11 Low-level Microprocessor Programming Concepts

Parts List

EMMAII educational Board Digi Designer

Laboratory Objective

In the first part of this lab, you will enter and run a program given to you that copies a block of data from one area in memory to another. In the second part, you will develop a traffic control signal program.

Last laboratory session provided the basics to low-level programming on the EMMAII system. During this laboratory session, you will learn some generic concepts of assembly level programming. By the end of this laboratory session, you should feel confident in approaching somewhat complex assembly-level programming problems. For each exercise, you will follow a systematic method for completing the problem. These steps were covered in the lecture on Assembly Level Programming Techniques and are briefly summarized below.

1) Understand Problem Statement. (dissect problem into discrete functional blocks, create "pseudo code" that overviews the basic programming processes)

2) Program Design. (flowchart the logical flow of the program, look for reusable code to set up subroutines, test logic to look for possible errors)

3) Initial program documentation and entry. ("map" your program design to the specific microprocessor assembly commands)

4) Debug program. (step through program, set up some temporary break points to test the program in blocks)

5) Optimize program operation and continue debugging.

6) Provide final program documentation. (neat flowchart, clear concise comments integrated into program source code, program explanation)

Pre-Lab Tasks

In preparation for this lab session read through the attached documentation. Note Part II contains a short section on how to develop and use subroutines to alleviate repetitive code segments such as delay routines.
Come to the lab with the program for Part II written as much as possible along with the actual machine code that belongs in which addresses.

In-Lab Tasks

Part I) A Program that copies Data.

The flowchart below is an example of performing a data block move. The following example will guide you through how to move a data block addresses 0020-0024 to addresses 02F0-02F4. Follow the program sheets below the flowchart and pay careful attention to the addressing modes:

* LDX#05: This is an immediate instruction to load the X register with 05. This has the effect of setting up the data pointer in this program.

* DEX: An implied instruction to decrement the contents of the X register

* LDA 20,X: A ZERO PAGE instruction to load accumulator with the contents of address 00200+X, where X is 4. Hence the instruction loads accumulator with data from 0024.

* STA 02F0,X: An absolute instruction to store accumulator at address 020F0+X which is 02F4. Hence data from 0024 is stored at 02F4.

* BNE: After the decrement instruction the BNE instruction is asking the branch if X is not equal to zero. At this point in the program X is now 4 and not equal to zero and the branch is taken back to address 0202. The data F8 in the HEX region of the program is a negative number giving the backward displacement of the branch.

* JMP: This is the next instruction after X reaches zero. The JMP instruction is the same as the previous program and jumps back on itself and terminates the program.

Note: A better way to terminate a program on the EMMAII is to jump to the monitor of the machines to effect a reset condition. The location of the reset condition in the monitor program is FEE0. Try this to see the results.

EXERCISE

Enter and run the program. Modify it so that it terminates to the monitor. Demonstrate to the TA. Make sure that you understand what is wrong with the program as given-regarding the conditional branch instruction. As given it works because the last data moved is (coincidentally) hex 00.
Example: Move data from data block addresses 0020-0024 to addresses 02F0-02F4.

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Symbolic Assembler Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR 1 2 3</td>
<td>LABEL MNEM OPERAND COMMENTS</td>
</tr>
<tr>
<td>0200 A2 05</td>
<td>LDX# 05 Set up data pointer</td>
</tr>
<tr>
<td>0202 CA 20</td>
<td>DEX 20 Set up data item</td>
</tr>
<tr>
<td>0203 B5 20</td>
<td>LDA 20,X Transfer data</td>
</tr>
<tr>
<td>0205 9D F0</td>
<td>STA 02F0,X Get next valid data item</td>
</tr>
<tr>
<td>0208 D0 F8</td>
<td>BNE Get next valid data item</td>
</tr>
<tr>
<td>020A 4C 0A</td>
<td>JMP FINISH</td>
</tr>
</tbody>
</table>

Data to be moved: 0020 00 0021 01 0022 02 0023 03 0024 04

Locations for data to be shifted to:
- 02F0
- 02F1
- 02F2
- 02F3
- 02F4

Memory space reserved for data
Part 2) A Program using SUBROUTINES.

It is very easy for a program to become repetitive, containing sections of code that are required several times. It is also possible for programs to become confusing because it is difficult to identify what routine does what. Both these problems are alleviated by the use of subroutines.

A subroutine is a section of program that is only written once but can be used as often as you require. For this to be possible, you need to use some kind of branching commands. They can be conditional, (i.e based on the immediate value of the process status registers) or non-conditional (simply jump to memory location where the routine is located and return once the subroutine is completed).

The JSR, (jump to subroutine) and RTS, (return from a subroutine) are an example of non-conditional jump and the corresponding PROPER return from non-linear program execution.

The RTS instruction MUST be included at the exit point in a section of program that is deemed to be a subroutine. It is an instruction using implied addressing. (i.e it already knows where to get the Program Counter; it “pops” it from the stack). The figure given illustrates subroutine use.

NOTE: Double lines distinguish a subroutine in a flowchart form and how several subroutines can be nested.

EXERCISE

Develop a program which will control a simple traffic signal (simulated with LEDs). Six separate signals should be generated - for the two sets of green, yellow, and red lights (a set for each direction through the intersection). The signals will be output from the EMMA computer system through Port A. Program bits 0 through 5 to be outputs and write a pattern of sequences to the lights so that they operate as a basic traffic signal should. Each pattern in the sequence should be “held” at Port A for a certain delay before the next program is written (that is, the green light should be on for a while [about 5 seconds], then the yellow [for about one second]...). To implement the delay use a subroutine.

More Information about delay: The delay routine is basically a way to “waste” clock cycles. The easiest way to implement a delay is to load a register with a value, decrement that value and then check to see if the decrement caused the register to be “zeroed”. Since this procedure will be used over and over again, this code can be put into a subroutine and called by the main program whenever a delay is required. It is possible to determine exactly how long the delay will be by counting cycles. For example, if you write a subroutine that takes one second to run, then you can easily implement the 1 and 5 second delays called for in the assignment.
The delay routine given to you in Lab #9 takes .00178 sec to run.

a) Effect of a subroutine call and return.

b) Nested subroutine calls.

Post Lab Report

Prepare a report that describes the program you wrote for PartII. In your report, state your program's purpose and verbally describe how it works (a flowchart may be helpful). Include the program code complete with addresses, machine code, and clear, concise comments. Give the details of the timing of the light pattern that your code produces.