# 66070286 ปนัสยา บุญประกอบ

# 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.

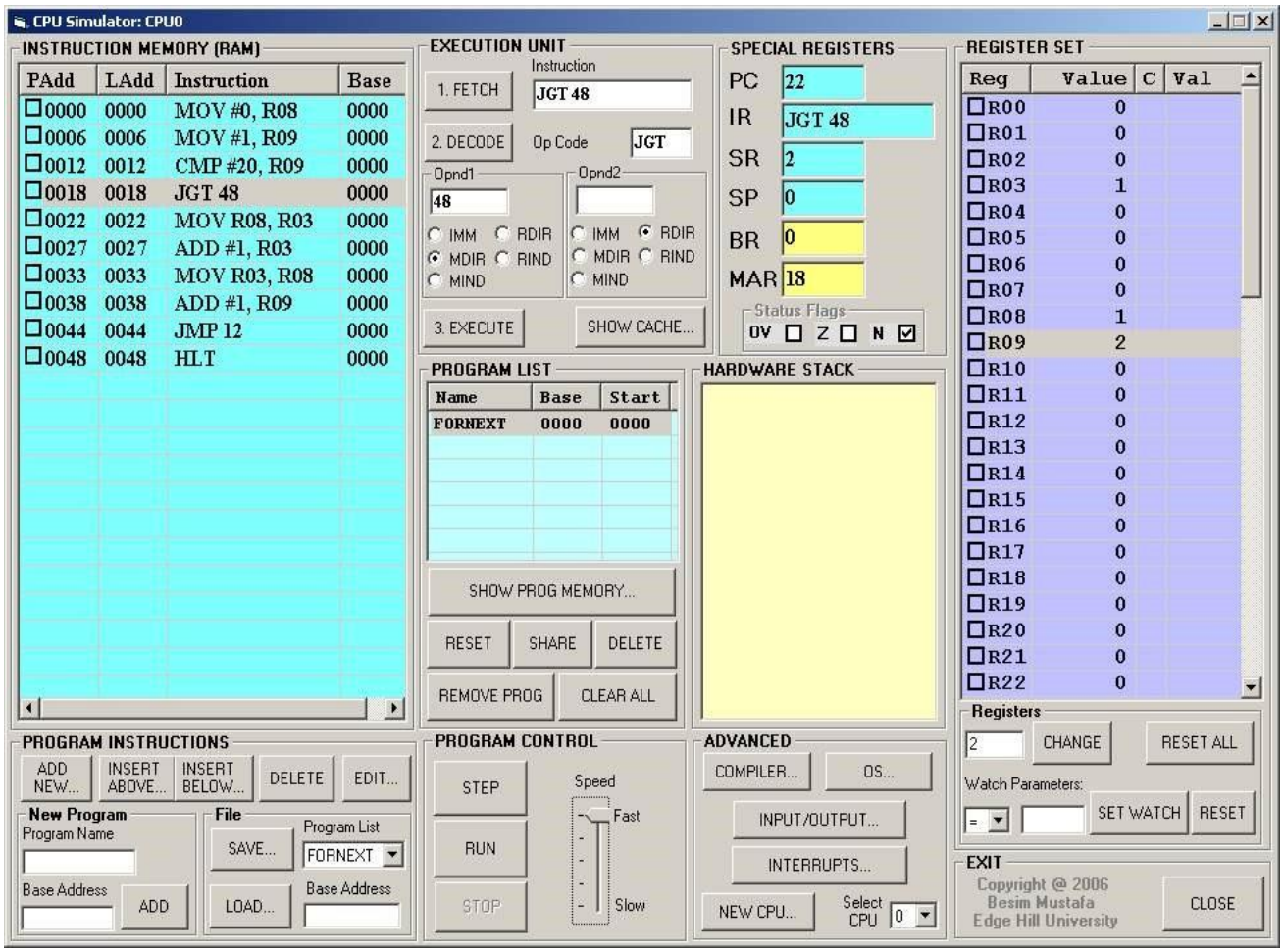
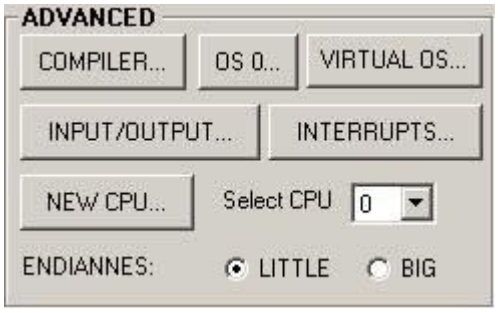



**Figure 1 - Main simulator window**

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**

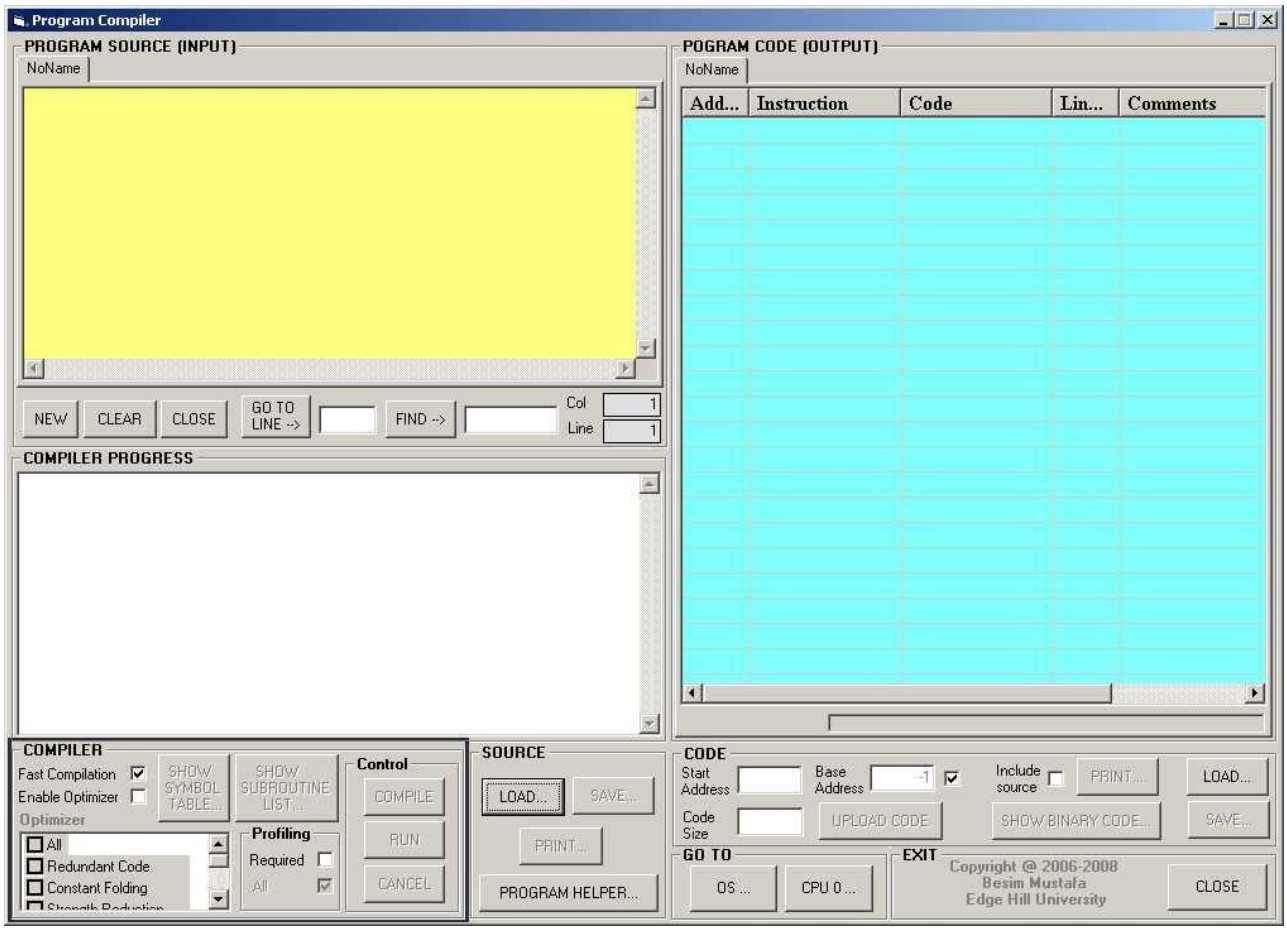



**Figure 3 - The main compiler window**

In the compiler window there are three main sub-windows

- **Program Source** - all high-level source statements appear here.
- **Compiler Progress** - information on the progress of a compilation appear here.
- **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.

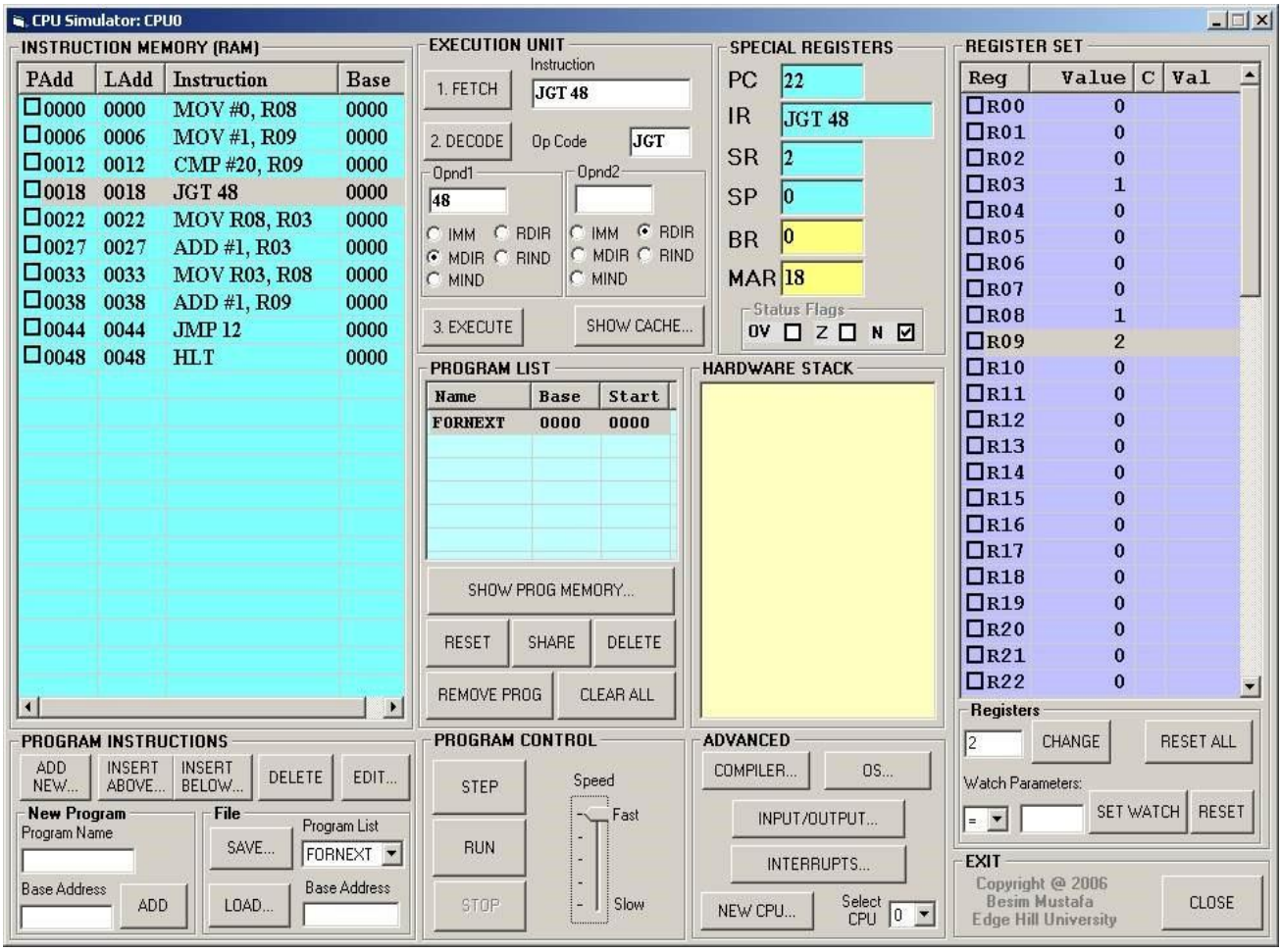


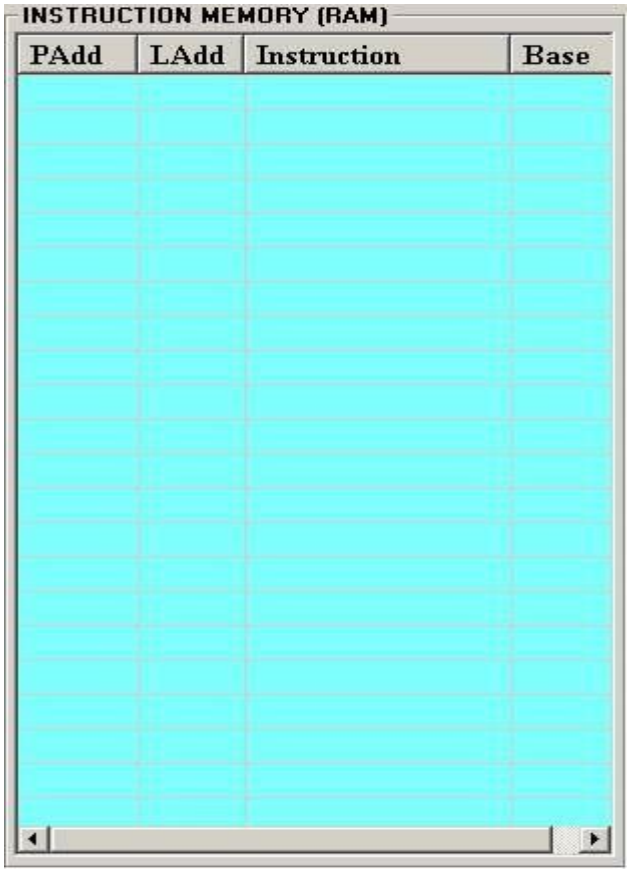

**Figure 4 - Main simulator window**

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




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

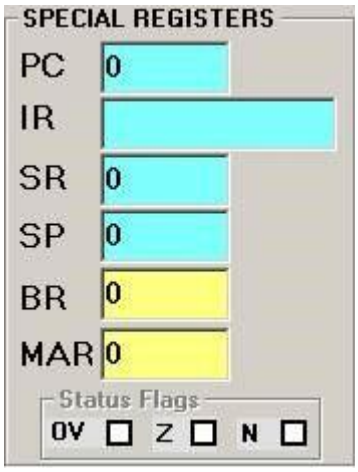
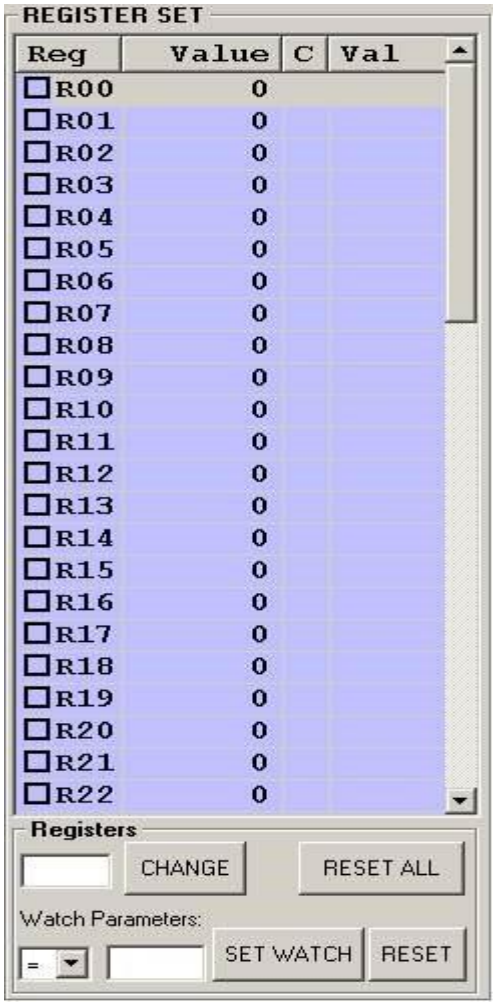



**Figure 6 - Special registers view**

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.

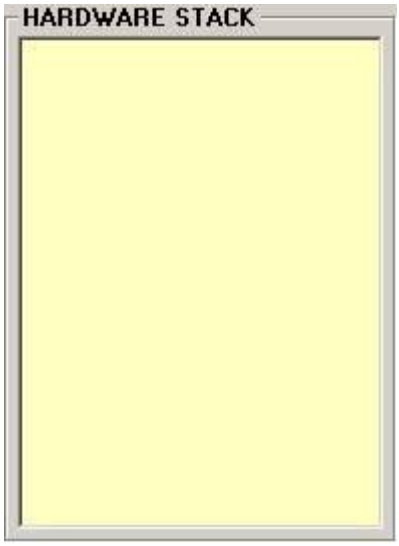
In this architecture, there are maximum 64 registers. These registers are often used to hold values of a program's variables as defined in high-level languages.

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.

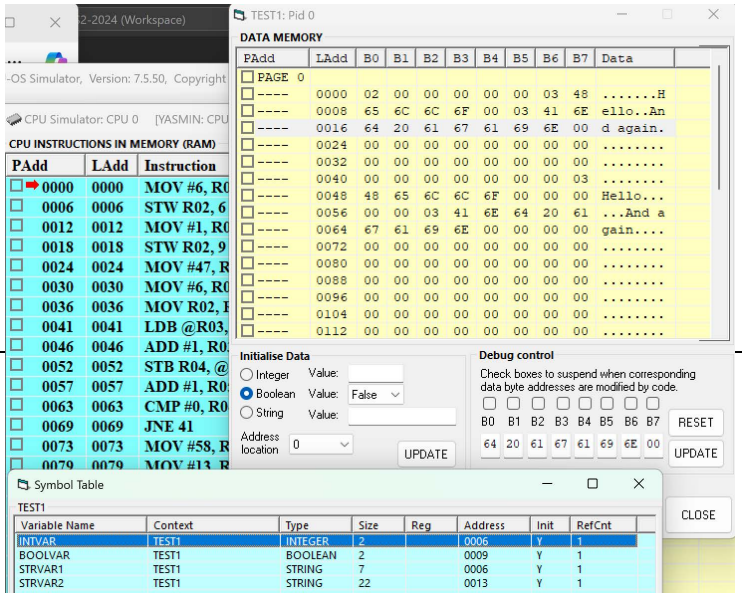
```
program Test1
     var IntVar integer
     var BoolVar boolean
     var StrVar1 string (5)
     var StrVar2 string(20)

     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:



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:

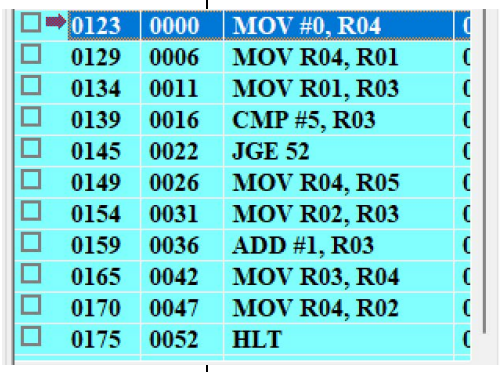
โปรแกรมนี้ไม่มีการ jump แต่มีการคำนวณเลข
โดยบรรทัดแรกคือ n=0
บรรทัดที่ 2 คือ i=n+1
บรรทัดที่ 3 คือ p=i*3
บรรสุดท้ายคือการแสดงผลค่าของ n, i และ p

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:



| | | | |
|---|---|---|---|
| 0123 | 0000 | MOV #0, R04 | |
| 0129 | 0006 | MOV R04, R01 | |
| 0134 | 0011 | MOV R01, R03 | |
| 0139 | 0016 | CMP #5, R03 | |
| 0145 | 0022 | JGE 52 | |
| 0149 | 0026 | MOV R04, R05 | |
| 0154 | 0031 | MOV R02, R03 | |
| 0159 | 0036 | ADD #1, R03 | |
| 0165 | 0042 | MOV R03, R04 | |
| 0170 | 0047 | MOV R04, R02 | |
| 0175 | 0052 | HLT | |

| LAdd | CPU Instruction | Binary Code | Line | Comments |
|---|---|---|---|---|
| **** | CODE: | | | |
| | | | | |
| **** | MAIN PROG... | | | |
| 0000 | MOV #0, R04 | 000000000104 | 0002 | Copy the value of 0 to _$Temp |
| 0006 | MOV R04, R01 | 0001040101 | 0002 | Copy the value of _$TempReg |
| 0011 | MOV R01, R03 | 0001010103 | 0003 | Copy the value of N to _$Temp |
| 0016 | CMP #5, R03 | 300000050103 | 0003 | Compare _$TempReg17$ with |
| 0022 | JGE 52 | 21020034 | 0003 | Jump to code at address 52 if s |
| 0026 | MOV R04, R05 | 0001040105 | 0003 | Copy the value of _$TempReg |
| 0031 | MOV R02, R03 | 0001020103 | 0004 | Copy the value of P to _$Temp |
| 0036 | ADD #1, R03 | 110000010103 | 0004 | Add: _$TempReg20$ = _$Tem |
| 0042 | MOV R03, R04 | 0001030104 | 0004 | Copy the value of _$TempReg |
| 0047 | MOV R04, R02 | 0001040102 | 0004 | Copy the value of _$TempReg |
| 0052 | HLT | 2F | 0006 | Stop simulation |
| | | | | |
| **** | DATA: | | | |

มี jump instruction อยู่ 1 ครั้ง คือ JGE 52

ซึ่งแปลว่า Jumps to Ladd 52 if greater than or equal

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:

มี jump instruction อยู่ 2 ครั้ง คือ

- JGT 68 แปลว่า Jumps to Ladd 68 if greater than
- JMP 28 แปลว่า Jumps unconditionally to Ladd 28

| LAdd | CPU Instruction | Binary Code | Line | Comments |
|---|---|---|---|---|
| **** | CODE: | | | |
| | | | | |
| **** | MAIN PROG... | | | |
| 0000 | MOV #1, R03 | 000000010103 | 0002 | Copy the value of 1 to _STemp |
| 0006 | MOV R03, R01 | 0001030101 | 0002 | Copy the value of _STempReg |
| 0011 | MOV #1, R04 | 000000010104 | 0003 | Copy the value of 1 to _STemp |
| 0017 | MOV R04, R02 | 0001040102 | 0003 | Copy the value of _STempReg |
| 0022 | MOV #10, R04 | 0000000A0104 | 0003 | Copy the value of 10 to _STem |
| 0028 | CMP R04, R02 | 3001040102 | 0003 | Compare N with _STempReg2 |
| 0033 | JGT 68 | 20020044 | 0003 | Jump to code at address 68 if s |
| 0037 | MOV R01, R03 | 0001010103 | 0004 | Copy the value of P to _STemp |
| 0042 | MUL #2, R03 | 140000020103 | 0004 | Multiply: _STempReg23$ = _S |
| 0048 | MOV R03, R05 | 0001030105 | 0004 | Copy the value of _STempReg |
| 0053 | MOV R05, R01 | 0001050101 | 0004 | Copy the value of _STempReg |
| 0058 | ADD #1, R02 | 110000010102 | 0005 | Add: N = N + 1 |
| 0064 | JMP 28 | 1D02001C | 0005 | Jump to code at address 28 (U |
| 0068 | HLT | 2F | 0006 | Stop simulation |
| | | | | |
| **** | DATA: | | | |
| | | | | |

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

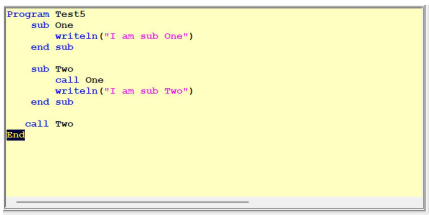
Click on the **RESET** button in **PROGRAM LIST** frame. Now you'll manually execute this program instruction by instruction. To do this double-click the currently highlighted instruction. So, you'll start with the **MSF** instruction, and then do the **CAL** instruction, etc. As you execute the program 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.

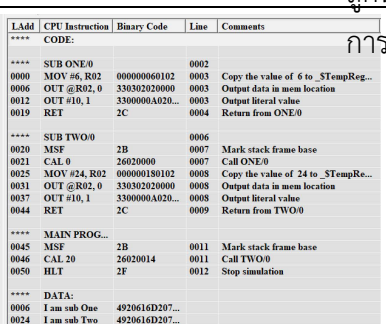
Can you explain what is happening each time a **CAL** or a **RET** instruction is executed and how they affect the **PROGRAM STACK** contents.

Answers and Results:

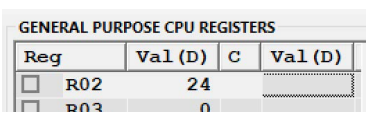
คำสั่ง CAL
- เมื่อคำสั่ง CAL ทำงาน โปรแกรมจะกระโดดไปยังที่อยู่ของ sub ที่ถูกเรียก
- ที่อยู่ของคำสั่งถัดไป (ที่อยู่ของคำสั่งหลังจาก CAL) จะถูกเก็บไว้ใน PROGRAM STACK

คำสั่ง RET
- เมื่อคำสั่ง RET ทำงาน โปรแกรมจะกระโดดกลับไปยังที่อยู่ที่ถูกเก็บไว้ใน PROGRAM STACK
- ที่อยู่ที่ถูกเก็บไว้ใน PROGRAM STACK จะถูกนำออกจาก stack และโปรแกรมจะดำเนินการต่อจากที่อยู่นั้น



| LAdd | CPU Instruction | Binary Code | Line | Comments |
|---|---|---|---|---|
| **** | CODE: | | | |
| **** | SUB ONE/0 | | 0002 | |
| 0000 | MOV #6, R02 | 000000060102 | 0003 | Copy the value of 6 to _STempReg... |
| 0006 | OUT @R02, 0 | 330302020000 | 0003 | Output data in mem location |
| 0012 | OUT #10, 1 | 3300000A020... | 0003 | Output literal value |
| 0019 | RET | 2C | 0004 | Return from ONE/0 |
| **** | SUB TWO/0 | | 0006 | |
| 0020 | MSF | 2B | 0007 | Mark stack frame base |
| 0021 | CAL 0 | 26020000 | 0007 | Call ONE/0 |
| 0025 | MOV #24, R02 | 000000180102 | 0008 | Copy the value of 24 to _STempRe... |
| 0031 | OUT @R02, 0 | 330302020000 | 0008 | Output data in mem location |
| 0037 | OUT #10, 1 | 3300000A020... | 0008 | Output literal value |
| 0044 | RET | 2C | 0009 | Return from TWO/0 |
| **** | MAIN PROG... | | | |
| 0045 | MSF | 2B | 0011 | Mark stack frame base |
| 0046 | CAL 20 | 26020014 | 0011 | Call TWO/0 |
| 0050 | HLT | 2F | 0012 | Stop simulation |
| **** | DATA: | | | |
| 0006 | I am sub One | 4920616D207... | | |
| 0024 | I am sub Two | 4920616D207... | | |



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
```

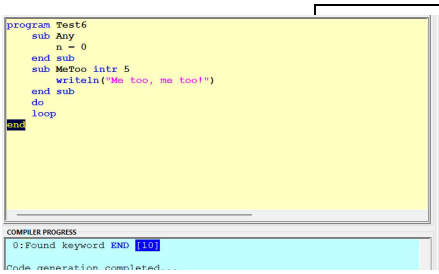
Compile the above program. Look at the code generated. What address does subroutine “MeToo” start at? Make a note of this number. Next, load the compiled program in memory. In the CPU simulator window click on the **INTERRUPTS…** button. The **Interrupts** window will show. Make a note of the interrupt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

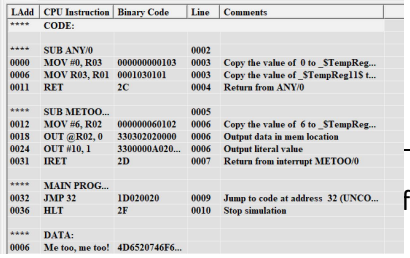
Make sure the **Interrupts** window stays on top. Click on the **INPUT/OUTPUT…** button and make sure the **Console** window also stays on top. Now slide the speed of the CPU simulator to the fastest speed and run the program. Make a note of what you are observing. What is the main purpose of the “**do**”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



| LAdd | CPU Instruction | Binary Code | Line | Comments |
|---|---|---|---|---|
| **** | CODE: | | | |
| **** | SUB ANY/0 | | 0002 | |
| 0000 | MOV #0, R03 | 000000000103 | 0003 | Copy the value of 0 to _STempReg... |
| 0006 | MOV R03, R01 | 0001030101 | 0003 | Copy the value of _STempReg11$ t... |
| 0011 | RET | 2C | 0004 | Return from ANY/0 |
| **** | SUB METOO... | | 0005 | |
| 0012 | MOV #6, R02 | 000000060102 | 0006 | Copy the value of 6 to _STempReg... |
| 0018 | OUT @R02, 0 | 330302020000 | 0006 | Output data in mem location |
| 0024 | OUT #10, 1 | 3300000A020... | 0006 | Output literal value |
| 0031 | IRET | 2D | 0007 | Return from interrupt METOO/0 |
| **** | MAIN PROG... | | | |
| 0032 | JMP 32 | 1D020020 | 0009 | Jump to code at address 32 (UNCO... |
| 0036 | HLT | 2F | 0010 | Stop simulation |
| **** | DATA: | | | |
| 0006 | Me too, me too! | 4D6520746F6... | | |

intr 5 เก็บค่ากาาเรียกใช้งานไว้ใน interrupts เมื่อกด trigger จะส่งสัญญาณ interrupt ไปให้เรียกใช้ค่านั้น

**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.

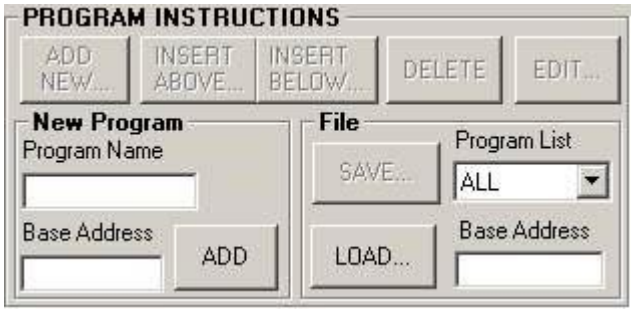
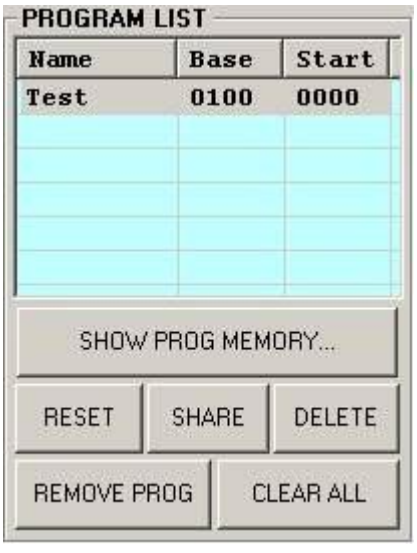



**Figure 9 - Program Instructions View**

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.

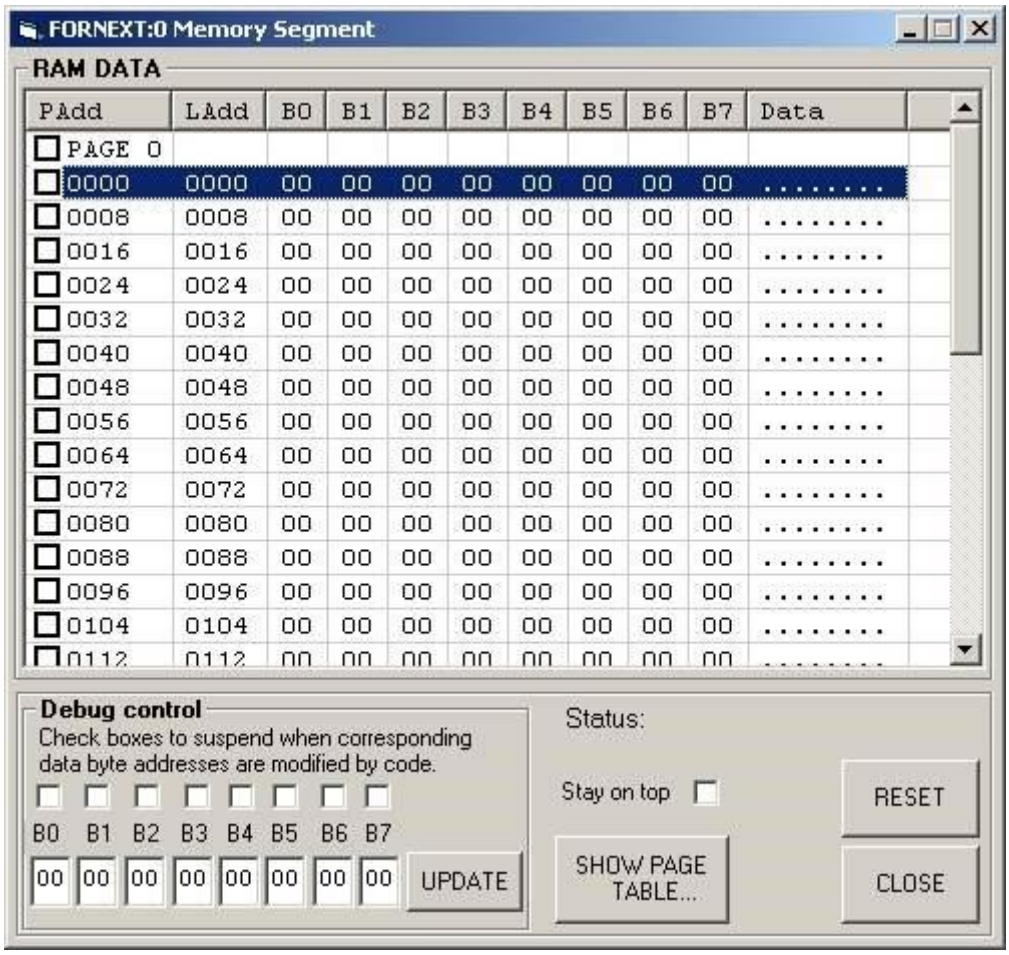



**Figure 11 - Program memory page**

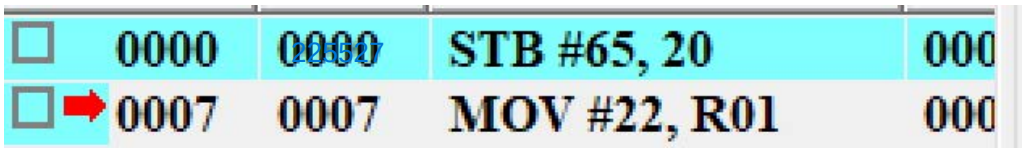
You are now ready to enter instructions into this view. You do this by clicking on the **ADD NEW…** button. This will display the **Instructions: CPU0** window. Use this window to enter the instructions. Use the appendix provided as a reference to the simulator's instruction set architecture.
Complete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

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

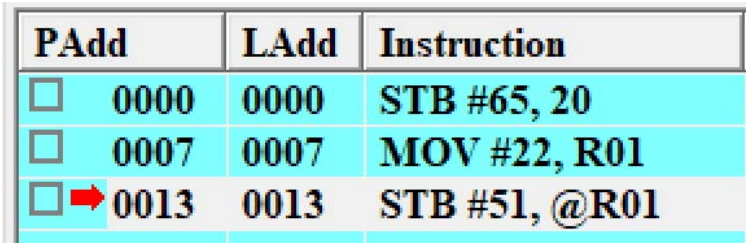
Results:



| | | | |
|---|---|---|---|
| 0000 | 0000 | STB #65, 20 | 000 |
| 0007 | 0007 | MOV #22, R01 | 000 |

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**.

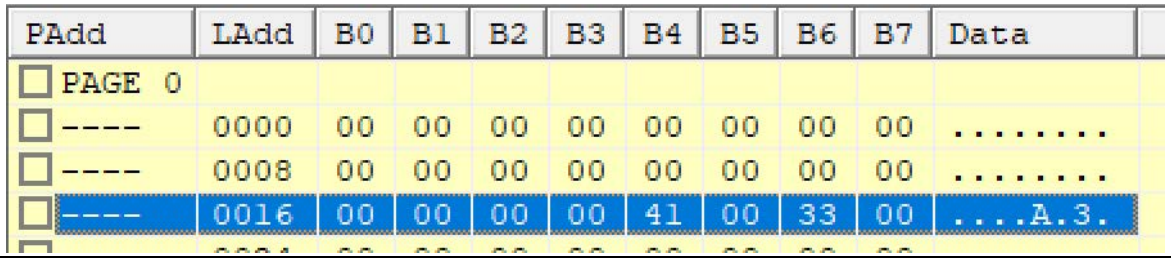
Results:



| PAdd | LAdd | Instruction |
|---|---|---|
| 0000 | 0000 | STB #65, 20 |
| 0007 | 0007 | MOV #22, R01 |
| 0013 | 0013 | STB #51, @R01 |

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.

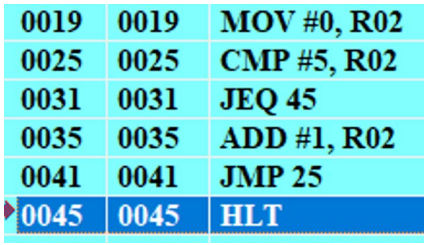
Results:



| PAdd | LAdd | B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 | Data |
|---|---|---|---|---|---|---|---|---|---|---|
| PAGE 0 | | | | | | | | | | |
| ---- | 0000 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ........ |
| ---- | 0008 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ........ |
| ---- | 0016 | 00 | 00 | 00 | 00 | 41 | 00 | 33 | 00 | ....A.3. |

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:



| | | |
|---|---|---|
| 0019 | 0019 | MOV #0, R02 |
| 0025 | 0025 | CMP #5, R02 |
| 0031 | 0031 | JEQ 45 |
| 0035 | 0035 | ADD #1, R02 |
| 0041 | 0041 | JMP 25 |
| 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:

0D ขึ้นบรรทัดใหม่

0A เลื่อนบรรทัด

ต่อจาก hello 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:

| | | | | |
|---|---|---|---|---|
| 0026 | 0026 | MOV #24, R12 | 0000 | 0 |
| 0032 | 0032 | OUT @R12, 0 | 0000 | 4 |
| 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:

| dd | LAdd | Instruction | Base | T |
|---|---|---|---|---|
| 0000 | 0000 | CMP #5, R02 | 0000 | 3 |
| 0006 | 0006 | JEQ 25 | 0000 | 2 |
| 0010 | 0010 | MSF | 0000 | 2 |
| 0011 | 0011 | CAL 26 | 0000 | 2 |
| 0015 | 0015 | ADD #1, R02 | 0000 | 1 |
| 0021 | 0021 | JMP 0 | 0000 | 2 |
| 0025 | 0025 | HLT | 0000 | 2 |
| 0026 | 0026 | MOV #24, R12 | 0000 | 0 |
| 0032 | 0032 | OUT @R12, 0 | 0000 | 4 |
| 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:

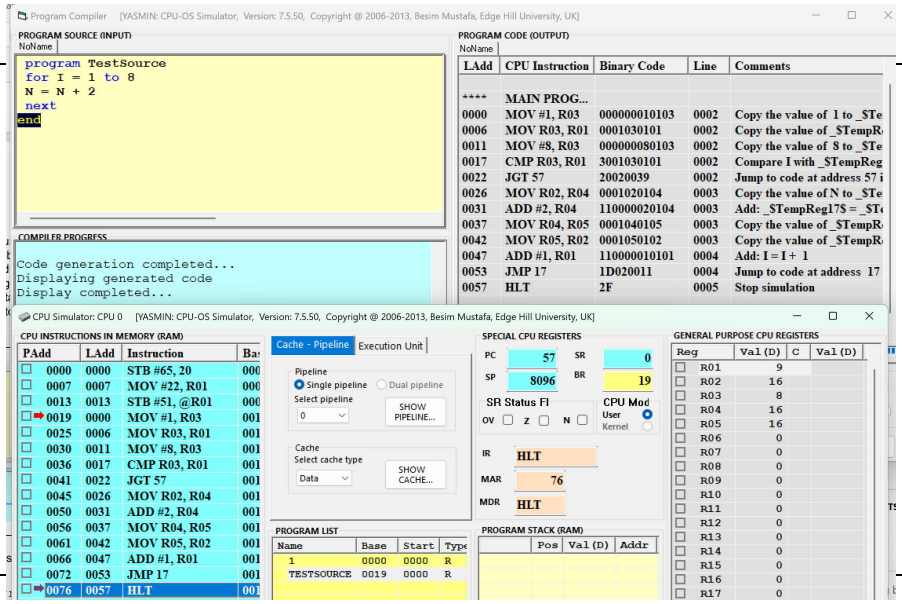
| ก่อนคำสั่ง CAL | หลังคำสั่ง CAL | หลังคำสั่ง RET |
|---|---|---|
| - PC register จะชี้ไปยังที่อยู่ของคำสั่ง CAL ที่จะถูกดำเนินการ<br>- Hardware stack จะไม่มีการเปลี่ยนแปลงใด ๆ ในขณะนี้ | - PC register จะชี้ไปยังที่อยู่ของคำสั่งแรกในซับรูทีนที่ถูกเรียก<br>- ที่อยู่ของคำสั่งหลังจาก CAL จะถูกเก็บใน Hardware stack เพราะต้องเก็บที่อยู่ของคำสั่งถัดไปเพื่อให้สามารถกลับมาทำงานต่อจากจุดที่หยุดได้ | - PC register จะชี้ไปยังที่อยู่ของคำสั่งถัดไปหลังจาก CAL ในโปรแกรมหลัก<br>- ที่อยู่ที่ถูกเก็บไว้ใน Hardware stack จะถูกนำออก เพราะต้องนำที่อยู่ที่ถูกเก็บไว้ใน Hardware stack ออกมา เพื่อให้สามารถกลับมาทำงานต่อจากจุดที่หยุดได้ |

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:



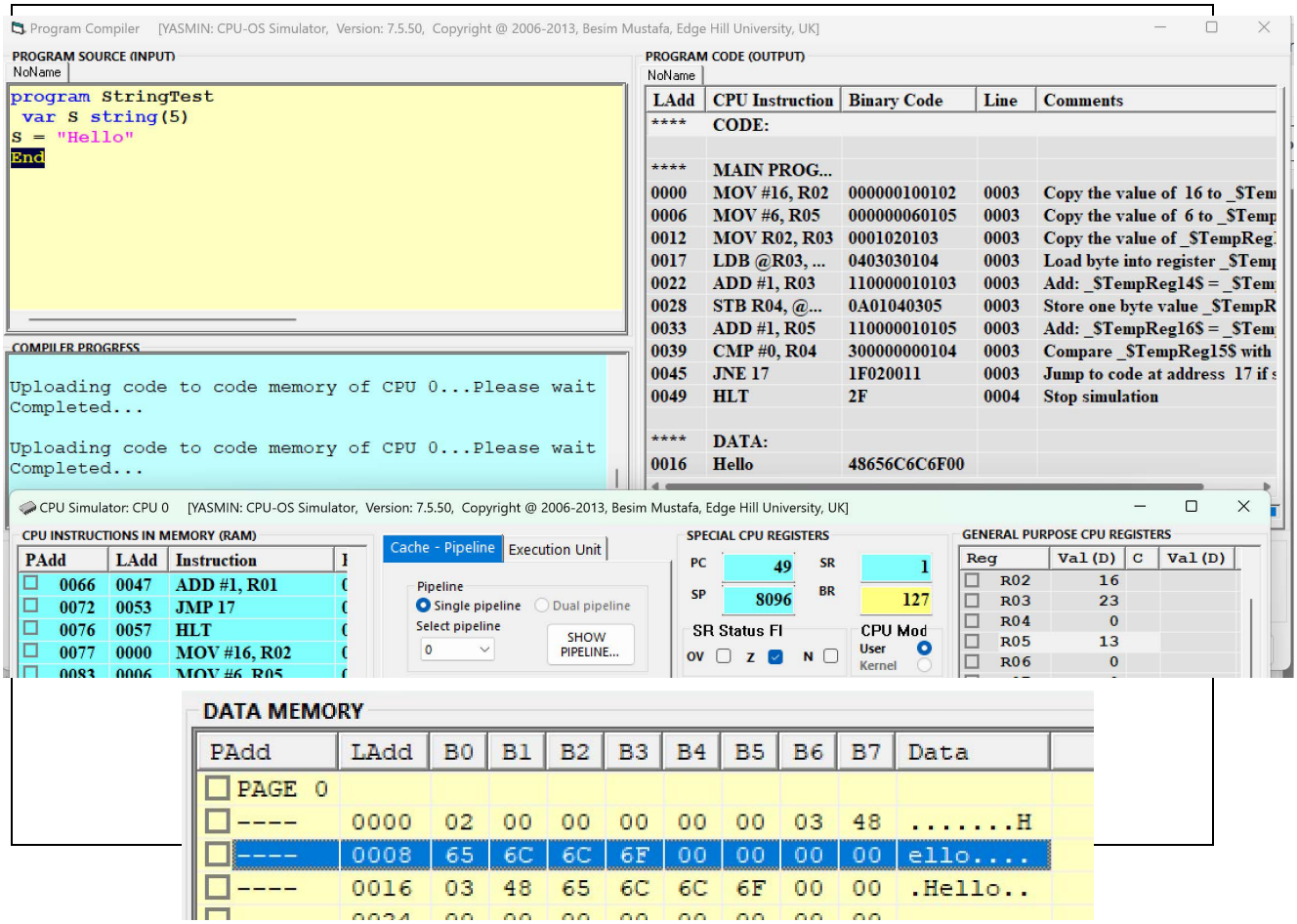

13.Enter the source below and compile it.

```
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.<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 e.g.<br>**LDB 1000, R02** ;loads one byte value from memory location 1000<br>**LDB @R00, R01** ;memory location is specified in register R00 |
| LDW | Load a word (2 bytes) from memory to register e.g.<br>**LDW 1000, R02** ;loads two-byte value from memory location 1000<br>**LDW @R00, R01** ;memory location is specified in register R00 |
| STB | Store a byte from register to memory e.g.<br>**STB #2, 1000** ;stores value 2 into memory location 1000<br>**STB R02, @R01** ;memory location is specified in register R01 |
| STW | Store a word (2 bytes) from register to memory e.g.<br>**STW R04, 1000** ;stores register R04 into memory location 1000<br>**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.<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 e.g.<br>**POP R05** ;pops contents of top of stack into register R05 |

| **Arithmetic instructions** | |
|---|---|
| ADD | Add number to register; add register to register 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 e.g.<br>**JMP 100** ;unconditionally jumps to address location 100 |
| JLT | Jump to instruction address if less than (after last comparison) e.g.<br>**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.<br>**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<br>e.g. To call a subroutine starting at address location 1000 use the following sequence of instructions<br>**MSF** ;always needed just before the following instruction<br>**CAL 1000** ;will cause a jump to address location 1000 |

| | |
|---|---|
| RET | Return from subroutine<br>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.<br>**HLT** ;stops the simulation run (<u>not</u> the simulator itself) |
| **Comparison instruction** | |
| CMP | Compare number with register; compare register with register 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 Note:<br>If R03 = R01 then the status flag **Z** will be set<br>If R03 > R01 then non of the status flags will be set<br>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<br>e.g. to display a string starting in memory address 120 (decimal) on console device do the following<br>**OUT 120, 0** ;the string is in address location 120 (direct addressing)<br>**OUT @R02, 0** ;register R02 has number 120 (indirect addressing) |

************************************************