CPU Instruction Set Architecture 1

Introduction

Objectives

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

- Use the simulator to execute basic CPU instructions
- Use direct and indirect addressing modes
- Create iterative loops
- Create sub-routines, sub-routine calls and return from sub-routines
- Compile source code and investigate code generated

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.

Basic Theory

The instruction sets of computer architectures define those low-level architectural components, which include the following

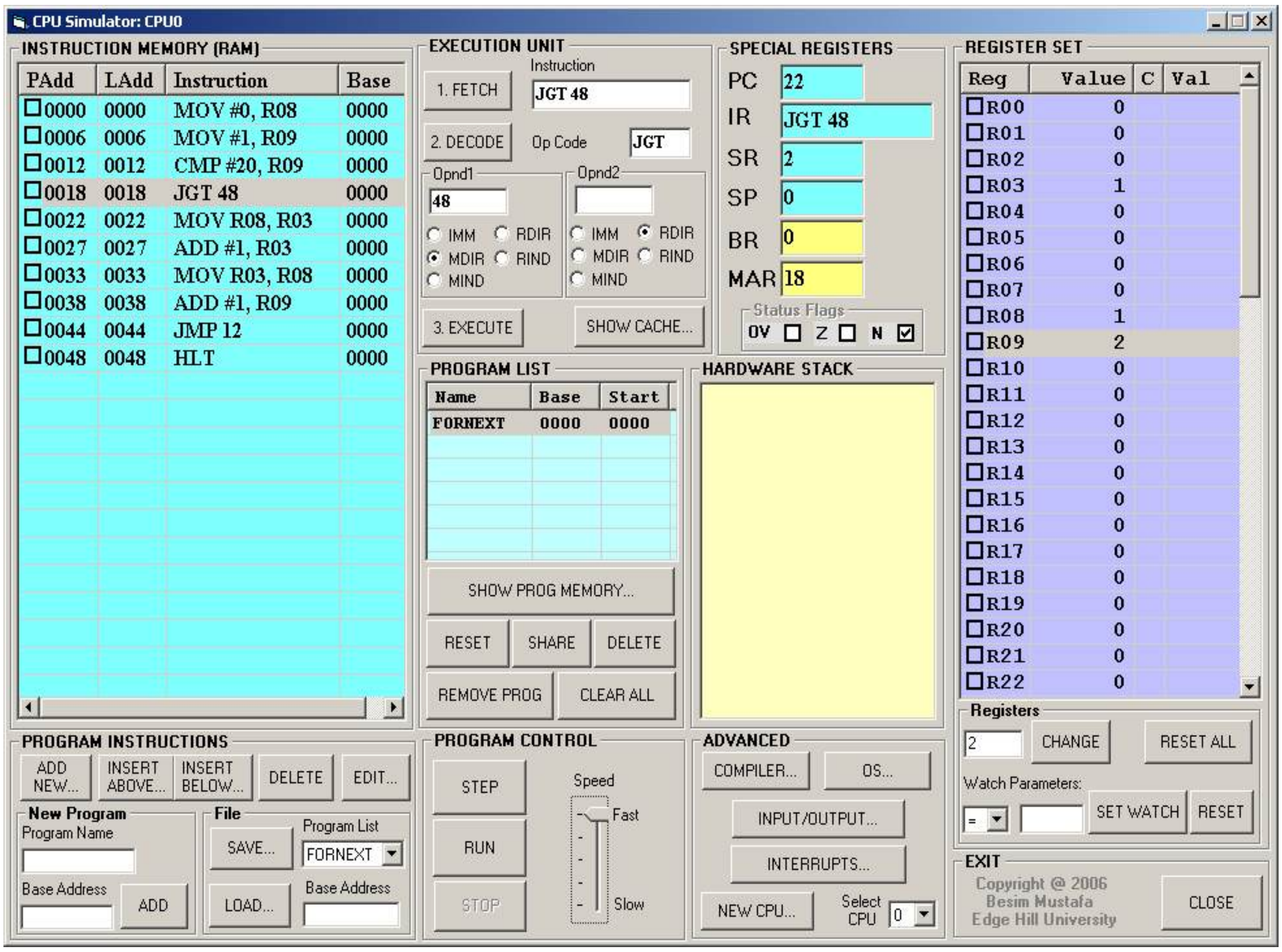
- Processor instructions
- Registers
- Modes of addressing instructions and data
- Interrupts and exceptions

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

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 and is composed of a single main window.



INSTRUCTION MEMORY (RAM)

PAdd	LAdd	Instruction	Base
☐0000	0000	MOV #0, R08	0000
☐0006	0006	MOV #1, R09	0000
☐0012	0012	CMP #20, R09	0000
☐0018	0018	JGT 48	0000
☐0022	0022	MOV R08, R03	0000
☐0027	0027	ADD #1, R03	0000
☐0033	0033	MOV R03, R08	0000
☐0038	0038	ADD #1, R09	0000
☐0044	0044	JMP 12	0000
☐0048	0048	HLT	0000

EXECUTION UNIT

Instruction: JGT 48

Op Code: JGT

Opnd1: 48 — IMM, RDIR, [✓ MDIR], RIND, MIND

Opnd2: — IMM, [✓ RDIR], MDIR, RIND, MIND

SPECIAL REGISTERS

PC	22
IR	JGT 48
SR	2
SP	0
BR	0
MAR	18

Status Flags: OV ☐ Z ☐ N ☑

PROGRAM LIST

Name	Base	Start
FORNEXT	0000	0000

HARDWARE STACK

REGISTER SET

Reg	Value	C	Val
☐R00	0		
☐R01	0		
☐R02	0		
☐R03	1		
☐R04	0		
☐R05	0		
☐R06	0		
☐R07	0		
☐R08	1		
☐R09	2		
☐R10	0		
☐R11	0		
☐R12	0		
☐R13	0		
☐R14	0		
☐R15	0		
☐R16	0		
☐R17	0		
☐R18	0		
☐R19	0		
☐R20	0		
☐R21	0		
☐R22	0		

Registers: 2

Watch Parameters: =

PROGRAM INSTRUCTIONS — New Program: Program Name, Base Address — File: Program List FORNEXT, Base Address

PROGRAM CONTROL

ADVANCED — Select CPU 0

EXIT — Copyright @ 2006 Besim Mustafa Edge Hill University

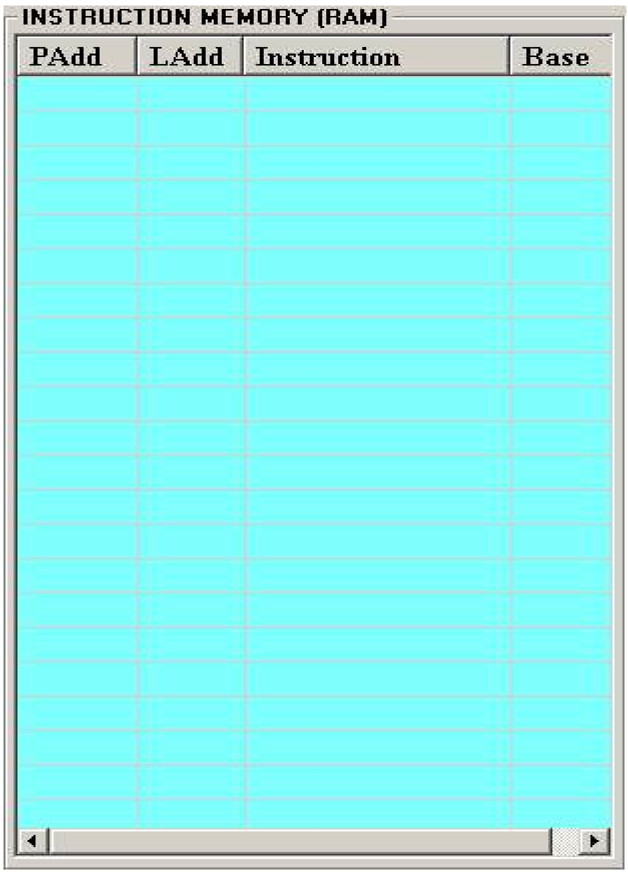
Image 1 - Main simulator window

The main window is composed of several views, which represent different functional parts of the simulated processor. These are

- Instruction memory, i.e. RAM
- Special registers
- Register set
- Hardware stack

The parts of the simulator relevant to this lab are described below.

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.

Each instruction has 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.

Image2 - Instruction memory view

Special registers view

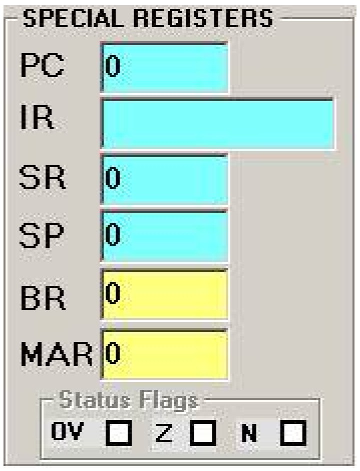
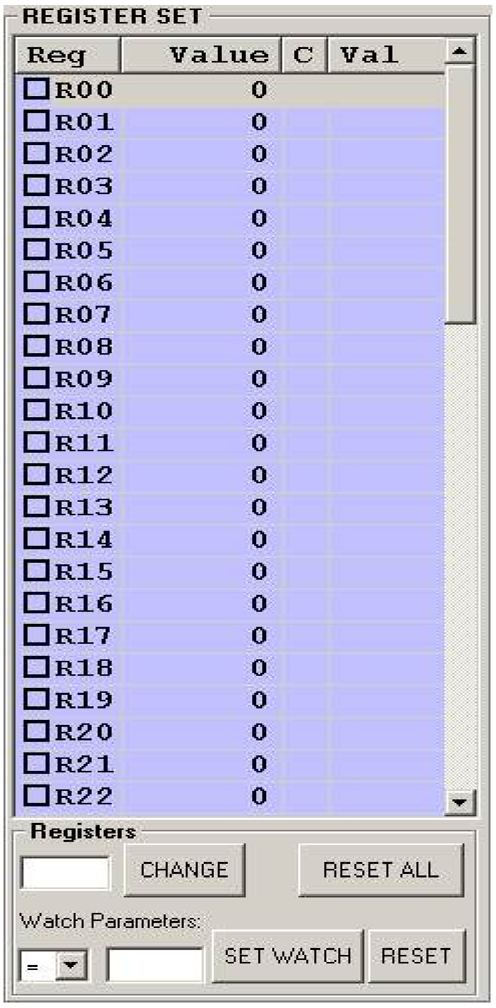


Image 3 - Special registers view

This view presents the set of 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 hardware 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

Register set view



Reg	Value	C	Val
R00	0		
R01	0		
R02	0		
R03	0		
R04	0		
R05	0		
R06	0		
R07	0		
R08	0		
R09	0		
R10	0		
R11	0		
R12	0		
R13	0		
R14	0		
R15	0		
R16	0		
R17	0		
R18	0		
R19	0		
R20	0		
R21	0		
R22	0		

Image 4 - Register set view

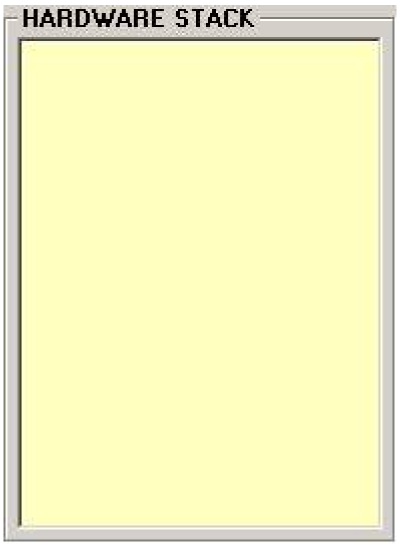
The register set view shows the contents of all the general-purpose registers, which are used to maintain temporary values as the program's instructions are executed.

In this architecture, there are maximum 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 (**Value**) 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.

Hardware stack view



The hardware stack 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.

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

Image 5 - Hardware stack view

Lab Exercises - Investigate and Explore

The lab exercises are a series of exercises, which are attempted by the students under guidelines. The students are encouraged to carry out further investigations on their own in order to form a better understanding of the technology.

First we need to place some instructions in the **Instruction Memory View** (i.e. representing the RAM in the real machine) before executing any instructions. How are instructions placed in the Instruction Memory View? Follow the procedure below for this.

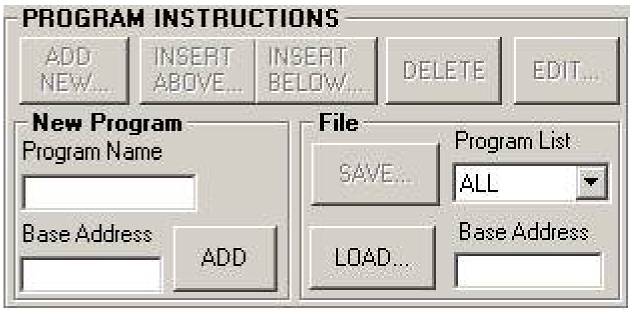
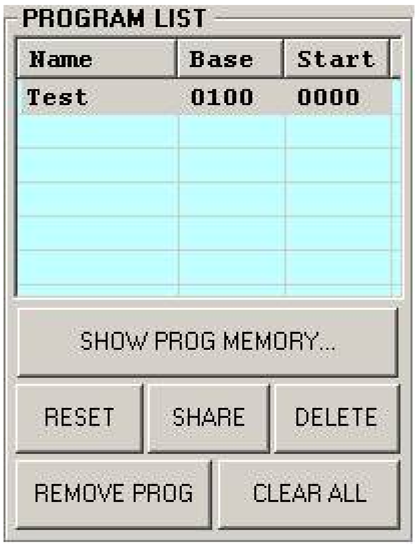


Image 6 - Program Instructions View

In the **Program Instructions View**, 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** shown below. Use the **SAVE… / LOAD...** buttons to save instructions in a file and load the instructions from a file.

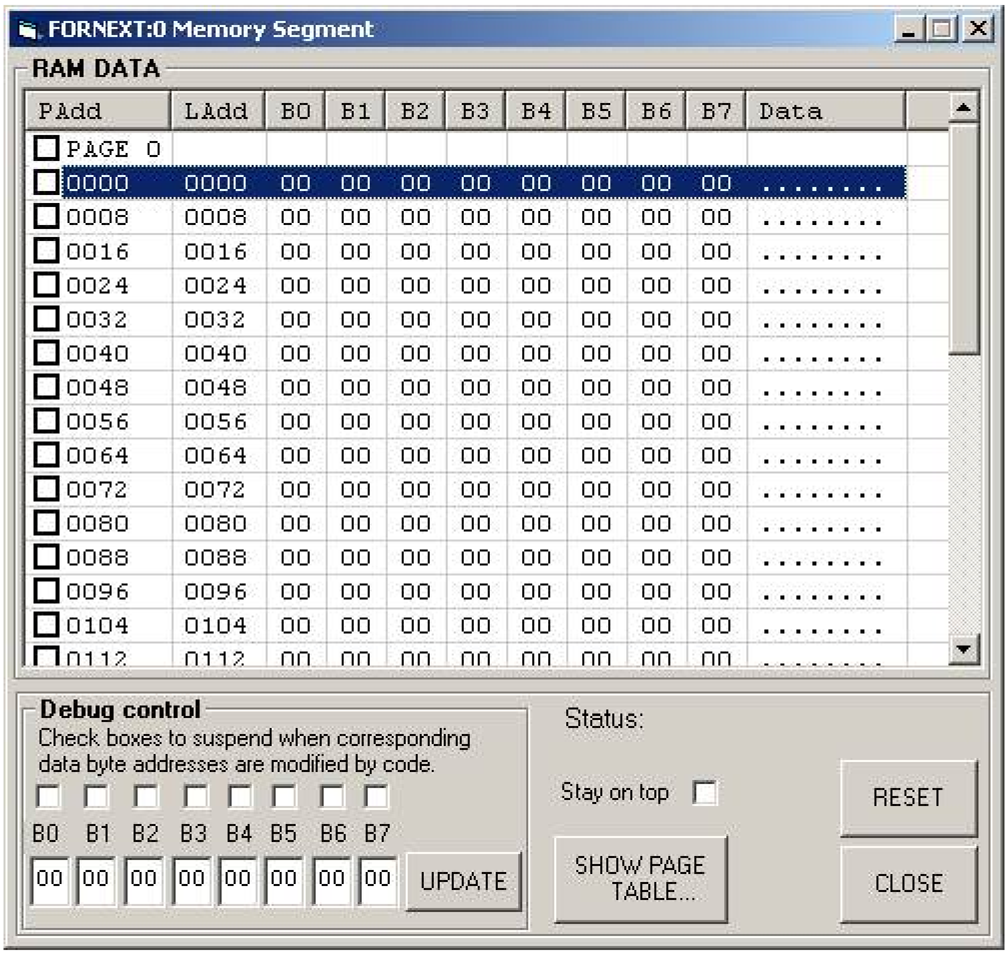


Name	Base	Start
Test	0100	0000

Image 7 - Program List View

Use the **DELETE** button to delete the selected program from the list; use the **CLEAR ALL** button to remove all the programs from the list. Note that when a program is deleted, its instructions are also removed from the **Instruction Memory View** too.

In the following exercises, you'll also need to see the contents of user memory assigned to your program. To do this click on the **SHOW PROG MEMORY…** button (see Image 7 above) in the **PROGRAM LIST** view. The memory contents will be displayed in a separate window as shown below. The addresses are displayed in decimal and the memory data are displayed in hexadecimal formats.



PAdd	LAdd	B0	B1	B2	B3	B4	B5	B6	B7	Data
PAGE 0										
0000	0000	00	00	00	00	00	00	00	00	....
0008	0008	00	00	00	00	00	00	00	00	....
0016	0016	00	00	00	00	00	00	00	00	....
0024	0024	00	00	00	00	00	00	00	00	....
0032	0032	00	00	00	00	00	00	00	00	....
0040	0040	00	00	00	00	00	00	00	00	....
0048	0048	00	00	00	00	00	00	00	00	....
0056	0056	00	00	00	00	00	00	00	00	....
0064	0064	00	00	00	00	00	00	00	00	....
0072	0072	00	00	00	00	00	00	00	00	....
0080	0080	00	00	00	00	00	00	00	00	....
0088	0088	00	00	00	00	00	00	00	00	....
0096	0096	00	00	00	00	00	00	00	00	....
0104	0104	00	00	00	00	00	00	00	00	....
0112	0112	00	00	00	00	00	00	00	00	....

Debug control

Check boxes to suspend when corresponding data byte addresses are modified by code.

B0	B1	B2	B3	B4	B5	B6	B7
00	00	00	00	00	00	00	00

Status:

Stay on top ☐

Image 8 - Program memory page

You are now ready to enter instructions into this view. You do this by clicking on the **ADD NEW…** button. This will display the **Instructions: CPU0** window. Use this window to enter the instructions. Use the appendix provided as a reference to the simulator's instruction set architecture.

Complete the following activities:

1. In the appendix, locate the instruction, which is used to store a byte of data in a memory location.

2. Use it to store number 65 in address location 20 (all numbers are in decimal). This is an example of **direct addressing**.

3. Create an instruction to move number 22 to register R01 and execute it.

4. Create an instruction to store number 51 in address location currently stored in register R01 and execute it. This is an example of **indirect addressing**.

5. Verify that the specified bytes are written to the correct address locations (see Image 8). You should see an **A** and a **3** under the **Data** column.

6. Now, let's create a loop: First set R02 to 0 (zero). Increment R02's value by 1 (one). If R02's value is 5 then exit this loop and stop the program; otherwise continue the loop.

7. Let's plant a short text into memory (we are hacking now!). Click and highlight memory location 0024 (under **PAdd** column). Now enter **'h**, **'e**, **'l**, **'l**, **'o, 0D** (i.e. decimal 13)**, 0A** (i.e. decimal 10) in boxes **B0** to **B6** and click on the **UPDATE** button. The text "**hello**" should now be in memory (starting from address location 24). What do the last two hex bytes 0D0A do?

8. Create a small sub-routine which when called will display the text "**hello**". You may need your tutor's help on this.

9. Modify the above loop (i.e. insert a call to subroutine instruction) to call this subroutine each time the value of R02 is incremented by 1 (one).

10. Verify that when the loop is executed, the text "**hello**" is displayed. To see the text click on the **INPUT/OUTPUT…** button in **ADVANCED** view (see Image 1 above).

11. Observe the contents of the PC register and the hardware stack just before the subroutine call. Observe these again just after the subroutine return instruction is executed. Explain your observations.

12. Go to the compiler screen (click on the **Compiler…** button) and enter the following source code in the **Program Source** frame:

```
program TestSource
    for I = 1 to 8
        N = N + 2
    next
end
```

First make sure you check the **Enable Optimizer** and the **Redundant Code** check-boxes at the bottom left corner of the window. Now compile this code and observe the code generated on the right. Investigate the binary code generated (shown in hex format) against each instruction and try to understand how this is constructed for each instruction. You may need your tutor's help on this.

13.Enter the source below and compile it.

```
program StringTest
    var S string(5)
    S = “Hello”
End
```

When the above source is successfully compiled do the following

On the compiler window click on the **SHOW…** button in the **BINARY CODE** view (near bottom right corner). You should see the **Binary Code for StringTest** window displayed. In this window you should see the binary code generated for this program (it is actually displayed as hex values). Let’s analyse the code generated. Do the following

Click on the **RESET** button. Click on the **NEXT INSTRUCTION** button. You should see a value in the **Address** text box and the opcode of the instruction in the **Opcode** text box. At the same time the relevant part of the instruction will be highlighted in the **Binary Code Data** view. To decode the instruction further, click on the → button. If the instruction has any operand you should now see it in the **Opnd1** text box. At the same time, observe which radio button gets selected. The radio buttons indicate the addressing modes of the operands as they get decoded. By repeatedly clicking on the → button (when enabled) you should see the rest of the instruction decoded. At the end of the instruction the → button will be disabled. To decode the next instruction, you should click on the **NEXT INSTRUCTION** button again (do not click on the **RESET** button unless you wish to start from the beginning again).

Now, analyze the code generated and explain what is happening. To help you understand this better, you can go back to the compiler window, load the code in CPU simulator (use the **LOAD IN MEMORY** button) and step through the code. You may need to look at the memory where data is written to (use the **SHOW PROG MEMORY…** in the CPU simulator window).

Appendix - Simulator Instruction Sub-set

Instruction	**Description and examples of usage**
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 1000, R02** ;loads one byte value from memory location 1000 **LDB @R00, R01** ;memory location is specified in register R00
LDW	Load a word (2 bytes) from memory to register e.g. **LDW 1000, R02** ;loads two-byte value from memory location 1000 **LDW @R00, R01** ;memory location is specified in register R00
STB	Store a byte from register to memory e.g. **STB #2, 1000** ;stores value 2 into memory location 1000 **STB R02, @R01** ;memory location is specified in register R01
STW	Store a word (2 bytes) from register to memory e.g. **STW R04, 1000** ;stores register R04 into memory location 1000 **STW R02, @2000** ;memory location is specified in memory 2000
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

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
JLT	Jump to instruction address if less than (after last comparison) e.g. **JLT 1000** ;jumps to address location 1000 if the previous comparison instruction result indicates that CMP operand 2 is less than operand 1.
JGT	Jump to instruction address if greater than (after last comparison)
JEQ	Jump to instruction address if equal (after last comparison) e.g. **JEQ 200** ;jumps to address location 200 if the previous comparison instruction result indicates that the two CMP operands are equal.
JNE	Jump to instruction address if not equal (after last comparison)
CAL	Jump to subroutine address e.g. To call a subroutine starting at address location 1000 use the following sequence of instructions **MSF** ;always needed just before the following instruction **CAL 1000** ;will cause a jump to address location 1000
RET	Return from subroutine e.g. The last instruction in a subroutine must always be the following instruction

	RET ;will jump to the instruction after the last CAL instruction.
SWI	Software interrupt (used to request OS help)
HLT	Halt simulation. This must be the last instruction. e.g. **HLT** ;stops the simulation run (<u>not</u> the simulator itself)
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 R03 = R01 then the status flag **Z** will be set If R03 > R01 then non of the status flags will be set If R03 < R01 then the status flag **N** will be set
Input, output instructions	
IN	Get input data (if available) from an external IO device
OUT	Output data to an external IO device e.g. to display a string starting in memory address 120 (decimal) on console device do the following **OUT 120, 0** ;the string is in address location 120 (direct addressing) **OUT @R02, 0** ;register R02 has number 120 (indirect addressing)