of the program stack, and then execute it. Observe the value in **Program Stack** (Figure 5).

10. Create an instruction to push number 2 on top of the program stack and execute it. Observe the value in **Program Stack** (Figure 5).

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

12. Execute it.

13. Observe the value in the **PC** register. This is the address of the next instruction to be executed. Make a note of the instruction it is pointing to?

14. Create an instruction to pop the value on top of the **Program Stack** into register R02.

15. Execute it.

16. Create an instruction to pop the value on top of the **Program Stack** into register R03.

17. Execute it.

18. Execute the last instruction once again. What happened? Explain.

19. Create a compare instruction, which compares values in registers R04 and R05.

20. Manually insert two equal values in registers R04 and R05 (Figure 4).

21. Execute the above compare instruction.

22. Which of the status flags **OV/Z/N** is set (i.e. box is checked)?

23. Manually insert a value in first register greater than that in second register.

24. Execute the compare instruction again.

25. Which of the status flags **OV/Z/N** is set?

26. Manually insert a value in first register smaller than that in second register.
27. Execute the compare instruction once again.

28. Which of the status flags **OV/Z/N** is set?

29. Create an instruction, which will jump to the first instruction if the values in registers R04 and R05 are equal.

30. Test the above instruction by manually putting equal values in registers R04 and R05, then first executing the compare instruction followed by executing the jump instruction (**Remember**: You execute an instruction by double-clicking on it). If it worked the first instruction should be highlighted.

31. Now that you have some understanding of basic CPU instructions and are able to program the simulator here is a bit of challenge for you: preparing a little program loop. Program loops are extremely useful and are very frequently used by computer programs. Here's what you have to do:

    1. Create an instruction that moves number 0 into register R01
    2. Create an instruction that adds number 1 to register R01
    3. Create an instruction that compares number 10 and register R01
    4. Create an instruction that jumps back to instruction 2 above if R01 is not equal to number 10.
    5. Create a HLT instruction.

Make a note of the instructions 1 to 5 you created above in the box below:

6. Starting from instruction 1 manually execute instructions 1 to 4 one after the other. What happened when you executed instruction 4?

7. Now first click on the **RESET PROGRAM** button (see **Program Control** tab in Figure 1) and then highlight instruction 1 above. Next click on the **RUN** button. Now observe the loop in action.

Part B: (Refer to the appendix for help with CPU instructions)

1. Produce the code for a conditional statement such that if the value in register R02 is **greater than** (>) the value in register R01 then register R03 is set to 8. Test it on the simulator.

2. Produce the code for a loop that repeats **5 times** where the value of register R02 is incremented by 2 every time the loop repeats. Test it on the simulator.

3. The numbers 4, -3, 5 and -6 are manually pushed on top of stack in that order. Produce the code for a routine that **pops** two numbers from top of stack **multiplies** them and **pushes** the result back to top of stack. The routine repeats this until there is only one number left on top of stack. Test the code on the simulator.

Part C: Instructions for writing to and reading from memory (RAM):

1. Locate the instruction that **stores** a byte in program data memory and use it to store number 65 in memory address location 20 (this uses **memory direct** addressing method).

2. Move number 51 into register R04. Use the store instruction to **store** the contents of R04 in program data memory location 21 (this uses **register direct** addressing method).

3. Move number 22 into register R04. Use this information to indirectly **store** number 59 in program data memory (hint: you will need to use the '@' prefix for this – see the list of instructions in appendix) - (this uses **register indirect** addressing method).

4. Locate the instruction that **loads** a byte from program data memory into a register. Use this to load the number in memory address 22 into register R10.

5. Write a loop in which 10 numbers from 41 to 50 are written in program data memory starting from memory address 24 (hint: use **register indirect** addressing where register R04 indirectly represents memory address to write to and increment it to address increasing memory locations).

6. Manually initialise part of program data memory starting from address 40 with string “CPU INSTRUCTIONS RULE” (see section D above on how to do this). Write a loop in which this string is copied to another part of the program data memory starting from address 64.

Part D: Instructions for calling subroutines and passing parameters to subroutines:

1. Add the following code and run it starting from the first MOV instruction (to do this you need to first select this instruction and then click on the RUN button).

   NOTE:
   Ask your tutor how to add labels to your code. A Label represents the address of the instruction immediately following it. For example, 'Label2' below represents the address of the MOV instruction following it. Labels are used by the jump instructions by putting a '$' in front of the label name, e.g. the 'JMP $Label2' instruction will jump to the instruction at address represented by the label 'Label2'.

```
Label2
MOV #16, R03
MOV #h41, R04
Label3
STB R04, @R03
ADD #1, R03
ADD #1, R04
CMP #h4F, R04
JNE $Label3
HLT
```

   NOTE:
   This code stores numbers hex 41 to hex 4F starting from program memory address 16. These numbers are ASCII codes for displayable characters of the alphabet. You need to understand how this is done by this code in order to benefit from these exercises.

   a. Make a note of what you see in the program's data area after the program stops running:

   b. Suggest what the significance of @ in **@R03** might be:

2. Add the following subroutine calling code but do **NOT** run it yet:

```
MSF
CAL $Label2
HLT
```

   Now convert the code in (1) above into a subroutine by simply replacing the **HLT** instruction with the **RET** instruction.

a. Make a note of the contents of the **PROGRAM STACK** after the instruction **MSF** is executed (you can execute this instruction by simply double-clicking on it):

b. Make a note of the contents of the **PROGRAM STACK** after the instruction **CAL** is executed (you can execute this instruction by simply double-clicking on it):

c. What is the significance of the additional information on the stack after executing the **CAL** instruction?

3. Let's make the above subroutine a little more flexible. Suppose we wish to change the number of characters stored when calling the subroutine. Modify the calling code in (2) as below:

```
MSF
PSH #h60        ← puts the number hex 60 on top of the stack

CAL $Label2
HLT
```

a. Now modify the subroutine code in (1) as below and run the above calling code (starting from the **MSF** instruction). Pay particular attention to the behaviour of the stack:

```
Label2
MOV #16, R03
MOV #h41, R04
POP R05         ← puts the number on top of the stack in R05

Label3
STB R04, @R03
ADD #1, R03
ADD #1, R04
CMP R05, R04    ← compares R05 with R04

JNE $Label3
RET
```

b. Add a second parameter to the above code that can provide different starting addresses for the data transfer to memory and test it on the simulator. Write the modified code below:

4. Write a subroutine that takes two numbers as parameters, adds them and returns the result in register R00, i.e. when the subroutine is exited the result is available in R00 to the rest of the program. Test it on the simulator by writing the calling instructions that include passing two numbers on the stack. Copy the code below (including the calling instructions):

5. Write a subroutine that takes two numbers as parameters, compares them and returns the higher of the two as the result in register R00 (consider what should happen if the two numbers are the same). Test it on the simulator by writing the calling instructions that include passing two numbers on the stack. Copy the subroutine code alone below:

***** End of Exercises *****

Appendix – CPU Simulator Instruction Sub-set

Inst.	Description
Data transfer instructions	
MOV	Move data to register; move register to register e.g. **MOV #2, R01** moves number 2 into register R01 **MOV R01, R03** moves contents of register R01 into register R03
LDB	Load a byte from memory to register e.g. **LDB 1022, R03** loads a byte from memory address 1022 into R03 **LDB @R02, R05** loads a byte from memory the address of which is in R02
LDW	Load a word (2 bytes) from memory to register Same as in LDB but a word (i.e. 2 bytes) is loaded into a register
STB	Store a byte from register to memory **STB R07, 2146** stores a byte from R07 into memory address 2146 **STB R04, @R08** stores a byte from R04 into memory address of which is in R08
STW	Store a word (2 bytes) from register to memory Same as in STB but a word (i.e. 2 bytes) is loaded stored in memory
PSH	Push data to top of hardware stack (TOS); push register to TOS e.g. **PSH #6** pushes number 6 on top of the stack **PSH R03** pushes the contents of register R03 on top of the stack
POP	Pop data from top of hardware stack to register e.g. **POP R05** pops contents of top of stack into register R05 **Note**: If you try to POP from an empty stack you will get the error message "Stack underflow".
Arithmetic instructions	
ADD	Add number to register; add register to register e.g. **ADD #3, R02** adds number 3 to contents of register R02 and stores the result in register R02. **ADD R00, R01** adds contents of register R00 to contents of register R01 and stores the result in register R01.
SUB	Subtract number from register; subtract register from register
MUL	Multiply number with register; multiply register with register
DIV	Divide number with register; divide register with register

Control transfer instructions	
JMP	Jump to instruction address unconditionally e.g. **JMP 100** unconditionally jumps to address location 100 where there is another instruction
JLT	Jump to instruction address if less than (after last comparison)
JGT	Jump to instruction address if greater than (after last comparison)
JEQ	Jump to instruction address if equal (after last comparison instruction) e.g. **JEQ 200** jumps to address location 200 if the previous comparison instruction result indicates that the two numbers are equal, i.e. the **Z** status flag is set (the **Z** box will be checked in this case).
JNE	Jump to instruction address if not equal (after last comparison)
MSF	Mark Stack Frame instruction is used in conjunction with the CAL instruction. e.g. **MSF** reserve a space for the return address on program stack **CAL 1456** save the return address in the reserved space and jump to subroutine in address location 1456
CAL	Jump to subroutine address (saves the return address on program stack) This instruction is used in conjunction with the **MSF** instruction. You'll need an MSF instruction before the CAL instruction. See the example above
RET	Return from subroutine (uses the return address on stack)
SWI	Software interrupt (used to request OS help)
HLT	Halt simulation
Comparison instruction	
CMP	Compare number with register; compare register with register e.g. **CMP #5, R02** compare number 5 with the contents of register R02 **CMP R01, R03** compare the contents of registers R01 and R03 Note: If R01 = R03 then the status flag **Z** will be set, i.e. the **Z** box is checked. If R01 < R03 then none of the status flags will be set, i.e. none of the status flag boxes are checked. If R01 > R03 then the status flag **N** will be set, i.e. the **N** status box is checked.

Input, output instructions	
IN	Get input data (if available) from an external IO device
OUT	Output data to an external IO device e.g. **OUT 16, 0** outputs contents of data in location 16 to the console (the second parameter must always be a 0)

****