Concordia University

Lab 3: Addressing Modes and 2-D arrays

COEN 311

Lab Section: SN-X

Computer Organization and Software

Penoelo Thibeaud

40212017

Lab Instructor: T. Obuchowicz and Milad Khanchi

Performed on : October 25, 2023

Submitted on : November 8, 2023

I certify that this submission is my original work and meets the Faculty’s Expectations of Originality

# Objective:

This report aims to explore various addressing modes in assembly language programming, focusing on accessing elements of a two-dimensional array stored in memory. Different addressing modes, such as immediate, register, register indirect, post-indexed, pre-indexed, scaled register indirect, and immediate offset, are discussed. The report delves into how these modes facilitate memory access and arithmetic operations, providing insights into the tasks performed by high-level programming language compilers. This exploration allows for an appreciation of the nuances of assembly language and its efficiency in memory access.

# Introduction:

The experiments involve writing assembly language programs to access elements of a 2D array using various addressing modes. The practical application of addressing modes like immediate, register, and register indirect modes is investigated. The main objective is to understand and apply different addressing modes in assembly language to efficiently access memory locations. The experiments are expected to reveal the efficiency and utility of using different addressing modes for specific tasks, leading to a deeper understanding of memory access in assembly language programming.

# Procedure (Methods):

The goal of this lab was to write an ARM assembly language program utilizing the 2-D array address translation formula to access array elements through the register indirect addressing mode, specifically "register indirect with register immediate offset" mode.

## Steps Followed:

1. **Setup of Data Section:**

The .data section of the program was defined with an array representing 3 rows and 2 columns of integers:

.data

array: .byte 3,2,4,1,5,6

1. **Initialization of Registers:**

Two registers were initialized to hold the row and column indices of the desired array element. For example:

mov r1, #1 @ r1 will hold the row index

mov r2, #1 @ r2 will hold the column index

1. **Computation of Offset:**

The mul and add instructions were used to compute the offset value based on the row and column indices.

1. **Accessing the Array Element:**

The calculated offset was used along with the base address of the array to load the desired element into a register through the register indirect addressing mode with offset.

1. **Testing:**

The program was tested using gdb for several typical values of row and column indices to ensure correct functionality. The source code was edited with new values for the row and column indices, re-assembled, re-loaded, and re-run with gdb for each test case. It was not necessary to use loops to iterate over each array element for this lab.

# Results and Discussion:

Understanding different addressing modes is crucial for efficient programming, especially when dealing with memory-intensive tasks.

## Questions:

1. **Determine the value loaded into register r2 by the following ARM assembly language program:**

The value loaded into the register r2 is 0Xdeadbeef. Here is a simple algorithm for the code

* 1. It stores 0Xdeadbeef in “mydata”
  2. It stores the address of “mydata” in “address\_of\_mydata”
  3. It load the address of “address\_of\_mydata” in r1 which is the address of “address\_of\_mydata”
  4. It load the value of the address contain in r1 into r1 which is the address of “mydata”
  5. It load the value of the address contain in r1 which is the value of “mydata” 0Xdeadbeef

1. **Consider the following line from the .lst file for the above program:**

15 0000 EFBEADDE mydata: .word 0xdeadbeef

The first memory used in the lab seems to store the word “deadbeef” data backward with no consistency. To understand what is going on, we will take a look a the endianness.

### Table 1: Both Endianness representation of “deadbeef”

|  |  |  |
| --- | --- | --- |
| **Memory Address** | **Value for little endian** | **Memory Address** |
| 0000 | EF | DE |
| 0008 | BE | AD |
| 0010 | AD | BE |
| 0018 | DE | EF |

The previous table illustrate 2 types of endianness, little and big endianness. Little endian start by storing the least significant byte at the first memory location and goes on where as big endian will start by the most significant bit at the first memory location. As we can see on the table that EFBEADDE is the little endian representation of “deadbeef”.

# Conclusion:

The experiments successfully showcased the utility and efficiency of different addressing modes in assembly language programming. The findings align with the objectives stated, providing a comprehensive understanding of how high-level programming language compilers access memory.

# Appendix:

## Here is my LST file :

ARM GAS 2dArray.s page 1

1 .syntax unified

2 .cpu cortex-m4

3 .thumb

4

5 .global start

6 .data

7 0000 00040020 .word 0x20000400

8 0004 ED000080 .word 0x800000ed

9 0008 00000000 .space 0xe4

9 00000000

9 00000000

9 00000000

9 00000000

10

11 00ec 03020401 array: .byte 3,2,4,1,5,6 @ 2D array with 3 rows and 2 columns.

11 0506

12

13 .text

14 start:

15 0000 0848 ldr r0, =array @ initialise array.

16 0002 4FF00005 mov r5, #0 @ loop counter = 0.

17

18 loop:

19

20 0006 05F10001 add r1, r5, #0 @ Row index

21 000a 4FF00002 mov r2, #0 @ Column index

22 000e 4FF00206 mov r6, #2 @ Move the multiplier into a register

23

24 @ Calculate ((row\_index \* number\_of\_columns) + column\_index)

25 0012 01FB06F3 mul r3, r1, r6 @ r3 = row\_index \* 2 (since we have 2 columns per row).

26 0016 1344 add r3, r3, r2 @ r3 = offset from the start of the array

27

28 @ Access the element using register indirect with offset addressing mode.

29

30 0018 C45C ldrb r4, [r0, r3] @ r4= array[r1][r2] - looks for the byte at the offset

31

32

33 @ Increment the loop counter and check if we've done 3 iterations.

34 001a 05F10105 add r5, r5, #1

35 001e 032D cmp r5, #3

36 0020 F1DB blt loop @ loop if r5 is lower than 3

37

38 0022 FEE7EC00 b . @ infinite loop

38 0000