Assignment 2

1. Introduction

The purpose of this lab is to familiarize the student with the basics of ARM assembly language programming. In this assignment, we will also study the ARM memory addressing modes. The ARM uses a Load/Store Architecture, where data is loaded from memory space to the registers, manipulated, and stored back in memory. In the next assignment, we will study the ARM operand addressing modes used for processing data once loaded into its registers.

2. Load/Store Instructions

The ARM processor is a Load/Store Architecture. This is common to most RISC processors. The ARM does not support memory-to-memory data processing operations. The data values must be moved INTO registers before using them.

The basic load and store instructions are:

- LDR/STR - Load and Store Word (32 bit)
- LDRB/STRB - Load and Store Byte (8 bit)
- LDRH/STRH - Load and Store Half Word (16 bit)
- LDRSB - Load Signed Byte (8 bit load sign extended to 32 bits)
- LDRSH - Load Signed Half Word (16 bit load sign extended to 32 bits)

All of these instructions can be conditionally executed by inserting the appropriate condition code after the STR/LDR. For example, the following instruction loads a byte if the condition code is EQ:

LDREQB

The general load/store instruction syntax is:

<LDR/STR>{<cond>}{<size>} Rp, <address>

where:

- <cond> is an optional processor condition code (to be discussed in later classes)
- <size> is an optional selection of operand size:

  blank for word (32 bit)
B for unsigned byte (8 bit)
H for unsigned half word (16 bit)
SB for signed byte (8 bit sign extended to 32 bits)
SH for signed half word (16 bit sign extended to 32 bits)

Rp is the processor register for the load/store operation

<address> is an address expression discussed in the next section

3. Addressing Modes

The ARM processor provides a number of addressing modes for the efficient implementation of C/C++ compilers. High-level languages need to access elements via pointers, arrays of elements and structure members. The ARM addressing modes accommodate all of these uses.

The ARM load/store instructions as described in the previous section are implemented in two forms (i.e. two different opcode groups):

- The first form can load or store a 32-bit word or an 8-bit unsigned byte
- The second form can load or store a 16-bit unsigned halfword, and can load and sign extend a 16-bit halfword or an 8-bit byte

The first form (word and unsigned byte) allows a wider range of addressing modes than the second (halfword and signed byte).

3.1 Word and Unsigned Byte Addressing Modes

The Word and Unsigned Byte addressing modes utilize two elements:

- base register
- offset

3.1.1 Base Register

The base register is any one of the general-purpose registers (including the PC, which allows PC-relative addressing for position-independent code). Commonly the base register is loaded with the beginning address of the array, string or structure being accessed. The address may be loaded from a label using either of the two pseudo instructions:
ADRL r0, label
LDR r0, =label

These pseudo instructions are actually implemented as a pair of ARM instructions each moving a portion of the address.

3.1.2 Offsets

The offset takes one of three forms:

- immediate offset
- register offset
- scaled register offset

3.1.2.1 Immediate Offset:

The offset is a 12-bit unsigned number, which is added to or subtracted from the base register. Immediate Offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets and IO registers.

Example:

```c
struct database {
    char name[20];
    int age;
    int ssn;
    int phone;
}
```

The offset to the start of each field is 0, 20, 24 and 28 respectively. To access the “ssn” field, the immediate offset would be 24 with the base register pointing to the start of the structure.

3.1.2.2 Register Offset:

The offset is a general-purpose register (not the PC), which is added to or subtracted from the base register. Register offsets are useful for accessing arrays or blocks of data.
Example:

    int array[20], index, value;
    index = 5;
    value = array[index];

The base register would be set to the beginning of the array. The register holding the offset value would contain the value of the index. (Actually, this would need to be the value of the index multiplied/shifted by the size, in bytes, of the array element, which in this case is 4.)

3.1.2.3 Scaled Register Offset:

The offset is a general-purpose register (not the PC) shifted by an immediate value, then added to or subtracted from the base register. The same shift operations used for data-processing instructions can be used (Logical Shift Left, Logical Shift Right, Arithmetic Shift Right and Rotate Right), but Logical Shift Left is the most useful as it allows an array indexed to be scaled by size of each array element.

Example:

    int array[20], index, value;
    index = 5;
    value = array[index];

In the same example as before, the register holding the offset value would contain the value of the index. The scaling operation would shift the value “index” by the size of the array element, which in this case is a shift by 2 (i.e. a multiply by 4 for 4 byte length). The base register again contains the start address of the array.

3.1.3 Pre/Post Addressing

As well as the three forms of offset, the offset and base register are used in three different ways to form the memory address.

- offset addressing
- pre-indexed addressing
- post-indexed addressing
3.1.3.1 Offset addressing:

The base register and offset are simply added or subtracted to form the memory address.

Examples:

```
LDR r0, [r1, #24] ;Immediate offset
LDR r0, [r1, r2] ;Register offset
LDR r0, [r1, r2, LSL #2] ;Scaled register offset
```

3.1.3.2 Pre-indexed addressing:

The base register and offset are added or subtracted to form the memory address. The base register is then updated with this new address, to allow automatic indexing through an array or memory block.

Examples:

```
LDR r0, [r1, #24] ! ;Immediate offset with r1 updated by 24
LDR r0, [r1, r2] ! ;Register offset with r1 updated by r2
LDR r0, [r1, r2, LSL #2] ! ;Scaled register offset with r1 updated by r2
```

3.1.3.3 Post-indexed addressing:

The value of the base register alone is used as the memory address. The base register and offset are added or subtracted and this value is stored back in the base register, to allow automatic indexing through an array or memory block.

Examples:

```
LDR r0, [r1, #24] ! ;Base register address with r1 updated by 24
LDR r0, [r1, r2] ! ;Base register address with r1 updated by r2
```

3.1.4 Enumerated Word and Unsigned Byte Addressing Modes

There are nine addressing modes which are used to calculate the address for a load and store word or unsigned byte instruction. Each addressing mode is summarized below:
Immediate Offset

\[
\text{LDR|STR\{cond\}\{B\} \ Rp, [Rn, +/-<12\_bit\_offset>]}
\]

Register Offset

\[
\text{LDR|STR\{cond\}\{B\} \ Rp, [Rn, +/-Rm]}
\]

Scaled Register Offset

\[
\text{LDR|STR\{cond\}\{B\} \ Rp, [Rn, +/-Rm, <shift> #<shift\_imm>]}
\]

Immediate Pre-indexed

\[
\text{LDR|STR\{cond\}\{B\} \ Rp, [Rn, +/-<12\_bit\_offset>] !}
\]

Register Pre-indexed

\[
\text{LDR|STR\{cond\}\{B\} \ Rp, [Rn, +/-Rm] !}
\]

Scaled Register Pre-indexed

\[
\text{LDR|STR\{cond\}\{B\} \ Rp, [Rn, +/-Rm, <shift> #<shift\_imm>] !}
\]

Immediate Post-indexed

\[
\text{LDR|STR\{cond\}\{B\} \ Rp, [Rn, +/-<12\_bit\_offset>}
\]

Register Post-indexed

\[
\text{LDR|STR\{cond\}\{B\} \ Rp, [Rn, +/-Rm}
\]

Scaled Register Post-indexed

\[
\text{LDR|STR\{cond\}\{B\} \ Rp, [Rn, +/-Rm, <shift> #<shift\_imm>}
\]

### 3.1.5 Additional Word and Unsigned Byte Addressing Examples

\[
\text{LDR R1, [R0] ; Load register 1 from the address in register 0}
\]
\[
\text{LDR R8, [R3, #4] ; Load R8 from the address in R3 + 4}
\]
\[
\text{LDR R12, [R13, #-4] ; Load R12 from R13 - 4}
\]
STR R2, [R1, #0x100]; Store R2 to the address in R1 + 0x100
LDRB R5, [R9]; Load a byte into R5 from R9 (zero top 3 bytes)
LDRB R3, [R8, #3]; Load byte to R3 from R8 + 3 (zero top 3 bytes)
STRB R4, [R10, #0x200]; Store byte from R4 to R10 + 0x200
LDR R11, [R1, R2]; Load R11 from the address in R1 + R2
STRB R10, [R7, -R4]; Store byte from R10 to the address in R7 – R4
LDR R11, [R3, R5, LSL #2]; Load R11 from R3 + (R5 x 4)
LDR R11, [R0, #4]; Load R1 from R0 + 4, then R0 = R0 + 4
STRB R7, [R6, -1]!; Store byte from R7 to R6 – 1, then R6 = R6 – 1
LDR R3, [R9], #4; Load R3 from R9, then R9 = R9 + 4
STR R2, [R5], #8; Store word from R2 to R5, then R5 = R5 + 8
LDR R0, [PC, #40]; Load R0 from PC + 8 + 0x40
LDR R0, [R1], R2; Load R0 from R1, then R1 = R1 + R2

3.2 Halfword and Signed Byte Addressing Modes

The Halfword and Signed Byte addressing modes are a subset of the above addressing modes. These addressing modes utilize the same two elements:

- base register
- offset

3.2.1 Base Register

The base register is again any one of the general-purpose registers (including the PC, which allows PC-relative addressing for position-independent code). Commonly the base register is loaded with the beginning address of the array, string or structure being accessed.

3.2.2 Offsets

The offset for halfword and signed byte addressing is restricted to one of two forms:

- immediate offset
- register offset

The scaled register offset is not supported, and the immediate offset contains 8 bits, not 12.

3.2.3 Pre/Post Addressing

The offset and base register are used in three different ways to form the memory address.

- offset addressing
• pre-indexed addressing
• post-indexed addressing

3.2.4 Enumerated Halfword and Signed Byte Addressing Modes

There are six addressing modes which are used to calculate the address for a load and store (signed or unsigned) halfword or load signed byte instruction. Each addressing mode is summarized below:

Immediate Offset

LDR|STR{cond}H|SH|SB Rp, [Rn, #+/-<8_bit_offset>]

Register Offset

LDR|STR{cond}H|SH|SB Rp, [Rn, +/-Rm]

Immediate Pre-indexed

LDR|STR{cond}H|SH|SB Rp, [Rn, #+/-<8_bit_offset>]

Register Pre-indexed

LDR|STR{cond}H|SH|SB Rp, [Rn, +/-Rm]

Immediate Post-indexed

LDR|STR{cond}H|SH|SB Rp, [Rn, #+/-<8_bit_offset>]

Register Post-indexed

LDR|STR{cond}H|SH|SB Rp, [Rn, +/-Rm]

3.2.4 Halfword and Signed Byte Addressing Examples

LDRH R1, [R0] ; Load a halfword to R1 from R0 (zero top bytes)
LDRH R8, [R3, #2] ; Load a halfword into R8 from R3 + 2
LDRH R12, [R13, #6] ; Load a halfword R12 from R13 - 6
STRH R2, [R1, #0x80] ; Store halfword from R2 to R1 + 0x80
LDRSH R5, [R9] ; Load signed halfword to R5 from R9
LDRSB R3, [R8, #3] ; Load signed byte to R3 from R8 + 3
LDRSB R4, [R10, #0xc1] ; Load signed byte to R4 from R10 + 0xc1
LDRH R11, [R1, R2] ; Load halfword R11 from the address in R1 + R2
STRH R10, [R7, -R4] ; Store halfword from R10 to R7 - R4
LDRSH R1, [R0, #2]! ; Load signed halfword R1 from R0+2, then R0=R0+2
LDRSB R7, [R6, # - 1]! ; Load signed byte to R7 from R6-1, then R6=R6-1
LDRH R3, [R9], #2 ; Load halfword to R3 from R9, then R9 = R9 + 2
STRH R2, [R5], #8 ; Store halfword from R2 to R5, then R5 = R5 + 8

4. Program Assignment

The assignment will be developed using a C main program and an assembly language subroutine. These programs are a variation of the third example of Lab 1. Please refer to the both the C and assembly language code for Lab 1.

The programs together are to perform the following steps:

1) allocate three character strings in memory
2) print all three strings
3) copy the second string to the first string
4) copy the second string backwards to the third string
5) print all three strings

Steps 1, 2 and 5 are performed in the C program as given below. Steps 3 and 4 are to be implemented in your assembly language subroutine. In order to demonstrate your understanding of the addressing modes, you will use three different addressing modes in three different, but equivalent implementations of Steps 3 and 4.

The second string should be copied first in the forward direction into the first string so that the pointer to the end can be determined. This pointer to the end of the second string is then used for copying in the backward direction into the third string.

C/C++ indicates the end of a string by a byte whose value is 0 (i.e. a NULL byte). The third example of Lab 1 show a loop of code to copy the second string to the first string. It repeats the byte copy until a byte of zero is moved. This ends the copy loop.

There are several ways to copy the second string in reverse order to the third string forming a loop. Copying the second string to the first string simply monitoring for the NULL byte as the string is copied. The number of characters copied may be counted for the benefit of the copy of the second string reversed to the third string. Alternatively, the initial address of the second string may be retained for comparison as the pointer is decremented for the reverse order copy. Finally, the third pointer may be moved forward to the end of the string (to the NULL byte) and then the copy from the second to the third may be performed incrementing the pointer to the second string and decrementing the third pointer. You may choose to use any of these methods. You MAY NOT USE a NULL byte BEFORE the second string to stop your copying as thisis
only coincidental for this program because of the manner and order of the data allocation.

The main program file, “lab2.c”, is the following:

```c
#include <stdio.h>
extern void reverse (char *d, const char *s, Char *e);

int main ()
{
    char one[]= " I love to code asm";    /* Each string is 19 characters */
    char two[]="A British computer?";     /* Do not deviate */
    char three[]="Where's my A?????????"; /* Or the code may not work */

    printf ("before reverse subroutine:\n");
    printf (" %s \n %s \n %s \n", one, two, three);

    reverse (one, two, three);    /* Calls "reverse" subroutine and places the */
                               /* start address of arrays one, two, and three */
                               /* in r0, r1, & r2 respectively */

    printf ("\nAfter reverse subroutine:\n");
    printf (" %s \n %s \n %s \n", one, two, three);
    return (0);
}
```

You will implement the reverse routine using 3 different forms of addressing modes. Write each version in a separate file. The basic assembly language file, “lab2a.s”, (where a is a, b or c for each of the three versions) will have the following structure:

```
AREA     Rerun, CODE, READONLY
EXPORT   reverse

reverse
    ;r0 points to the first destination string
    ;r1 points to the second source string
    ;r2 points to the third destination string

LDRB r3, [r1], #1 ;load byte and update pointer address
STRB r3, [r0], #1 ;store byte and update pointer address
------- ;first copy loop in forward direction
------- ;second copy loop in backward direction
MOV pc,lr ;return from subroutine
```
Assemble and link your program using the following commands:

    armcc lab2.c
    armasm lab2a.s
    armlink lab2.o lab2a.o -o lab2a.axf

Again, the ‘a’ in “lab2a” is the letter a, b or c, representing each of the three versions of your program.

5. Results

Test and debug your program using AXD as directed in Lab 1. Demonstrate to your TA the implementation of each of your three programs. We suggest the use of 1) Immediate Offset addressing, 2) Register Post-indexed addressing and 3) Register Offset addressing. You may rely on the strings all having exactly 20 characters for this assignment only.

Remember, you must develop and demonstrate 3 separate programs using different address modes for copying the strings. At least one of these solutions must use a loop. You may not look for a NULL byte BEFORE the second string to stop your copying in the reverse direction.