

66070113 ปณวัฒน์ ธรรมเจริญ

Instruction Set Architecture

Introduction

Objectives

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

Part A:

- Explain how common program variables are stored
- Distinguish between different types of high-level program statements
- Understand low-level code corresponding to program statements
- Explain how program subroutines work
- Use the simulator to create user interrupts

Part B:

- 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

Part A:

High-level language (HLL) programs are made of variables holding data values and multiple program statements as algorithms. These statements often control the flow of program execution under certain conditions. Calls to subroutines and interrupts all change the sequential flow of a program execution without which feature programs would not do any useful work.

Part B:

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

Part A:

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 multiple windows.

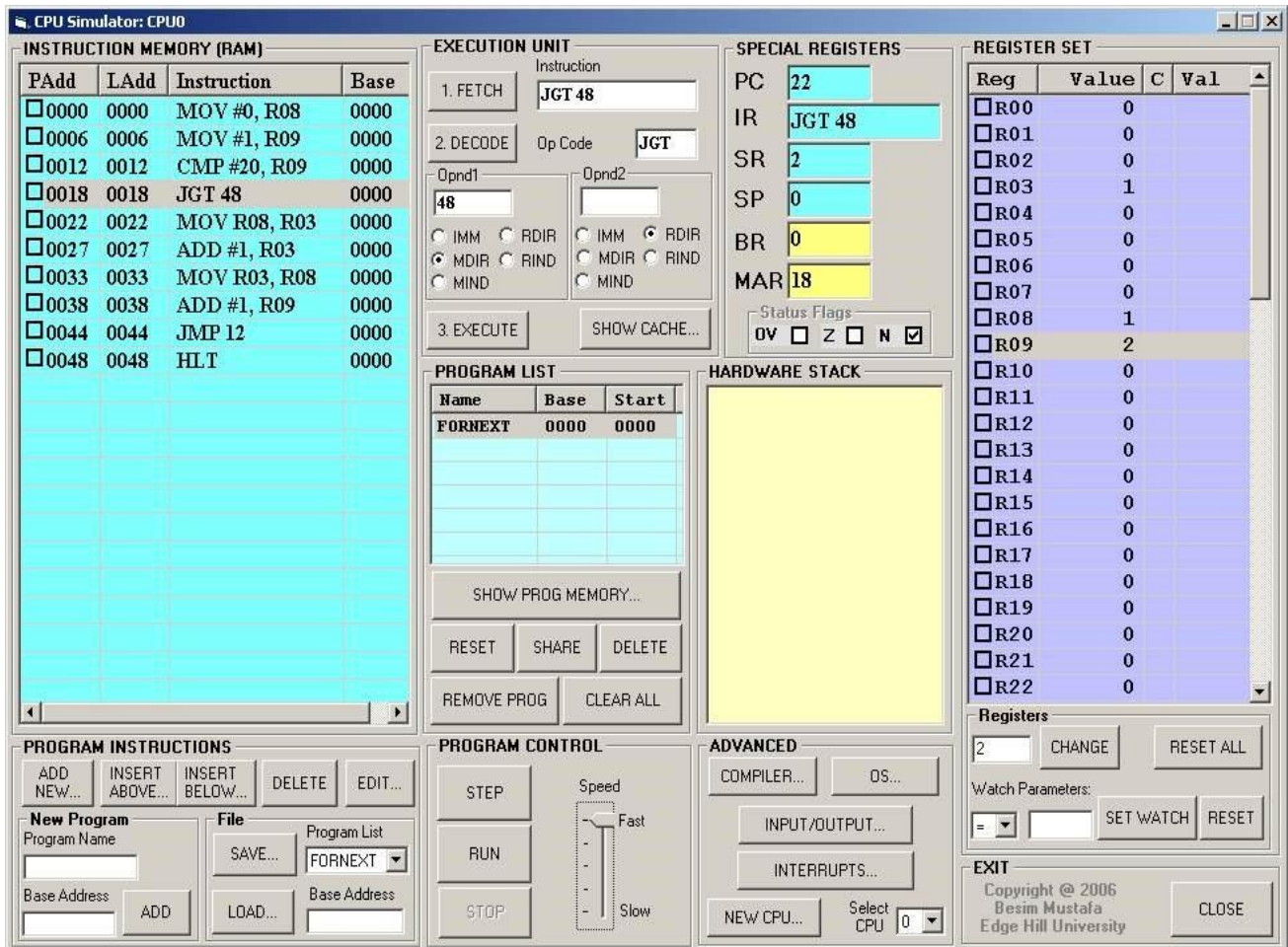
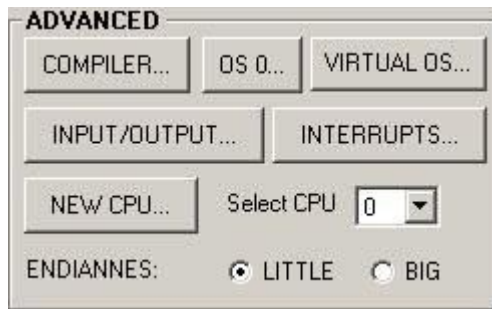


Figure 1 - Main simulator window

The main window shown in Figure 1 is composed of several sub-views, which represent different functional parts of the simulated processor. For this lab session we are interested only in the compiler part of the simulator.



In order to access the compiler, click on the **COMPILER...** button as shown in Figure 2 on the right. The compiler window shown in Figure 3 below will show.

Figure 2 - Advanced functions

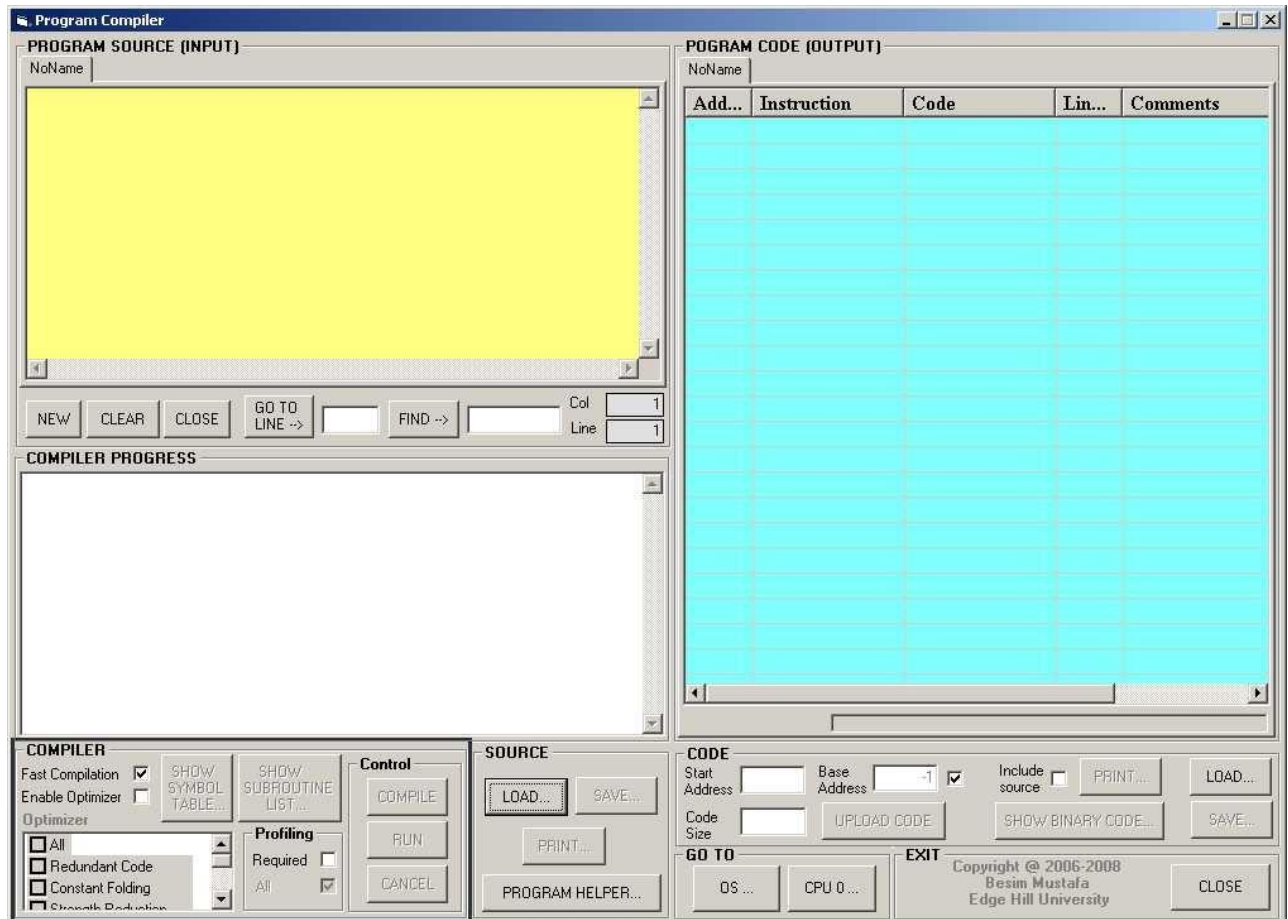


Figure 3 - The main compiler window

In the compiler window there are three main sub-windows

- ③ **Program Source** - all high-level source statements appear here.
- ③ **Compiler Progress** - information on the progress of a compilation appear here.
- ③ **Program Code** - assembly code generated by the compiler appear here.

Part B:

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.

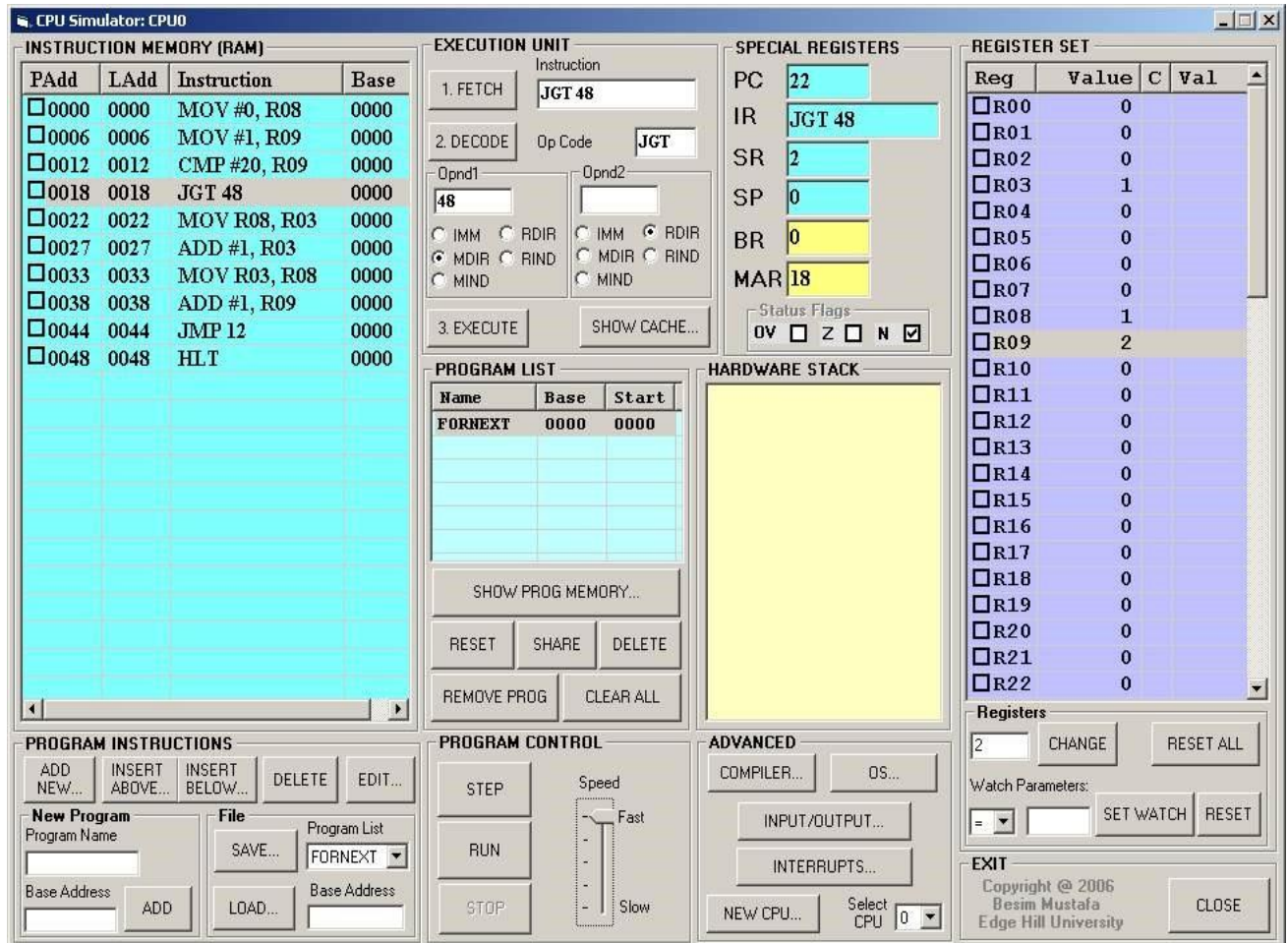


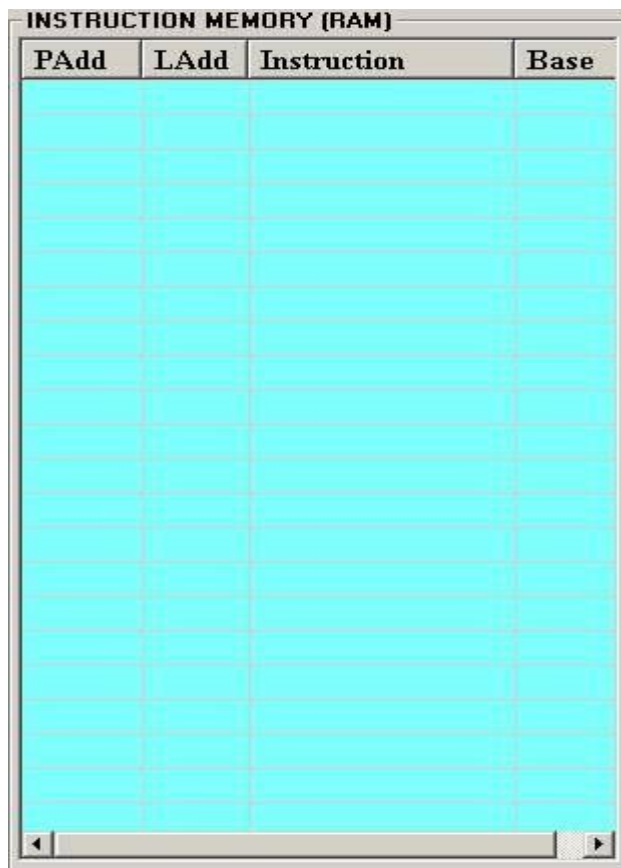
Figure 4 - Main simulator window

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

- ③ Instruction memory, i.e. RAM
- ③ Special registers
- ③ Register set
- ③ Hardware stack

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

Instruction memory view



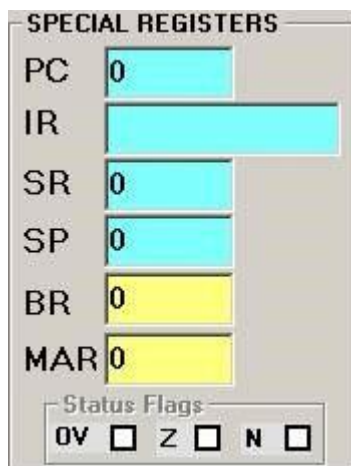
PAdd	LAdd	Instruction	Base
------	------	-------------	------

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.

Figure 5 - Instruction memory view

Special registers view



SPECIAL REGISTERS	
PC	0
IR	
SR	0
SP	0
BR	0
MAR	0

Status Flags

OV ☐ Z ☐ N ☐

Figure 6 - Special registers view

This view presents the set of registers, which have predefined specialist functions:

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

IR: Instruction Register contains the instruction currently being executed.

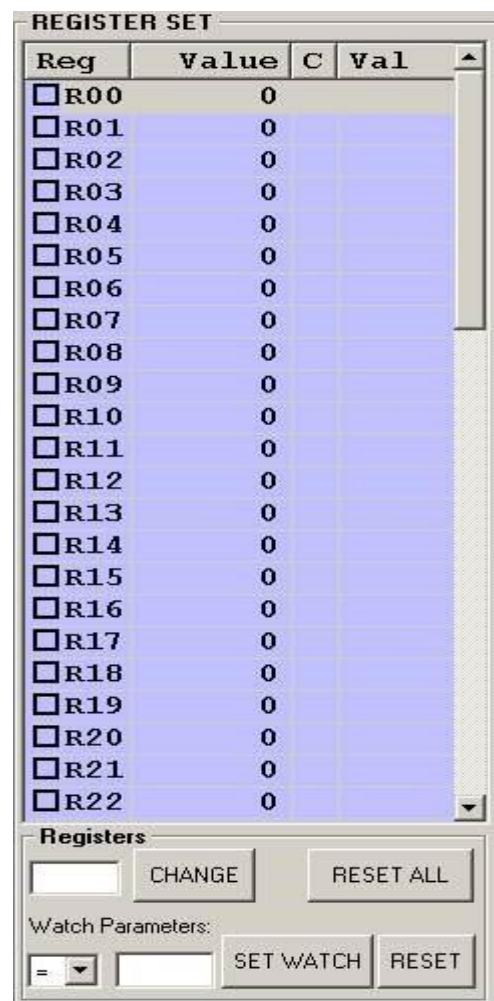
SR: Status Register contains information pertaining to the result of the last executed instruction.

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



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.

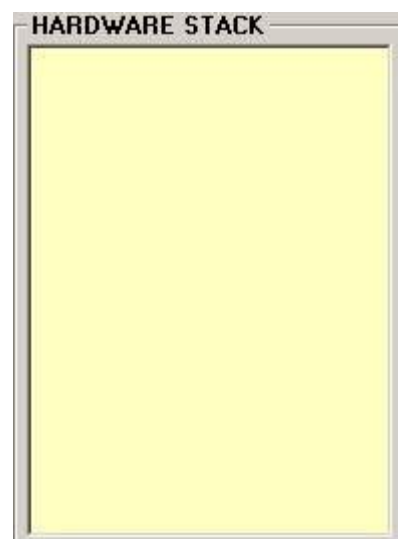
In this architecture, there are maximum 64 registers. These registers are often used to hold values of a program's variables as defined in high-level languages.

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.

Figure 7 - Register set view

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.

Figure 8 - Hardware stack view

Lab Exercises - Investigate and Explore

The lab exercises are a series of experiments, which are attempted by the students under guidelines. The students are expected to carry out further investigations on their own in order to form a better understanding of the technology. Now, have a go at the following activities:

Part A:

1. Enter the following source code and compile it.

```
program Test1
    var IntVar integer
    var BoolVar boolean
    var StrVar1 string (5)
    var StrVar2 string(20)

    IntVar = 6
    BoolVar = true
    StrVar1 = "Hello"
    StrVar2 = "And again"
end
```

Now click on the **SYMBOL TABLE...** button. The **Symbol Table** window will show. Observe the kind of information kept in the symbol table. Make a note of the **type**, **size** and **address** fields for each of the entries in the table.

Next, load the compiled program in memory. In the CPU simulator window click on the **SHOW PROG MEMORY...** button. The contents of the program data memory will show. Make sure it stays on top. Then run the program at maximum speed. Observe the contents of the memory paying attention to the address locations you noted before.

Answers and Results:

The screenshot displays the CPU simulator interface. On the left, the **Symbol Table** window is open, showing a list of variables and their addresses. On the right, the **DATA MEMORY** window is open, showing the memory contents at various addresses.

PAdd	LAdd	Instruction	Base	T
0000	0000	MOV #6, R02	0000	0
0006	0006	STW R02, 6	0000	0
0012	0012	MOV #1, R02	0000	0
0018	0018	STW R02, 9	0000	0
0024	0024	MOV #47, R02	0000	0
0030	0030	MOV #6, R05	0000	0
0036	0036	MOV R02, R03	0000	0
0041	0041	LDB @R03, R04	0000	0
0046	0046	ADD #1, R03	0000	0
0052	0052	STB R04, @R05	0000	0
0057	0057	ADD #1, R05	0000	0
0063	0063	CMP #0, R04	0000	0
0069	0069	JNE 41	0000	0
0073	0073	MOV #58, R02	0000	0
0079	0079	MOV #13, R05	0000	0
0085	0085	MOV R02, R03	0000	0
0090	0090	LDB @R03, R04	0000	0
0095	0095	ADD #1, R03	0000	0
0101	0101	STB R04, @R05	0000	0
0106	0106	ADD #1, R05	0000	0
0112	0112	CMP #0, R04	0000	0
0118	0118	JNE 90	0000	0
0122	0122	HLT	0000	0

The **DATA MEMORY** window shows the memory contents at various addresses. The memory is organized into pages, and the current page is 0. The memory contents are as follows:

PAGE 0	LAdd	B0	B1	B2	B3	B4	B5	B6	B7	Data
0000	02	00	00	00	00	00	03	48	H
0008	65	6C	6C	6F	00	03	41	6E		ello...An
0016	64	20	61	67	61	68	6E	00		d again.
0024	00	00	00	00	00	00	00	00	
0032	00	00	00	00	00	00	00	00	
0040	00	00	00	00	00	00	00	00	
0048	48	65	6C	6C	6F	00	00	00		Hello...
0056	00	00	03	41	6E	64	20	61		...And a
0064	67	61	69	6E	00	00	00	00		gain....
0072	00	00	00	00	00	00	00	00	
0080	00	00	00	00	00	00	00	00	
0088	00	00	00	00	00	00	00	00	
0096	00	00	00	00	00	00	00	00	
0104	00	00	00	00	00	00	00	00	
0112	00	00	00	00	00	00	00	00	

The **Initialise Data** section shows the following options:

- ☐ Integer Value:
- ☒ Boolean Value:
- ☐ String Value:

The **Debug control** section shows the following options:

- ☐ Check boxes to suspend when corresponding data byte addresses are modified by code.
- ☐ B0 ☐ B1 ☐ B2 ☐ B3 ☐ B4 ☐ B5 ☐ B6 ☐ B7
- ☐ 64 ☐ 20 ☐ 61 ☐ 67 ☐ 61 ☐ 69 ☐ 6E ☐ 00

The **Status** section shows the following options:

- ☐ Stay on top
-
- Pages: Size:
-

2. Enter the following source statements

```
Program Test2
  n = 0
  i = n + 1
  p = i * 3
  writeln (" n=", n, " i=", i, " p=", p)
end
```

Compile the above program. Now observe the code generated in the **PROGRAM CODE** window. You don't need to analyse it in detail. However, count the number of jump instructions (i.e. those that start with a letter 'J') and note this down. Can you tell what kind of program statements this program is using?

Answers and Results:

ไม่มีการ jump เลย โปรแกรมนี้เกี่ยวกับการคำนวณเลขโดยเริ่มจาก n=0
ต่อด้วย i = n+1 สุดท้าย p = i * 3 แล้วป้อนค่าทั้ง 3 ตัวออกมา

3. Enter the following source statements

```
Program Test3
  n = 0
  if n < 5 then
    p = p + 1
  end if
end
```

Compile the above program. Now observe the code generated. How many jump instructions are there? What do you think is the purpose of the jump instruction in this code? What kind of a statement is an 'if' statement?

Answers and Results:

มีการ jump 1 ครั้ง มีไว้เพื่อเช็คค่าว่ามีค่ามากกว่า หรือไม่ ถ้าใช่จะกระโดดไปที่ Ladd 52
ใช้ JGE: Jumps if greater than or equal ทำงานเหมือน IF เป็นคำสั่งเงื่อนไข

4. Enter the following source statements

```
program Test4
  p = 1
  for n = 1 to 10
    p = p * 2
  next
end
```

Compile the above program. Now observe the code generated. How many jump instructions are there? What do you think is the purpose of each of the jump instructions in this code? What kind of a statement is a 'for' statement? Can you think of another statement of this kind (you can give an example from any programming language you are familiar with)?

Answers and Results:

มีการ jump 2 ครั้ง คือ

JGT: Jumps if greater than ใช้เพื่อตรวจสอบเงื่อนไขว่าการเปรียบเทียบมีค่ามากกว่า
จะกระโดดไปที่ Ladd 68

JMP: Jumps unconditionally ใช้เพื่อกระโดดไปที่ Ladd 28 คำสั่งนี้เปรียบเสมือน for
ตัวอย่างภาษาอื่น for i in range(10):

5. Enter the following source statements

```
Program Test5
  sub One
    writeln("I am sub One")
  end sub

  sub Two
    call One
    writeln("I am sub Two")
  end sub

  call Two
End
```

Compile the above program. Next, load the compiled program in memory. In the CPU simulator window click on the **SHOW PROG MEMORY...** button. Click on the **SHOW PIPELINE...** button and check the checkbox labelled **No instruction pipeline**. Close the window. In the CPU simulator window do the following

Click on the **RESET** button in **PROGRAM LIST** frame. Now you'll manually execute this program instruction by instruction. To do this double-click the currently highlighted instruction. So, you'll start with the **MSF** instruction, and then do the **CAL** instruction, etc. As you execute the program in this manner, make the following observations: make a note of the **PROGRAM STACK** frame contents after executing a **CAL** instruction or a **RET** instruction. Keep executing instructions until you reach the **HLT** instruction.

Can you explain what is happening each time a **CAL** or a **RET** instruction is executed and how they affect the **PROGRAM STACK** contents.

Answers and Results:

จาก cal 20 คือการเรียกใช้คำสั่งตั้งแต่Ladd แล้วทำการpush ค่าLadd ของคำสั่งต่อจากนั้นลงในStack
ต่อมามีการ cal 0 จึงทำการไปใช้คำสั่งLadd 0 แล้วpush ค่าLaddของคำสั่งถัดไปลงStack
เมื่อถึงคำสั่ง RET จะทำการ popค่าในStackออกมาแล้วไปเรียกคำสั่งที่ Laddค่านั้น

6. Enter the following source statements

```
program Test6
  sub Any
    n = 0
  end sub

  sub MeToo intr 5
    writeln("Me too, me too!")
  end sub

do
loop
end
```

Compile the above program. Look at the code generated. What address does subroutine "MeToo" start at? Make a note of this number. Next, load the compiled program in memory. In the CPU simulator window click on the **INTERRUPTS...** button. The **Interrupts** window will show. Make a note of the interrupt number against which a number appears in the corresponding box. Do these numbers mean anything to you? Explain.

Make sure the **Interrupts** window stays on top. Click on the **INPUT/OUTPUT...** button and make sure the **Console** window also stays on top. Now slide the speed of the CPU simulator to the fastest speed and run the program. Make a note of what you are observing. What is the main purpose of the "**do**" loop statement in this program?

Next, click on the **TRIGGER** button in the **Interrupts** window while at the same time you keep your eye on the **Console** window. Make a note of what you are observing.

Slow down the CPU simulation (e.g. a little above half way on the sliding scale). Trigger the interrupt and observe the **PROGRAM STACK** contents. You can click on the **STOP** button as soon as you see numbers appearing on this stack so that you have time to look at its contents. What do you observe?

There are two main types of interrupts: vectored and polled. Which type is the above interrupt? Explain.

Answers and Results:

intr 5 คือ การเก็บค่าการเรียกใช้งาน ไว้ในinterrupts เมื่อกดtrigger โปรแกรมที่ทำงานอยู่จะไปเรียกใช้ค่าสิ่งนั้น

โปรแกรมจะทำงานจะวนลูปที่เดิม แต่เมื่อมีการกด trigger intr 5 จะเรียกใช้ subroutine Metooทันที เป็นVectored Interrupts

Part B:

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.

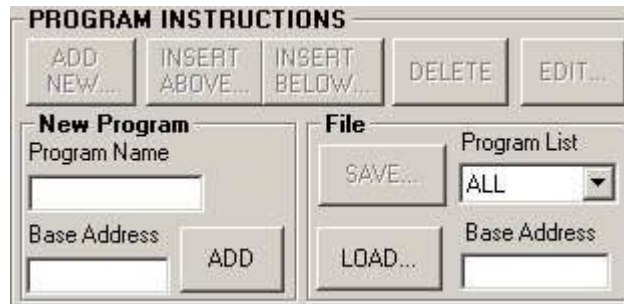
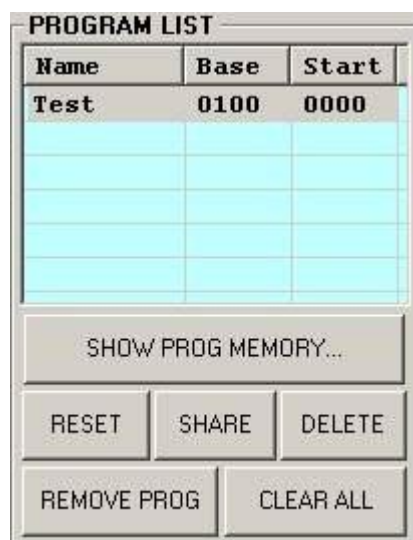


Figure 9 - Program Instructions View

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.



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.

Figure 10 - Program List View

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 Figure 10 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.

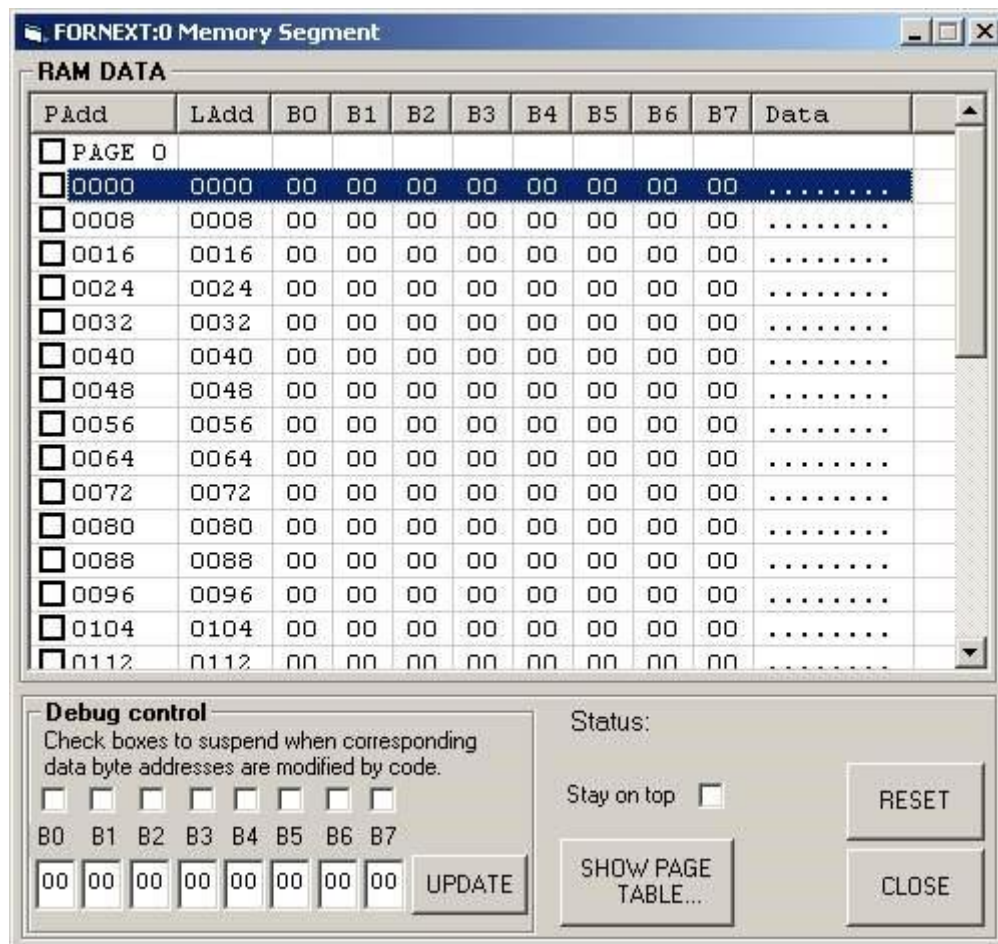


Figure 11 - Program memory page

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**.

Results:

PAdd	LAdd	Instruction	Base	T	
<input checked="" type="checkbox"/> 0000	0000	STB #65, 20	0000	0	

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

Results:

PAdd	LAdd	Instruction	Base	T	
<input type="checkbox"/> 0000	0000	STB #65, 20	0000	0	
<input checked="" type="checkbox"/> 0007	0007	MOV #22, R01	0000	0	

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**.

Results:

PAdd	LAdd	Instruction	Base	T	
<input type="checkbox"/> 0000	0000	STB #65, 20	0000	0	
<input type="checkbox"/> 0007	0007	MOV #22, R01	0000	0	
<input checked="" type="checkbox"/> 0013	0013	STB #51, @R01	0000	0	

5. Verify that the specified bytes are written to the correct address locations (see Figure 11). You should see an **A** and a **3** under the **Data** column.

Results:

PAdd	LAdd	B0	B1	B2	B3	B4	B5	B6	B7	Data
<input type="checkbox"/> 0000		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0008		00	00	00	00	00	00	00	00
<input checked="" type="checkbox"/> 0016		00	00	00	00	41	00	33	00A.3.
<input type="checkbox"/> 0024		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0032		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0040		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0048		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0056		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0064		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0072		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0080		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0088		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0096		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0104		00	00	00	00	00	00	00	00
<input type="checkbox"/> 0112		00	00	00	00	00	00	00	00

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.

Results:

<input type="checkbox"/> 0019	0019	MOV #0, R02
<input type="checkbox"/> 0025	0025	CMP #5, R02
<input type="checkbox"/> 0031	0031	JEQ 45
<input type="checkbox"/> 0035	0035	ADD #1, R02
<input type="checkbox"/> 0041	0041	JMP 25
<input checked="" type="checkbox"/> 0045	0045	HLT

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?

Results:

คือการขึ้นบรรทัดใหม่

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

Results:

<input type="checkbox"/>	0026	0026	MOV #24, R12	0000	0	
<input type="checkbox"/>	0032	0032	OUT @R12, 0	0000	4	
<input type="checkbox"/>	0038	0038	RET	0000	2	

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).

Results:

	PAdd	LAdd	Instruction	Base	T	
<input type="checkbox"/>	0000	0000	CMP #5, R02	0000	3	
<input type="checkbox"/>	0006	0006	JEQ 25	0000	2	
<input type="checkbox"/>	0010	0010	MSF	0000	2	
<input type="checkbox"/>	0011	0011	CAL 26	0000	2	
<input type="checkbox"/>	0015	0015	ADD #1, R02	0000	1	
<input type="checkbox"/>	0021	0021	JMP 0	0000	2	
<input checked="" type="checkbox"/>	0025	0025	HLT	0000	2	
<input type="checkbox"/>	0026	0026	MOV #24, R12	0000	0	
<input type="checkbox"/>	0032	0032	OUT @R12, 0	0000	4	
<input type="checkbox"/>	0038	0038	RET	0000	2	

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 Figure 4 above).

Results:



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.

Results and Answers:

ก่อนจะเรียก subroutine จะมีการสร้างพื้นที่ใน Stack (MSF) หลังจาก call จะนำค่าคำสั่ง
Ladd คำสั่งถัดไปใส่ใน Stack เมื่อจบการทำงานจะ pop ออกมาทำคำสั่งถัดไป

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.

Results:

The screenshot shows a compiler window with three main panes. The top-left pane displays the source code for a program named 'TestSource'. The bottom-left pane shows the 'COMPILER PROGRESS' with messages indicating the successful compilation of the source code. The right pane displays the generated binary code in a table format.




LAdd	CPU Instruction	Binary Code	Line	Comments
**** CODE:				
**** MAIN PROG...				
0000	MOV #1, R03	000000010103	0002	Copy the value of 1 to _STempReg...
0006	MOV R03, R01	0001030101	0002	Copy the value of _STempReg145 t...
0011	MOV #5, R03	000000050103	0002	Copy the value of 5 to _STempReg...
0017	CMP R03, R01	3001030101	0002	Compare 1 with _STempReg165
0022	JGT 57	20020039	0002	Jump to code at address 57 if status ...
0026	MOV R02, R04	0001020104	0003	Copy the value of N to _STempReg...
0031	ADD #2, R04	110000020104	0003	Add: _STempReg175 = _STempReg...
0037	MOV R04, R05	0001040105	0003	Copy the value of _STempReg175 t...
0042	MOV R05, R02	0001050102	0003	Copy the value of _STempReg185 t...
0047	ADD #1, R01	110000010101	0004	Add: I = I + 1
0053	JMP 17	1D020011	0004	Jump to code at address 17 (UNCO...
0057	HLT	2F	0005	Stop simulation
**** DATA:				

13. Enter the source below and compile it.

```

program StringTest
    var S string(5)
    S = "Hello"
End

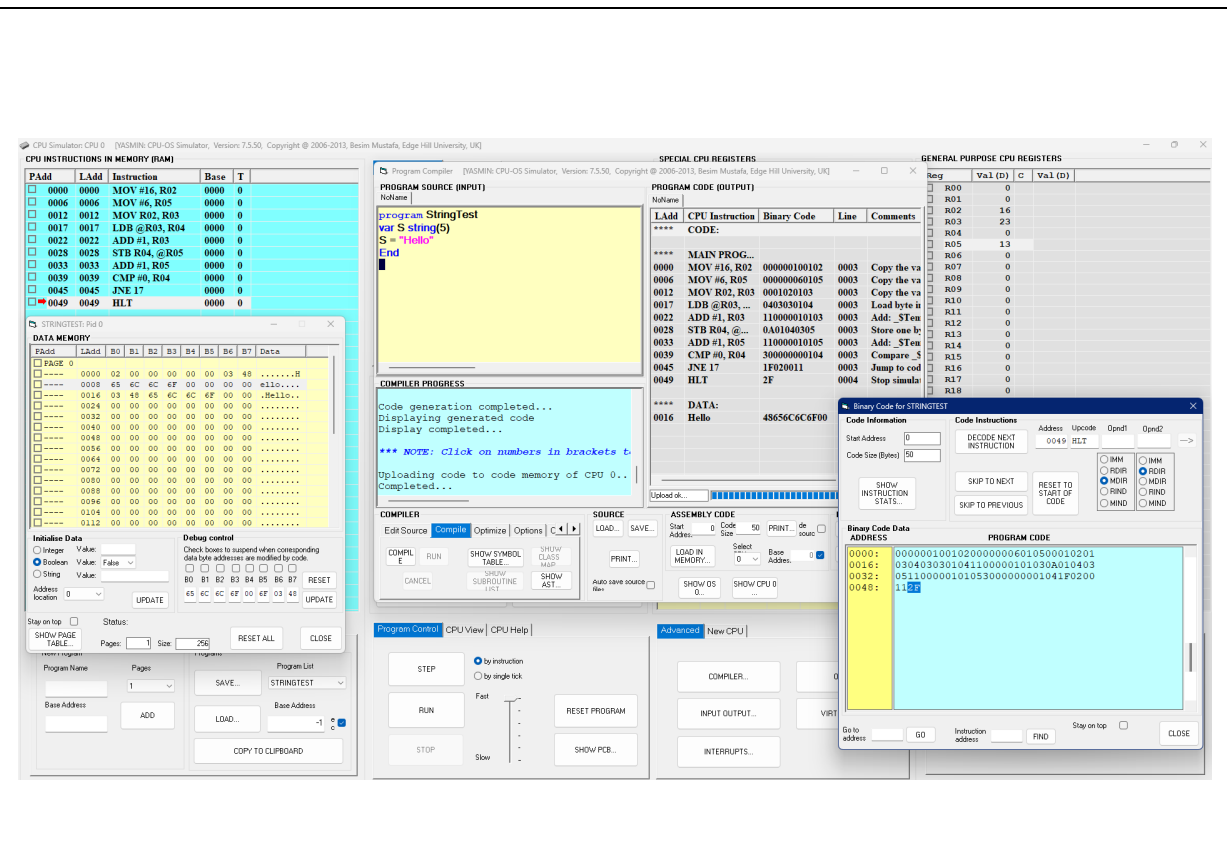
```

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).

Answers:



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

Appendix – CPU 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	<p>Add number to register; add register to register e.g.</p> <p>ADD #3, R02 ;adds number 3 to contents of register R02 and stores the result in register R02.</p> <p>ADD R00, R01 ;adds contents of register R00 to contents of register R01 and stores the result in register R01.</p>
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	<p>Jump to instruction address unconditionally e.g.</p> <p>JMP 100 ;unconditionally jumps to address location 100</p>
JLT	<p>Jump to instruction address if less than (after last comparison) e.g.</p> <p>JLT 1000 ;jumps to address location 1000 if the previous comparison instruction result indicates that CMP operand 2 is less than operand 1.</p>
JGT	Jump to instruction address if greater than (after last comparison)
JEQ	<p>Jump to instruction address if equal (after last comparison) e.g.</p> <p>JEQ 200 ;jumps to address location 200 if the previous comparison instruction result indicates that the two CMP operands are equal.</p>
JNE	Jump to instruction address if not equal (after last comparison)
CAL	<p>Jump to subroutine address</p> <p>e.g. To call a subroutine starting at address location 1000 use the following sequence of instructions</p> <p>MSF ;always needed just before the following instruction</p> <p>CAL 1000 ;will cause a jump to address location 1000</p>

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 none 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)
