Programming Model 2

A. Introduction

Objectives

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

- Use direct addressing mode of accessing data in memory
- Use indirect addressing mode of accessing data in memory
- Write a subroutine and call it
- 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, i.e. addressing modes such as direct and indirect addressing.

They also define interactions between the above components. It is this low-level programming model which makes programmed computations possible.

D. Simulator Details

You can find the simulator details relevant to this tutorial in Programming Model 1 tutorial. As a result, these are not repeated here except the new information on how to view and access program data memory view; this tutorial requires access to this part of the simulator. You will find the details below.

Program data memory view

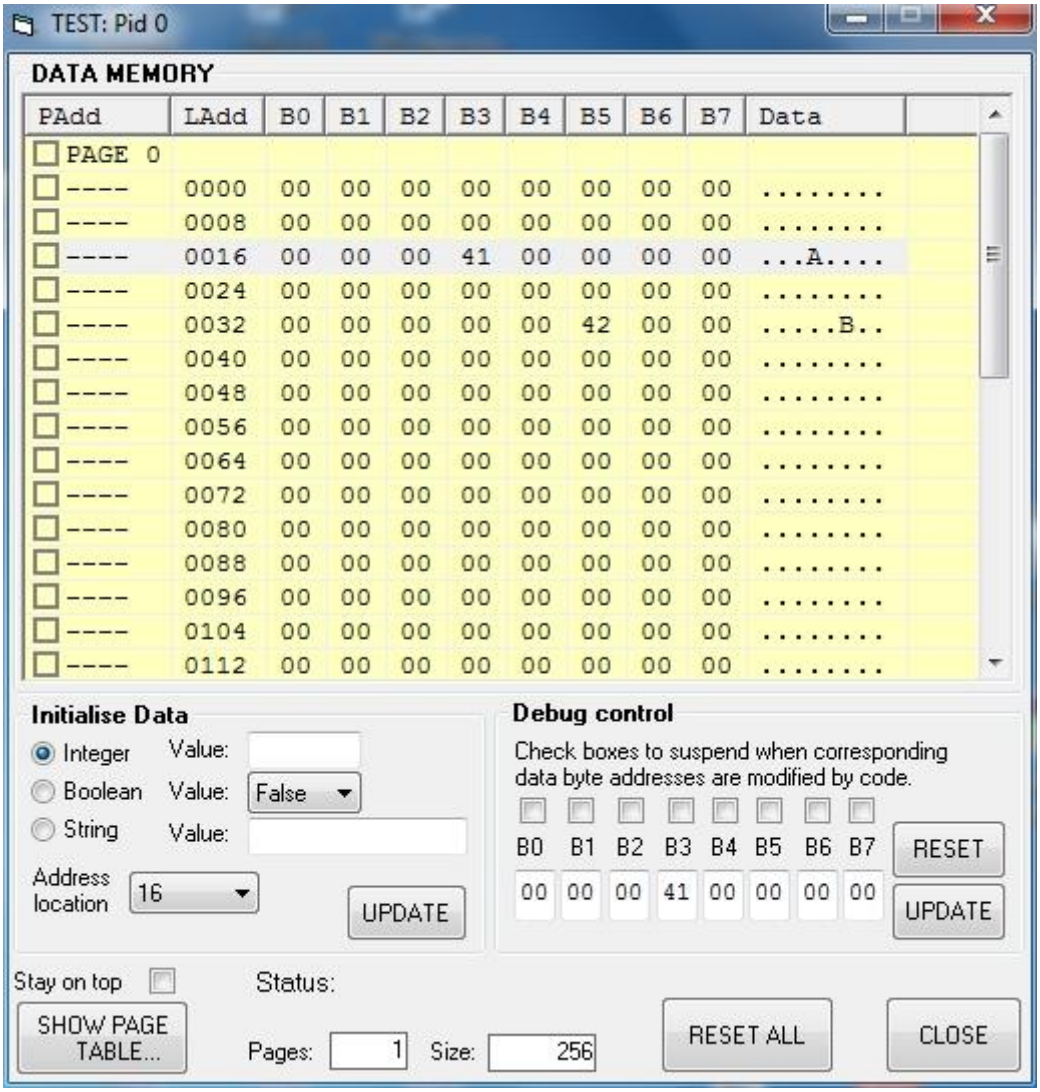

Image 1 - Program data memory view

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 1 above. You can display this window by clicking the **SHOW PROGRAM DATA MEMORY…** button in the main CPU window. 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 1, there are non-zero data bytes in address locations 19 and 37. These data bytes correspond to displayable characters capital A and B.

To manually 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.

E. Lab Exercises - Investigate and Explore

Enter the instructions you create in order to answer the questions in the blank boxes. Refer to Appendix at the end of this document to find the details on the desired instructions. You are expected to execute the instructions you created on the simulator in order to verify your answers.

A. Instructions for writing to and reading from memory (RAM):

1. Locate the instruction that **stores** a byte in program data memory and use it to store number 65 in memory address location 20 (this uses **memory direct** addressing method).

2. Move number 51 into register R04. Use the store instruction to **store** the contents of R04 in program data memory location 21 (this uses **register direct** addressing method).

3. Move number 22 into register R04. Use this information to indirectly **store** number 59 in program data memory (hint: you will need to use the '@' prefix for this – see the list of instructions in appendix) - (this uses **register indirect** addressing method).

4. Locate the instruction that **loads** a byte from program data memory into a register. Use this to load the number in memory address 22 into register R10.

5. Write a loop in which 10 numbers from 41 to 50 are written in program data memory starting from memory address 24 (hint: use **register indirect** addressing where register R04 indirectly represents memory address to write to and increment it to address increasing memory locations).

6. Manually initialise part of program data memory starting from address 40 with string "CPU INSTRUCTIONS RULE" (see section D above on how to do this). Write a loop in which this string is copied to another part of the program data memory starting from address 64.

B. Instructions for calling subroutines and passing parameters to subroutines:

1. Add the following code and run it starting from the first MOV instruction (to do this you need to first select this instruction and then click on the RUN button).

   NOTE:
   Ask your tutor how to add labels to your code. A Label represents the address of the instruction immediately following it. For example, 'Label2' below represents the address of the MOV instruction following it. Labels are used by the jump instructions by putting a '$' in front of the label name, e.g. the 'JMP $Label2' instruction will jump to the instruction at address represented by the label 'Label2'.

```
Label2
MOV #16, R03
MOV #h41, R04
Label3
STB R04, @R03
ADD #1, R03
ADD #1, R04
CMP #h4F, R04
JNE $Label3
HLT
```

NOTE:
This code stores numbers hex 41 to hex 4F starting from program memory address 16. These numbers are ASCII codes for displayable characters of the alphabet. You need to understand how this is done by this code in order to benefit from these exercises.

   a. Make a note of what you see in the program's data area after the program stops running:

   b. Suggest what the significance of @ in **@R03** might be:

2. Add the following subroutine calling code but do **NOT** run it yet:

```
MSF
CAL $Label2
HLT
```

   Now convert the code in (1) above into a subroutine by simply replacing the **HLT** instruction with the **RET** instruction.

a. Make a note of the contents of the **PROGRAM STACK** after the instruction **MSF** is executed (you can execute this instruction by simply double-clicking on it):

b. Make a note of the contents of the **PROGRAM STACK** after the instruction **CAL** is executed (you can execute this instruction by simply double-clicking on it):

c. What is the significance of the additional information on the stack after executing the **CAL** instruction?

3. Let's make the above subroutine a little more flexible. Suppose we wish to change the number of characters stored when calling the subroutine. Modify the calling code in (2) as below:

```
MSF
PSH #h60        ← puts the number hex 60 on top of the stack
CAL $Label2
HLT
```

a. Now modify the subroutine code in (1) as below and run the above calling code (starting from the **MSF** instruction). Pay particular attention to the behaviour of the stack:

```
Label2
MOV #16, R03
MOV #h41, R04
POP R05         ← puts the number on top of the stack in R05
Label3
STB R04, @R03
ADD #1, R03
ADD #1, R04
CMP R05, R04    ← compares R05 with R04
JNE $Label3
RET
```

b. Add a second parameter to the above code that can provide different starting addresses for the data transfer to memory and test it on the simulator. Write the modified code below:

4. Write a subroutine that takes two numbers as parameters, adds them and returns the result in register R00, i.e. when the subroutine is exited the result is available in R00 to the rest of the program. Test it on the simulator by writing the calling instructions that include passing two numbers on the stack. Copy the code below (including the calling instructions):

5. Write a subroutine that takes two numbers as parameters, compares them and returns the higher of the two as the result in register R00 (consider what should happen if the two numbers are the same). Test it on the simulator by writing the calling instructions that include passing two numbers on the stack. Copy the subroutine code alone below:

Appendix - Simulator Instruction Sub-set

Inst.	Description
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 1022, R03** loads a byte from memory address 1022 into R03 **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 Same as in LDB but a word (i.e. 2 bytes) is loaded into a register
STB	Store a byte from register to memory **STB R07, 2146** stores a byte from R07 into memory address 2146 **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 Same as in STB but a word (i.e. 2 bytes) is loaded stored in memory
PSH	Push data to top of hardware stack (TOS); push register to TOS e.g. **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 **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 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 where there is another instruction

JLT	Jump to instruction address if less than (after last comparison)
JGT	Jump to instruction address if greater than (after last comparison)
JEQ	Jump to instruction address if equal (after last comparison instruction) e.g. **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. e.g. **MSF** reserve a space for the return address on program stack **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) 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 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 R01 = R03 then the status flag **Z** will be set, i.e. the **Z** box is checked. If R01 < R03 then none of the status flags will be set, i.e. none of the status flag boxes are checked. If R01 > R03 then the status flag **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 e.g. **OUT 16, 0** outputs contents of data in location 16 to the console (the second parameter must always be a 0)