# Programming Model

## 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

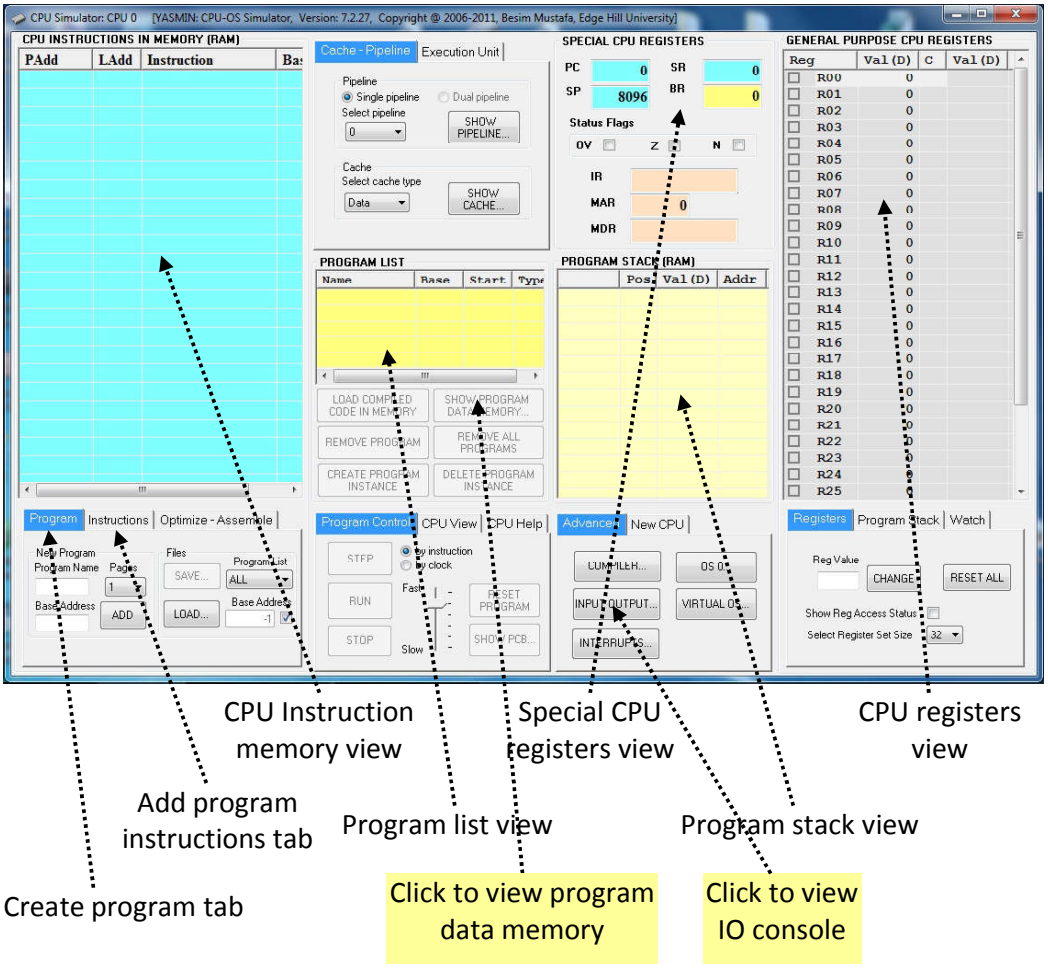


**Image 1 – CPU Simulator window**

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

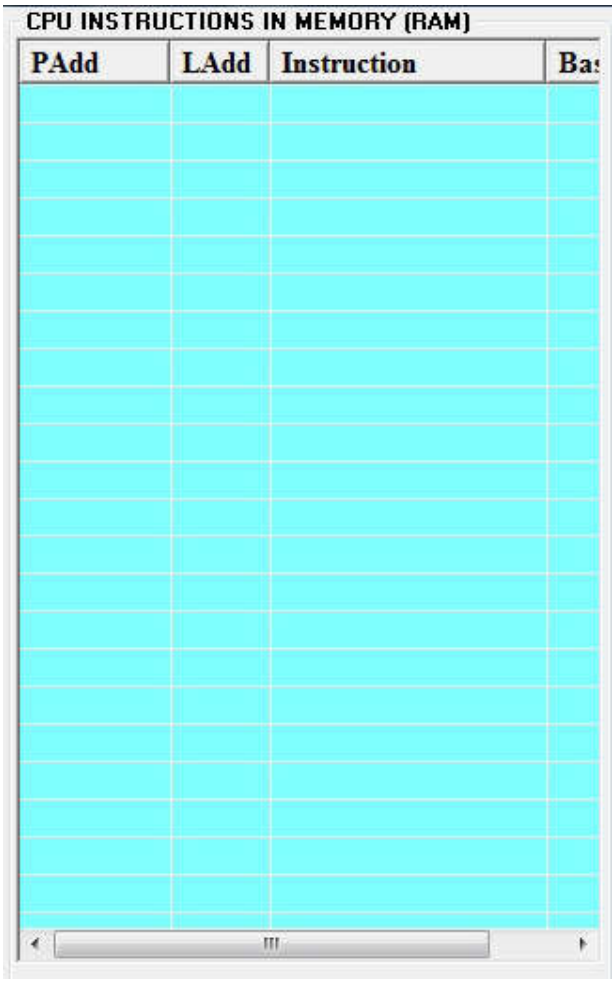


**Image2 - Instruction memory view**

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

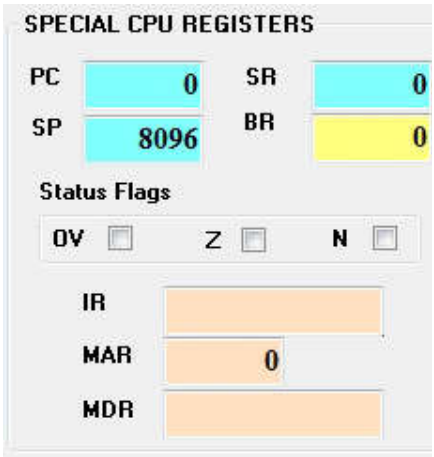


**Image 3 - Special CPU registers view**

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

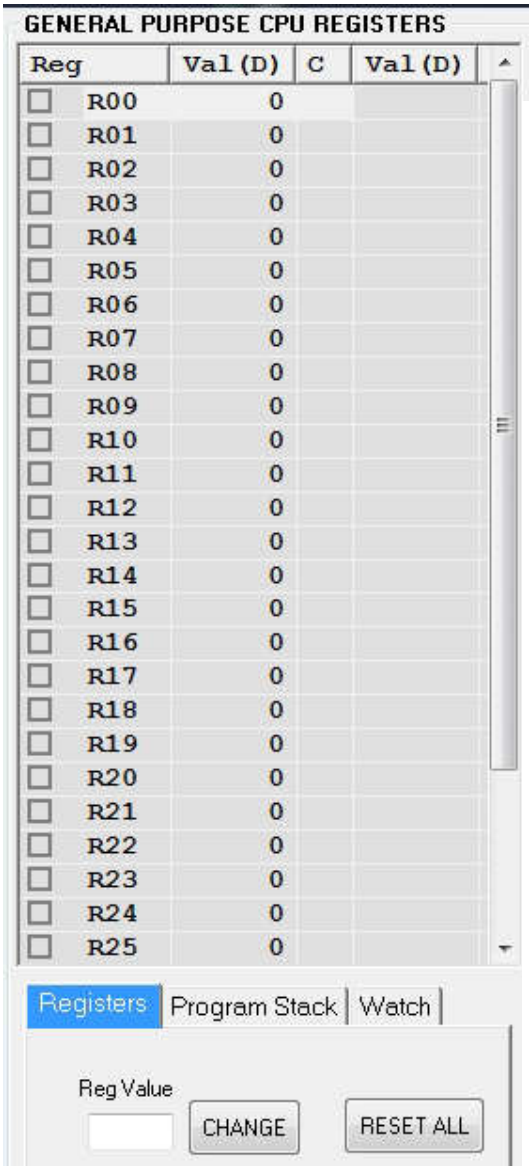


**Image 4 – CPU Registers view**

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

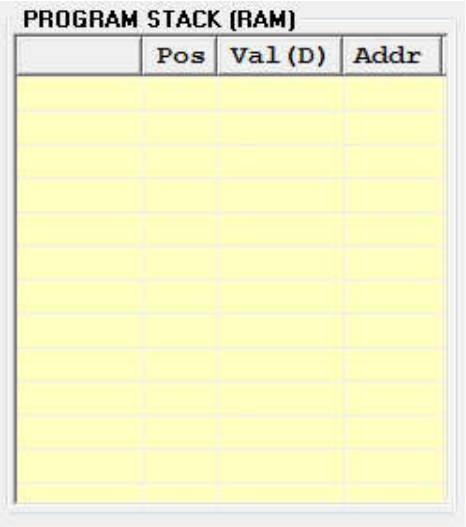


**Image 5 - Program stack view**

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

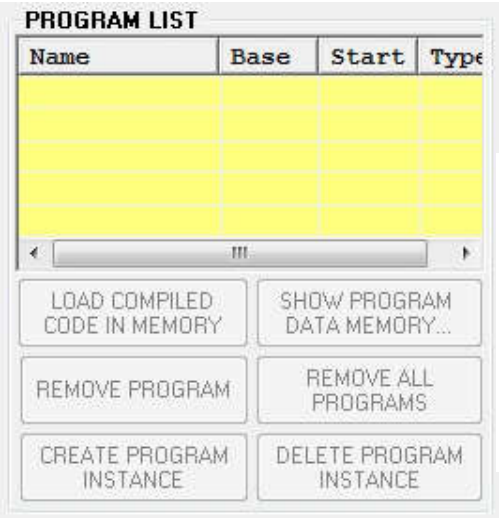


**Image 6 - Program List View**

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

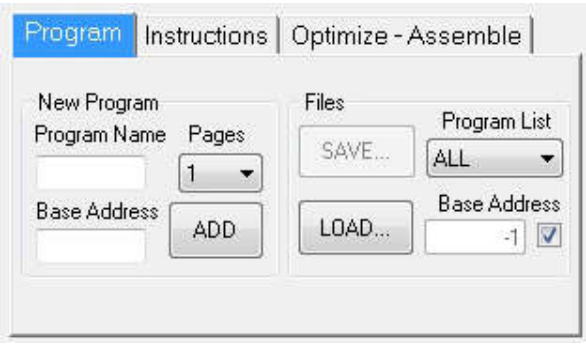


**Image 7 – Create program tab**

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).

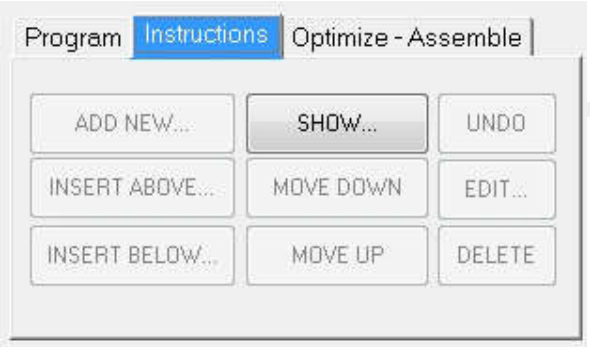


**Image 8 – Add program instructions tab**

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

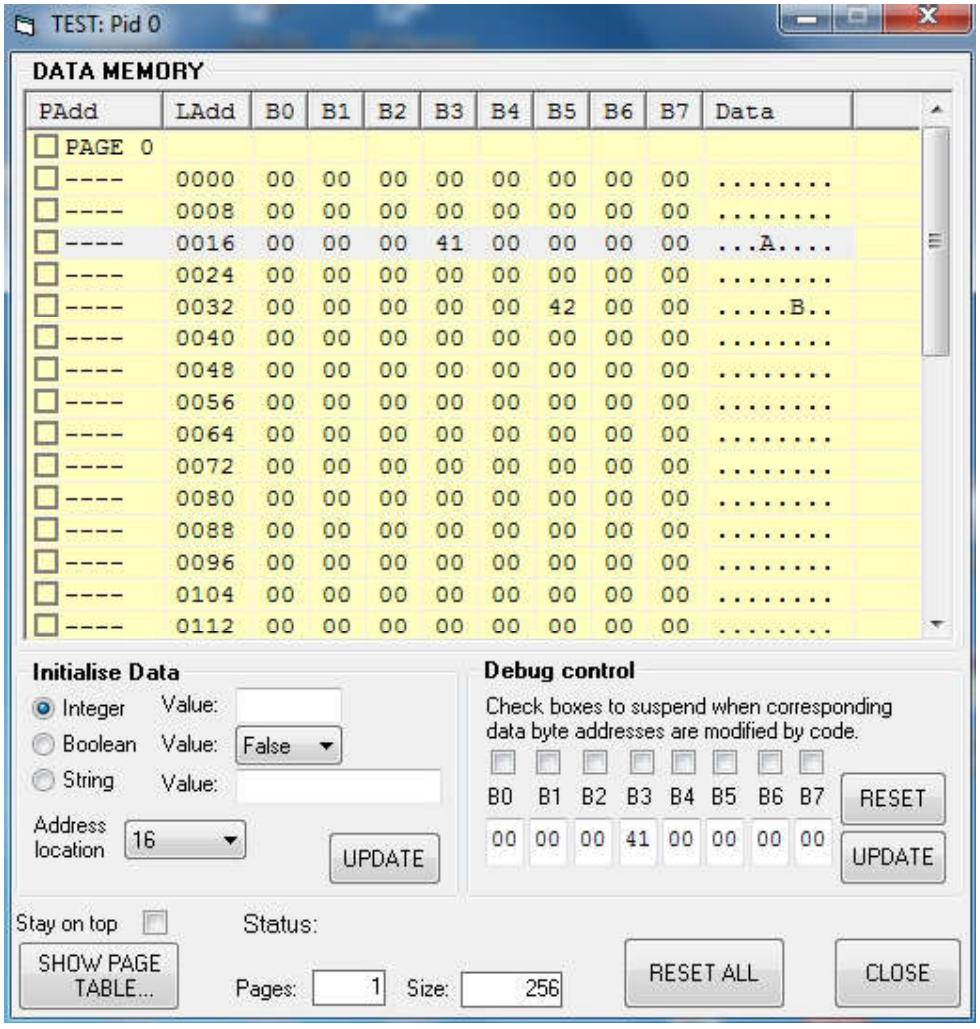


**Image 9 - Program data memory view**

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

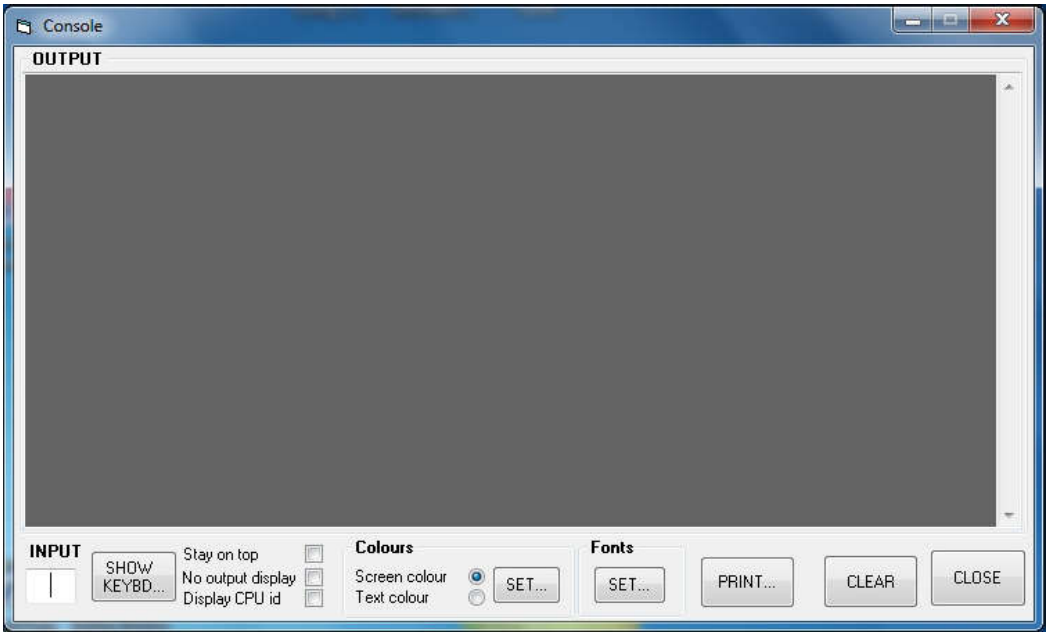


**Image 10 – Input, output console view**

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.

## Appendix - Simulator Instruction Sub-set

| Inst. | Description |
|---|---|
| **Data transfer instructions** | |
| MOV | Move data to register; move register to register<br>e.g.<br>**MOV #2, R01** moves number 2 into register R01<br>**MOV R01, R03** moves contents of register R01 into register R03 |
| LDB | Load a byte from memory to register<br>e.g.<br>**LDB 1022, R03** loads a byte from memory address 1022 into R03<br>**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<br>Same as in LDB but a word (i.e. 2 bytes) is loaded into a register |
| STB | Store a byte from register to memory<br>**STB R07, 2146** stores a byte from R07 into memory address 2146<br>**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<br>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<br>e.g.<br>**PSH #6** pushes number 6 on top of the stack<br>**PSH R03** pushes the contents of register R03 on top of the stack |
| POP | Pop data from top of hardware stack to register<br>e.g.<br>**POP R05** pops contents of top of stack into register R05<br>**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<br>e.g.<br>**ADD #3, R02** adds number 3 to contents of register R02 and stores the result in register R02.<br>**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<br>e.g.<br>**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)<br>e.g.<br>**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.<br>e.g.<br>**MSF** reserve a space for the return address on program stack<br>**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)<br>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<br>e.g.<br>**CMP #5, R02** compare number 5 with the contents of register R02<br>**CMP R01, R03** compare the contents of registers R01 and R03<br>Note:<br>If R01 = R03 then the status flag **Z** will be set, i.e. the **Z** box is checked.<br>If R01 < R03 then none of the status flags will be set, i.e. none of the status flag boxes are checked.<br>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<br>e.g.<br>**OUT 16, 0** outputs contents of data in location 16 to the console (the second parameter must always be a 0) |