Lab 1: Introduction to Microprocessor Programming

CEG 3136 B - Computer Architecture II

Fall 2019

School of Electrical Engineering and Computer Science
University of Ottawa

Professor Basim HAFIDH

Teaching Assistants: Hamzah ALGHAMDI
Harsimran SINGH

Kimberley Paradis #8569079

Brad Halko #300000853

Experiment Date: 20/09/2018

Submission Date: 04/10//2018

Objectives

- Become familiar with the Debug-12 Monitor
- Introduce Assembly language programming
    - Plan, write, and run assembler code,
    - Learn basic principles of debugging,
    - Apply different addressing modes,
- Design and Implement a simple software module.

Equipment

- Windows PC
- Dragon 12 Plus Trainer

Purpose

This lab presents the design of an alarm system simulation dependent on the assembler and executed on the Dragon 12 Plus Trainer. Based upon modular and structured programming , the system will have a diverse amount of features.

Lab 1 dealt with the implementation of the delay module and the arming module for the alarm system, as well as debugging and testing small subroutines and functions within the assembly code and the C code pseudo-functions.

Design and Construction

Part 1

Section A

LDX $EE86: Direct addressing, loads into X at the specified memory block.
JSR 0,X: Offset addressing, directs a "Jump to SubRoutine" instruction to the memory address at 0, in the index register X.
BNE $200F: Branch if Not Equal, directs to another function within the subroutine or program held within memory. Branch instructions use 8-bit relative addressing.

Address	**Content**	**Instructions**	**Description**
2000	CC003A	LDD #$3A	Load ASCII code for ":"
2003	*FEEE86*	LDX $EE86	Load the vector for putchar routine
2006	*1500*	JSR 0,X	Print what's in B on terminal
2008	*FEEE84*	LDX $EE84	Load the vector for getchar routine
2008	*1500*	JSR 0,X	Get a new character in B
200D	*8603*	LDAA #3	Initialize loop counter
200F	*36*	PSHA	Save the counter on stack
2010	*37*	PSHB	Save contents of B on stack

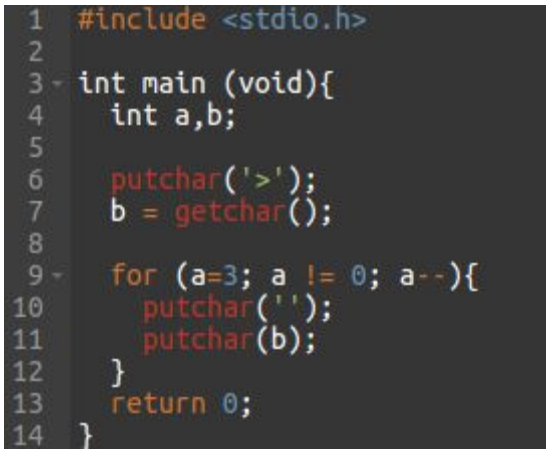
2011	*CC0020*	LDD #$20	Load B with a space
2014	*FEEE86*	LDX $EE86	Load the vector for putchar routine
2017	*1500*	JSR 0,X	Print it on terminal
2019	*33*	PULB	Get original character
201A	*FEEE86*	LDX $EE86	Load the vector for putchar routine
201D	*1500*	JSR 0,X	Print it on terminal
201F	*32*	PULA	Retrieve the counter
2020	*43*	DECA	Decrement loop counter
2021	*26EC*	BNE $200F	If counter <> 0, repeat
2023	*3F*	SWI	Return to the monitor

Section B

Instructions	**Description**	**New Instruction**	**New Description**
LDD #$3A	Load ASCII code for ":"	LDD #$3E	Load ASCII code for ">"
LDD#$20	Load B with a space	LDD#$2C	Load B with a ","

LDAA #3	Initialize Loop counter	LDAA #F	Initialize loop counter to 15

Pseudo C Code



```c
#include <stdio.h>

int main (void){
  int a,b;

  putchar('>');
  b = getchar();

  for (a=3; a != 0; a--){
    putchar('');
    putchar(b);
  }
  return 0;
}
```

Part 2

Tracing and inputting breakpoints into the Debug module allowed us to read the contents of memory at specific target points within the program. Using this, tracing and debugging is able to be conducted in order to diagnose and correct errors without having to try and read line-by-line through the code and find errors the hard way.

D						
A	**B**	**X**	**Y**	**SP**	**PC**	**CCR**
00	00	0000	0000	3C00	0400	SXHI NZVC 1001 0000
00	00	0000	0000	2000	0403	SXHI NZVC 1001 0000
00	00	0000	0000	1FFE	0427	SXHI NZVC 1001 0000
00	00	0000	0000	1FFE	0429	SXHI NZVC 1001 0000
00	00	0000	0000	1FFE	042C	SXHI NZVC 1001 0100
0C	9C	0000	0000	1FFE	0450	SXHI NZVC 1001 0000
0C	9C	0000	0000	1FFE	0456	SXHI NZVC 1001 0000
0C	9C	0000	0000	1FFE	045C	SXHI NZVC 1001 0000

An error that appeared during tracing was “BEQ CDE endif”. This line would not give instructions to branch if the inputted alarm code was incorrect, flagging the inputted code as valid every time. By changing this line to “BNE CDE endif”, the program would not disarm the alarm if the characters are not equivalent to the code. This modification removes the error from the program.

The highlighted line in the table above is the location of the modified code. The lines directly below (0456, 045C) consist of MOVW instructions. By definition, MOVW functions copy the first operand (or register pair) to the second operand (or register pair). MOVW functions only affect data in the memory so only the contents of the PC are affected. The last instruction is an “rts” command that returns the address pointed by the SP register. It decrements by 2 as addresses are stored in bytes of 2.

The addressing mode for the first MOVW instruction (#0) is immediate. It is immediately followed by MOVW #1000 with an absolute addressing mode.

Part 3

In Part 3 of the lab, development and implementation of the Alarm Delay functions was necessary. Using the examples provided within the *delay.asm* file, a short subroutine was developed that would perform the following tasks:

- When called, initiate a countdown that would cycle off the clock of the microprocessor that could be interrupted by the input of the proper alarm arm/disarm code,
- If not interrupted, poll the clock cycles in order to countdown 15 seconds after arming,
- After the countdown was completed, put the system into an “armed” state.

In order to perform the tasks listed above, a simple two-loop subroutine was required. The first loop was the 1 millisecond delay counter, implemented using **NOP** “no operation” mnemonic commands to achieve the correct delay time in addition to a simple decremented counter. The second loop was a simple check to the decrement counter that branched out of the loop when the counter reached 0, which would place the alarm to the “armed” state.

See attached *delay.asm* program (uploaded to Brightspace along with this report) for implementation and functional code, as demonstrated to the TA in the lab session.

Conclusion

The purpose of this lab was to become familiar with the Debug-12 Monitor and assembly language programming. It dealt with a delay module and arming module for an alarm system. In Part1, the process of reviewing the code provided and making slight modifications was very straightforward. Part2 of the lab took more time as the team got familiar with creating breakpoints. There were some issues when attempting to run the code from the first breakpoint to the second. The board had to be reset and a new breakpoint created every time the team reached the previous breakpoint. Finally, part 3, development and implementation of Alarm Delay functions, was done during the second lab session for Lab 1.

With all 3 parts completed, a functional alarm system was created and demonstrated to the TA.