

# Programming Model 2

---

## A. Introduction

### Objectives

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

- Use direct and indirect addressing modes of accessing data in memory
- Create an iterative loop of instructions
- Display text on console using an IO instruction
- Create a sub-routine, call and return from subroutine
- Pass parameters to a subroutine

---

## B. Processor (CPU) Simulators

The computer architecture tutorials are supported by simulators, which are created to 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

It also defines interaction between the above components. It is this low-level programming model which makes programmed computations possible. **You should do additional reading in order to form a better understanding of the different parts of a modern CPU architecture (refer to the recommended reading list available in the module handbook and on the BB).**

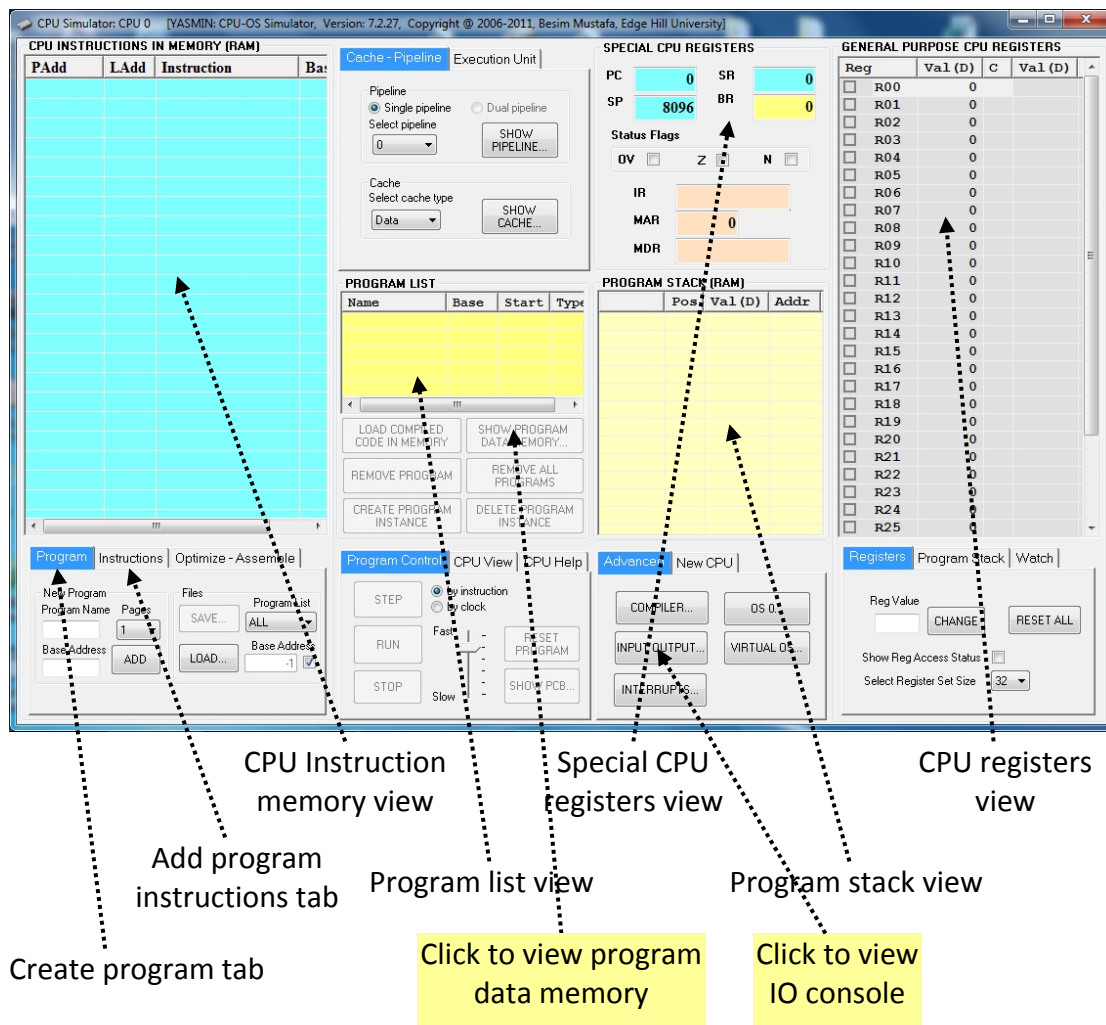
---

## D. Simulator Details

This section includes some basic information on the simulator, which should enable the students to use the simulator. The tutor(s) will be available to help anyone experiencing difficulty in using the simulator. The simulator for this lab is an application running on a PC running MS Windows operating system.

The main simulator window is composed of several views, which represent different functional parts of the simulated processor. These are shown in Image 1 below and are composed of

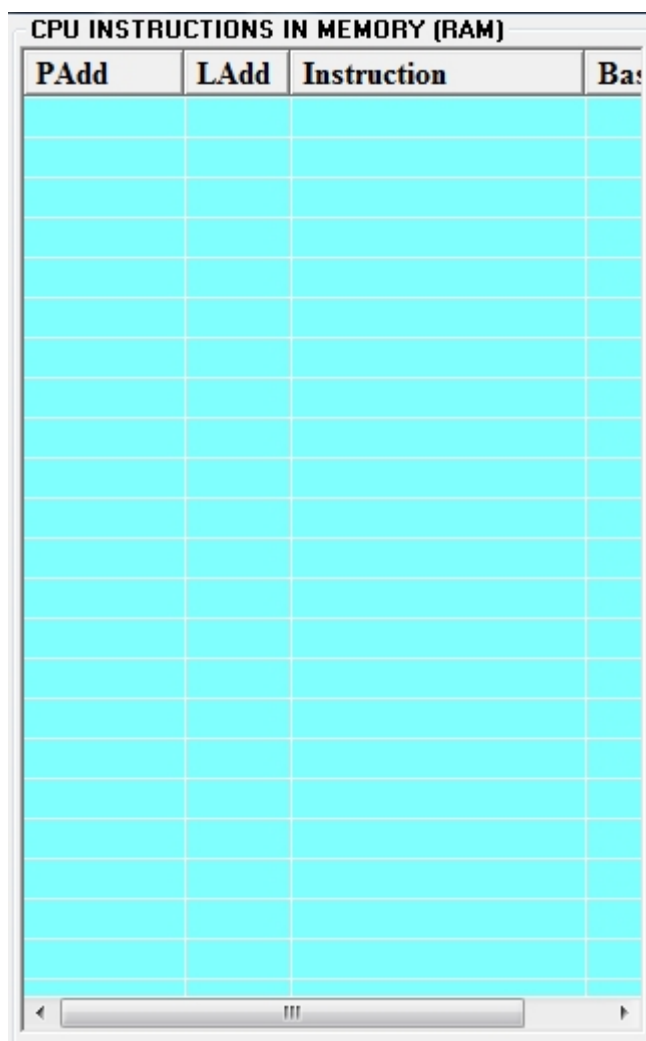
- CPU Instruction memory
- Special CPU registers
- CPU (general purpose) registers
- Program stack
- Program creation and running features
- Memory in which data is stored
- Input, output console



**Image 1 – CPU 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 CPU Simulator window BEFORE attempting the following exercises. Use the information in this section in conjunction with the exercises that follow.**

## 1. CPU instruction memory view



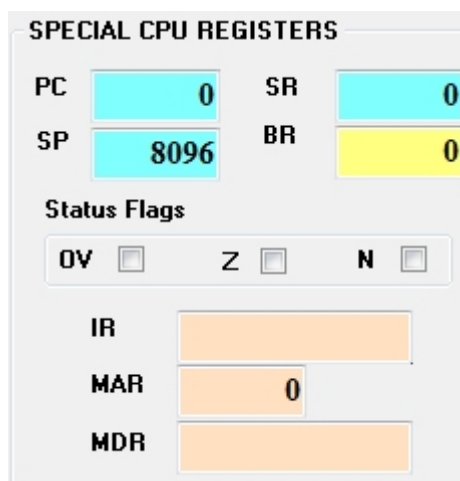
PAdd	LAdd	Instruction	Base
------	------	-------------	------

**Image2 - Instruction memory view**

This view contains the program instructions. The instructions are displayed as sequences of low-level instruction mnemonics (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



SPECIAL CPU REGISTERS	
PC	0
SP	8096
SR	0
BR	0
Status Flags	
OV	<input type="checkbox"/>
Z	<input type="checkbox"/>
N	<input type="checkbox"/>
IR	
MAR	0
MDR	

**Image 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

GENERAL PURPOSE CPU REGISTERS				
Reg	Val (D)	C	Val (D)	
<input type="checkbox"/> R00	0			
<input type="checkbox"/> R01	0			
<input type="checkbox"/> R02	0			
<input type="checkbox"/> R03	0			
<input type="checkbox"/> R04	0			
<input type="checkbox"/> R05	0			
<input type="checkbox"/> R06	0			
<input type="checkbox"/> R07	0			
<input type="checkbox"/> R08	0			
<input type="checkbox"/> R09	0			
<input type="checkbox"/> R10	0			
<input type="checkbox"/> R11	0			
<input type="checkbox"/> R12	0			
<input type="checkbox"/> R13	0			
<input type="checkbox"/> R14	0			
<input type="checkbox"/> R15	0			
<input type="checkbox"/> R16	0			
<input type="checkbox"/> R17	0			
<input type="checkbox"/> R18	0			
<input type="checkbox"/> R19	0			
<input type="checkbox"/> R20	0			
<input type="checkbox"/> R21	0			
<input type="checkbox"/> R22	0			
<input type="checkbox"/> R23	0			
<input type="checkbox"/> R24	0			
<input type="checkbox"/> R25	0			

Registers

Program Stack

Watch

Reg Value

CHANGE

RESET ALL

Image 4 – CPU Registers view

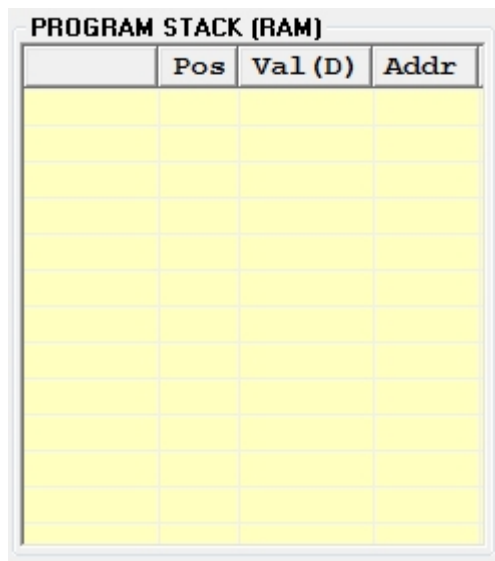
The register set view shows the contents of all the general-purpose registers, which are used to maintain temporary values as the program's instructions are executed. **Registers are very fast memories that hold temporary values while the CPU executes instructions.**

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



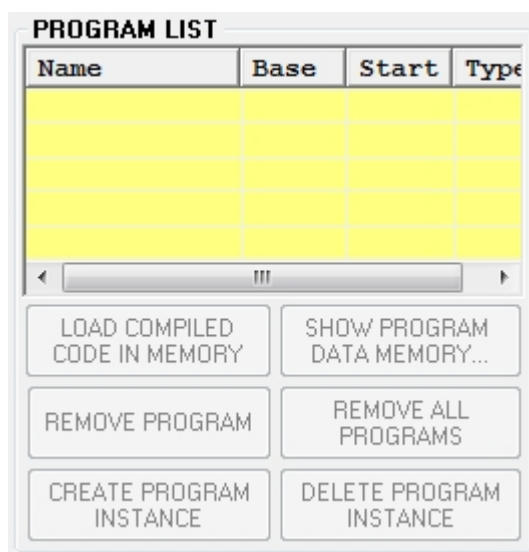
	Pos	Val (D)	Addr

Image 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 often 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



Name	Base	Start	Type

LOAD COMPILED CODE IN MEMORY

SHOW PROGRAM DATA MEMORY...

REMOVE PROGRAM

REMOVE ALL PROGRAMS

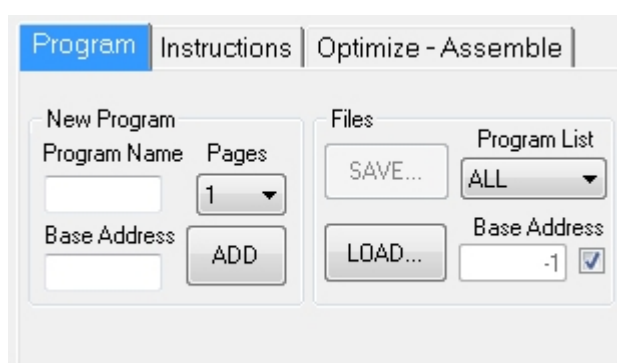
CREATE PROGRAM INSTANCE

DELETE PROGRAM INSTANCE

Image 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** too.

#### 6. Program creation



Program | Instructions | Optimize - Assemble

New Program

Program Name

Base Address

PAGES

1

ADD

Files

SAVE...

LOAD...

Program List

ALL

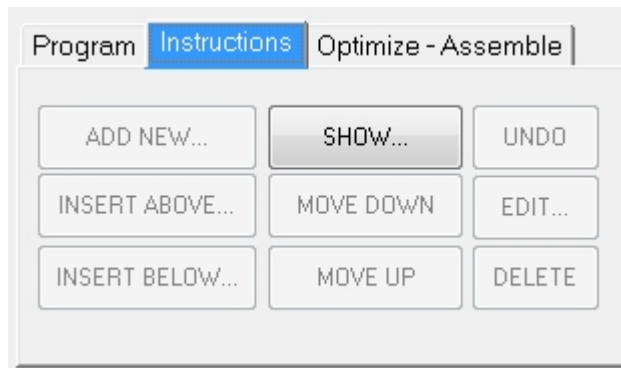
Base Address

-1

☒

Image 7 – Create program tab

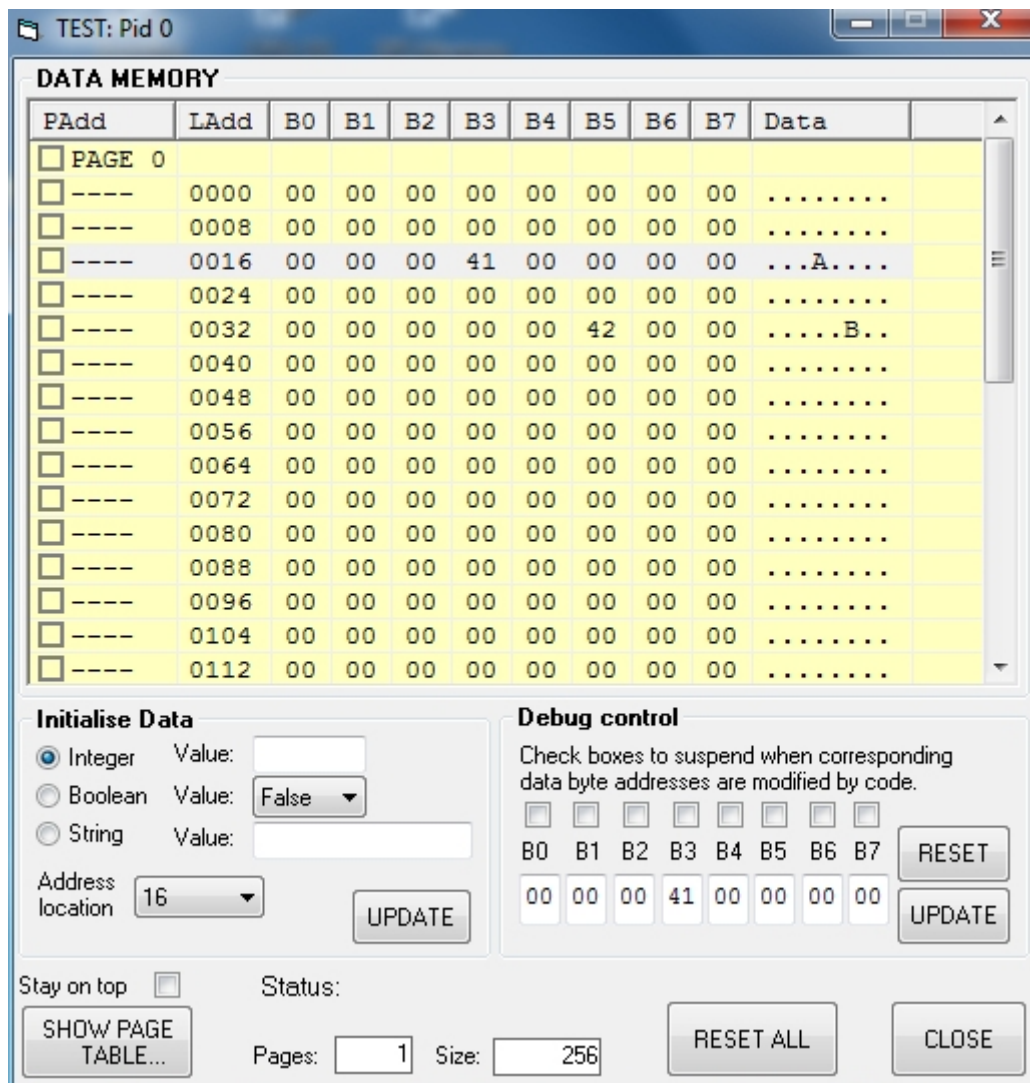
To create a new program enter its name in the **Program Name** box and its base address in the **Base Address** box then click on the **ADD** button. The new program's name will appear in the Program List view (see Image 6).



**Image 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

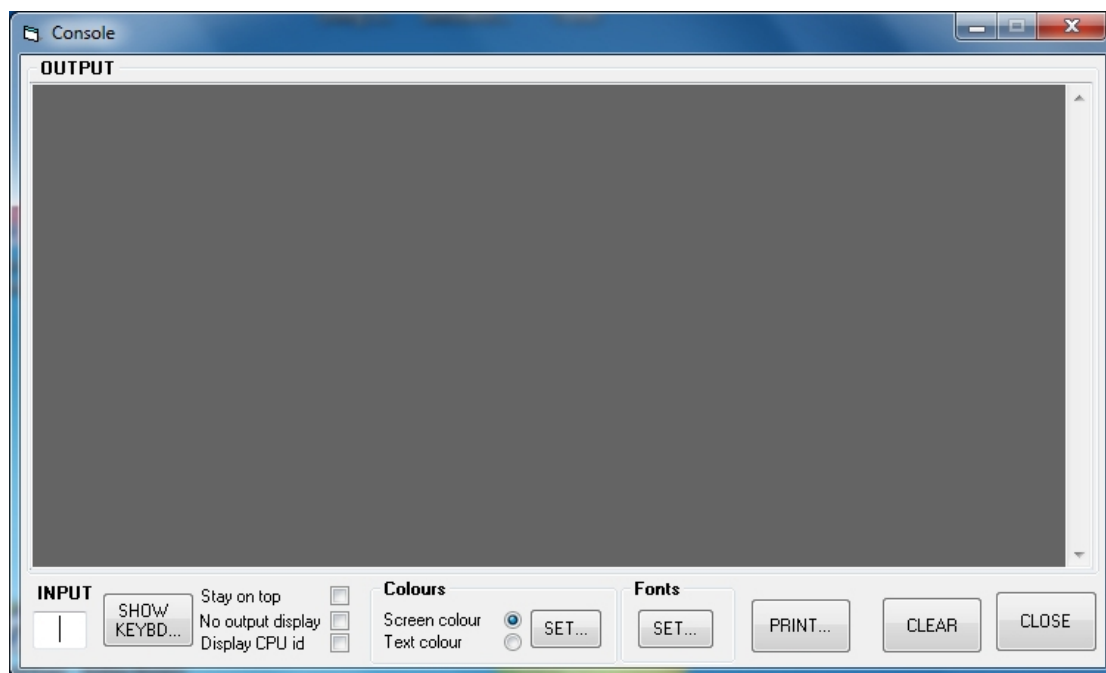


**Image 9 - Program data memory view**

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 Image 9 above. You can display this window by clicking the **SHOW PROGRAM DATA MEMORY...** button shown in Image 6 above. 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 Image 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 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.

## 8. IO console view



**Image 10 – Input, output console view**

Image 10 above shows the console which is used by programs to write messages to and read data from. It can be displayed by clicking on the **INPUT OUTPUT...** button shown in Image 1 above. Click on the **SHOW KEYBD...** button to display a small keyboard window which can be used to input data to programs requesting input.



---

## 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, make notes of their observations and offer explanations for these observations where this requested. 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.**

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

In the **Program** tab (see Image 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 Image 6). 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 Image 8). This will display the **Instructions: CPU0** window. You use this window to select and enter the CPU instructions. **Appendix** lists some of the instructions this simulator uses and also gives examples of their usage.

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.



1. In the **Appendix** at the end of this document, locate the instruction, which is used to store one byte of data in a memory location. Use it to store number 65 in address location 20 (all numbers are in decimal). This is an example of **direct addressing**. Refer to Image 9 to see how to display the contents of data memory. Make a note below of the instruction used:

STB #65, 20

2. Create an instruction to move decimal number 22 to register R01 and make a note of it below. Execute this instruction and verify the result in R01.

MOV #22, R01

3. Create an instruction to store decimal number 51 in memory location the address of which is currently stored in register R01. This is an example of **indirect addressing**. Note the use of the “@” prefix next to R01 in this case.

STB #51, @R01

4. Make a note of what you see in data memory locations 20 and 22 (refer to Image 9 for help information on how to display the data memory).

41 and 33 (HEX)

5. Now, let's create a loop. First, enter the following code. The # prefix is used to denote a literal value thus distinguishing it from an address value which does not use it. **R01** represents an arbitrary register; you can use any of the registers from **R00** to **R31**.

```
MOV #0, R01
ADD #1, R01
CMP #5, R01
JNE 0
HLT
```

6. The above code is not quite ready yet. The **JNE** instruction uses a numeric value as the address to jump to. In this case it is 0. This may not always be the case so in order to make the code more flexible we can use labels to represent instruction addresses. The simulator allows you to do this. Follow the instructions below for this:

Highlight the above **MOV** instruction (i.e. the one in the box above)  
Click on the **INSERT BELOW...** button

Type label name **L0** in the box next to the **ENTER LABEL** button in the window you use to enter instructions  
Click the **ENTER LABEL** button  
The new code should now look like this (modifications are in red colour):

```
MOV #0, R01
L0:
ADD #1, R01
CMP #5, R01
JNE 0
HLT
```

Next, highlight the **JNE** instruction  
Click on the **EDIT...** button  
Select **L0** in the drop-down list under the **Source Operand** section button in the window you use to enter instructions  
Click the **EDIT** button  
The new code should now look like this:

```
MOV #0, R01
L0:
ADD #1, R01
CMP #5, R01
JNE $L0
HLT
```

7. As you can see, the label **L0** represents the address of the instruction immediately below it, i.e. the **ADD** instruction. So now the **JNE** instruction can use **L0** as the address to jump to. As the label **L0** can represent any address this code should work anywhere in memory making it very flexible. The \$ sign indicates that L0 is a label. The above code is now ready to run. To run this program, follow the instructions below:

Click on the **RESET PROGRAM** button in the CPU Simulator window  
Highlight the **MOV** instruction, i.e. the first instruction of the program  
Adjust the speed slider to a value, say, nearest to the value 80  
Click on the **RUN** button  
After a short while the program should stop. If it appears to run too long then click on the **STOP** button and check your code. Correct it if necessary and repeat the above instructions once again.  
When the program stops make a note of the value in **R01** below

5

8. Now you'll make a slight modification to the above program. Change the program code so that the program loop is repeated as long as the value of **R01** is less than or equal to 3 (you may wish to refer to the Appendix for this) and test

it. When you get it right make a note of the value in **R01** and copy the new code below. Now, change the modified instructions back to the original instructions (you can use the **UNDO** button for this – see Image 8 above).

```
MOV #0, R01
L0:
ADD #1, R01
CMP #3, R01
JLE $L0
HLT
```

9. Ok, let's create a simple subroutine. Enter the following new code. You need to create a new label **L1** at the start of the subroutine. This label represents the starting address of the subroutine. You must enter the label using the **ENTER LABEL** button only as explained in (6). Also, make sure you select the **Direct Mem** radio button when entering the first operand value 24 of the **OUT** instruction:

```
L1:
OUT 24, 0
RET
```

10. The above subroutine code simply displays the text starting at data memory location 24 and returns (see **RET** instruction in appendix). For it to work there needs to be some text in data address location 24. You can do this manually by following the steps below:

Click on the **SHOW PROGRAM DATA MEMORY...** button (see Image 6).  
In the displayed window highlight the line 0024 under **LAdd** column  
Under **Initialise Data** click on the **String** radio button  
Enter some text in the text box labelled **Value**, e.g. *My name is Besim*  
Click the **UPDATE** button

11. Now, a subroutine is of no use by itself. For it to be useful your program must call it using the instructions **MSF** followed by **CAL** (refer to the Appendix). The **MSF** (Mark Stack Frame) is needed to reserve a place for the return address on the program stack. The **CAL** instruction needs to specify the starting address of the called subroutine. Let's modify our code so that when the above subroutine is called it displays the text repeatedly in a loop. For example, using the code added in (6) and (9) the modified program should look something like this:

```
MOV #0, R01
L0:
ADD #1, R01
MSF
CAL $L1
CMP #5, R01
JNE $L0
HLT
L1:
OUT 24, 0
RET
```

12. The above code is now ready to run. In order to see the displayed text you need to show the console window. Click on the **INPUT OUTPUT...** button (see Image 1) which will display the simulated console window. To run this program, follow the instructions below:

Click on the **RESET PROGRAM** button

Highlight the **MOV** instruction, i.e. the first instruction of the program

Adjust the speed slider to a value nearest to the value 80

Click on the **RUN** button

13. We need to make a small change to our subroutine. Currently the **OUT** instruction uses direct memory addressing, i.e. the memory address 24 is part of the instruction. We now wish to make it use indirect addressing in a way similar to that in (3). So, you'll need to place the memory address 24 in a register (any spare register). Then you need to have the **OUT** instruction use this register indirectly as the source of the address of the text to display. Run the code to test your modification. Make a note of the modified part of the program code below. Use the **UNDO** button to restore the instructions before this modification:

```
MOV #24, R02
OUT @R02, 0
```

14. Ok, let's get a little bit more ambitious as a challenge. Let's convert the loop into another subroutine and then call it. So, now we will have two subroutines where one calls the other. The following code represents this change. Notice that the **HLT** instruction is changed to the **RET** instruction and the new instructions **MSF**, **CAL** and **HLT** are added together with the new label **L2** at the top of the code. **CAL \$L2** calls the subroutine with the loop and **CAL \$L1** calls the subroutine that displays the text.

```
MSF
CAL $L2
HLT
L2:
MOV #0, R01
L0:
ADD #1, R01
MSF
CAL $L1
CMP #5, R01
JNE $L0
RET
L1:
OUT 24, 0
RET
```

Now, first reset the program then highlight the first **MSF** instruction. Run the program and verify the result in the console window as before.

15. Why stop here! Let's make it a bit more interesting. The above code will do the loop 5 times and this number is fixed. For flexibility we can pass the number of loops as a parameter to the subroutine (starting at label L2). For this we will use the **PSH** and **POP** instructions (see the Appendix). Modify your code to look like the one below and run it observing the displays on the console:

```
MSF
PSH #8
CAL $L2
HLT
L2:
POP R02
MOV #0, R01
L0:
ADD #1, R01
MSF
CAL $L1
CMP R02, R01
JNE $L0
RET
L1:
OUT 24, 0
RET
```

16. Examine the above code and briefly explain how the parameter passing works:

8 is pushed on the stack and is popped in subroutine L2, there it is used to compare with the value in R01.

17. Finally, as a real challenge, modify the above code so that a second parameter is passed to the subroutine (starting at label L2) in the same way as the first parameter is passed. The second parameter is used to initialise the register R01 to the value of this second parameter. Copy the modified code only to the point of the last modification in the box below:

```
MSF
PSH #8
PSH #0
CAL $L2
HLT
L2
POP R01
POP R02
L0
```

## Appendix - Simulator Instruction Sub-set

Inst	Description
<b>Data transfer instructions</b>	
MOV	Move data to register; move register to register e.g. <b>MOV #2, R01</b> moves number 2 into register R01 <b>MOV R01, R03</b> moves contents of register R01 into register R03
LDB	Load a byte from memory to register e.g. <b>LDB 1022, R03</b> loads a byte from memory address 1022 into R03 <b>LDB @R02, R05</b> 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 <b>STB R07, 2146</b> stores a byte from R07 into memory address 2146 <b>STB R04, @R08</b> 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. <b>PSH #6</b> pushes number 6 on top of the stack <b>PSH R03</b> pushes the contents of register R03 on top of the stack
POP	Pop data from top of hardware stack to register e.g. <b>POP R05</b> pops contents of top of stack into register R05 <b>Note:</b> If you try to POP from an empty stack you will get the error message "Stack underflow".
<b>Arithmetic instructions</b>	
ADD	Add number to register; add register to register e.g. <b>ADD #3, R02</b> adds number 3 to contents of register R02 and stores the result in register R02. <b>ADD R00, R01</b> 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
<b>Control transfer instructions</b>	
JMP	Jump to instruction address <u>unconditionally</u> e.g. <b>JMP 100</b> 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. <b>JEQ 200</b> jumps to address location 200 if the previous comparison instruction result indicates that the two numbers are equal, i.e. the <b>Z</b> status flag is set (the <b>Z</b> 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. <b>MSF</b> reserve a space for the return address on program stack <b>CAL 1456</b> 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 <b>MSF</b> 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
<b>Comparison instruction</b>	
CMP	Compare number with register; compare register with register e.g. <b>CMP #5, R02</b> compare number 5 with the contents of register R02 <b>CMP R01, R03</b> compare the contents of registers R01 and R03 Note: If R01 = R03 then the status flag <b>Z</b> will be set, i.e. the <b>Z</b> box is checked. If R03 > R01 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 <b>N</b> will be set, i.e. the <b>N</b> status box is checked.
<b>Input, output instructions</b>	
IN	Get input data (if available) from an external IO device
OUT	Output data to an external IO device e.g. <b>OUT 16, 0</b> outputs contents of data in location 16 to the console (the second parameter must always be a 0)