Programming Model 3

A. Introduction

Objectives

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

- Create conditional CPU instructions (compare and jump)
- Create iterative loops of CPU instructions
- Use indirect addressing modes of accessing data in memory

This lab session is designed to illustrate the concepts and support the lectures about CPU instruction sets and the CPU programming model.

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 data in instructions, i.e. addressing modes.

They also define interactions between the above components. It is this low-level programming model which makes programmed computations possible. You should do additional reading in order to form a better understanding of the different parts of a modern CPU architecture (refer to the recommended reading list available in the module handbook and on the BB).

D. Simulator Details

You can find the simulator details relevant to this tutorial in Programming Model 1 tutorial you previously completed. 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 in the next page.

Program data memory view

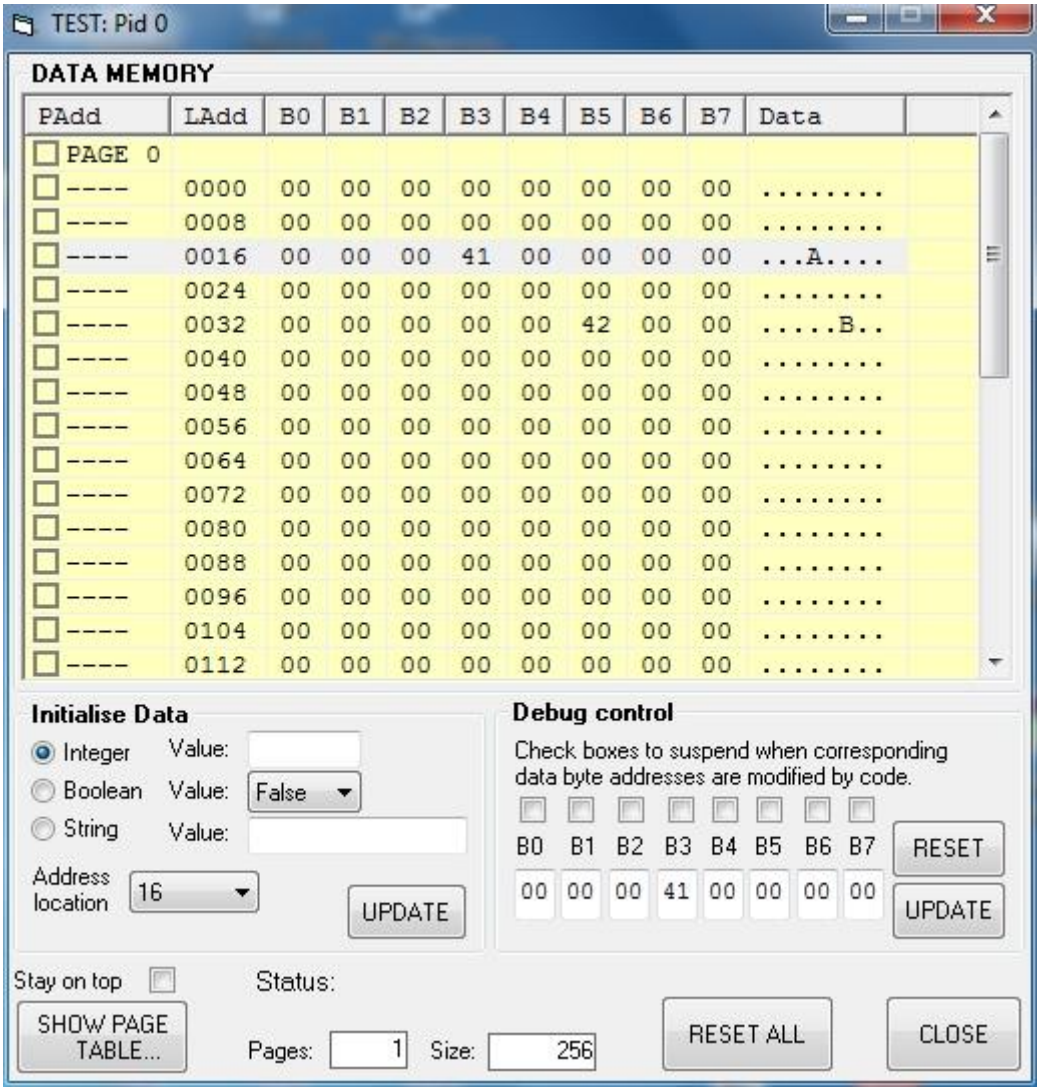

Image 1 - Program data memory view

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 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. Your tutor(s) will be available to help you if you require assistance.

A. Loops using jump and compare instructions:

1. Write a conditional statement such that if R02 is **greater than** (>) R01 then R03 is set to 8. (Use R01 as the first operand and R02 as the second operand)

2. Write a conditional statement such that if R02 is **less than or equal to** (<=) R01 then R03 is set to -5. (Use R01 as the first operand and R02 as the second operand)

3. Write a conditional statement such that if R01 = 0 then R03 is set to 5 **else** R03 is set to R01 plus 1.

4. Write a loop that repeats **5 times** where R02 is incremented by 2 every time the loop repeats.

5. Write a loop that repeats **while** R04 is > 0. Set the initial value of R04 to 8.

6. Write a loop that repeats **until** R05 is > R09. Set the initial values of R05 to 0 and R09 to 12.

7. Write a routine that **pushes** numbers 8 and 2 on top of stack. It then **pops** the two numbers one by one from stack, **adds** them and **pushes** the result back to top of stack.

8. Challenge #1: Place 15 numbers from 1 to 15 on top of stack using the **push** instruction in a loop. Then in a second loop use the **pop** instruction to pop two numbers from top of stack, **add** them and **push** the result back to top of stack. The second loop repeats this until there is only one number left on top of stack which should be the final result.

B. Instructions for writing to and reading from memory (RAM):
The following instructions access the program's data memory. You can display this memory so that you can observe the results by referring to Image 1 and the related text in section D above.

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

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

11. Move number 22 into register R04. Use this information to indirectly **store** number 58 in memory (hint: you will need to use the '@' prefix for this – see the list of instructions in appendix) - (this uses **register indirect** addressing method).

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

C. Putting it together:

13. Challenge #2: write a loop in which 10 numbers from 48 to 57 are stored as single bytes in memory starting from memory address 24. You should use **register indirect** addressing method of storing the numbers in memory (see exercise 11 above).

14. Challenge #3: write a loop in which the numbers stored in memory by the program in (13) above are copied to a different part of the memory starting from address location 80.

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)