COEN 311

Lab 3: Addressing Modes and 2-D arrays

T. Obuchowicz

2023

OBJECTIVES

• To obtain a strong understanding of the various addressing modes used in ARM assembly language.

• To gain familiarity with the basic arithmetic assembly language instructions.

• To apply one's knowledge of addressing modes and arithmetic operations to write an assembly language program which accesses the elements of a two dimensional array of integers which are stored in memory.

• To gain an appreciation of the task performed by a high level programming language compiler.

INTRODUCTION

Various different addressing modes exist within assembly language programming. An addressing mode specifies the method in which the instruction obtains the data operand. The first two labs have introduced the immediate and register indirect modes. This lab will explore several other available modes. We first review the basic modes presented in labs 1 and 2.

Immediate Mode

The source operand is contained within the instruction itself. Thus, once the instruction has been fetched from main memory and is residing within the CPU, it is immediately available for use. There is no further need to access main memory to obtain the operand.

Example: `mov r1, #5`

Register Mode

The operand is stored within a specified CPU register.

Example: `mov r1, r2`

Register Indirect Mode

A register holds the main memory address of the operand. Unlike the immediate and register modes, a register indirect operand access involves a memory access.

Example:

```
.data
list: .byte 1,2,3,4,5
.text
ldr r1, =list   @ register r1 holds the starting address
                @ of label 'list'
ldrb r2 , [r1]  @ register r2 will be loaded
                @ with 1 (fist element of the list)
```

It is often said that a picture is worth a thousand words, (or $2^{10} = 1024$ words); Figure 1 illustrates the concepts of the above three addressing modes. It is assumed in Figure 1, that the starting address of the list is 1000.

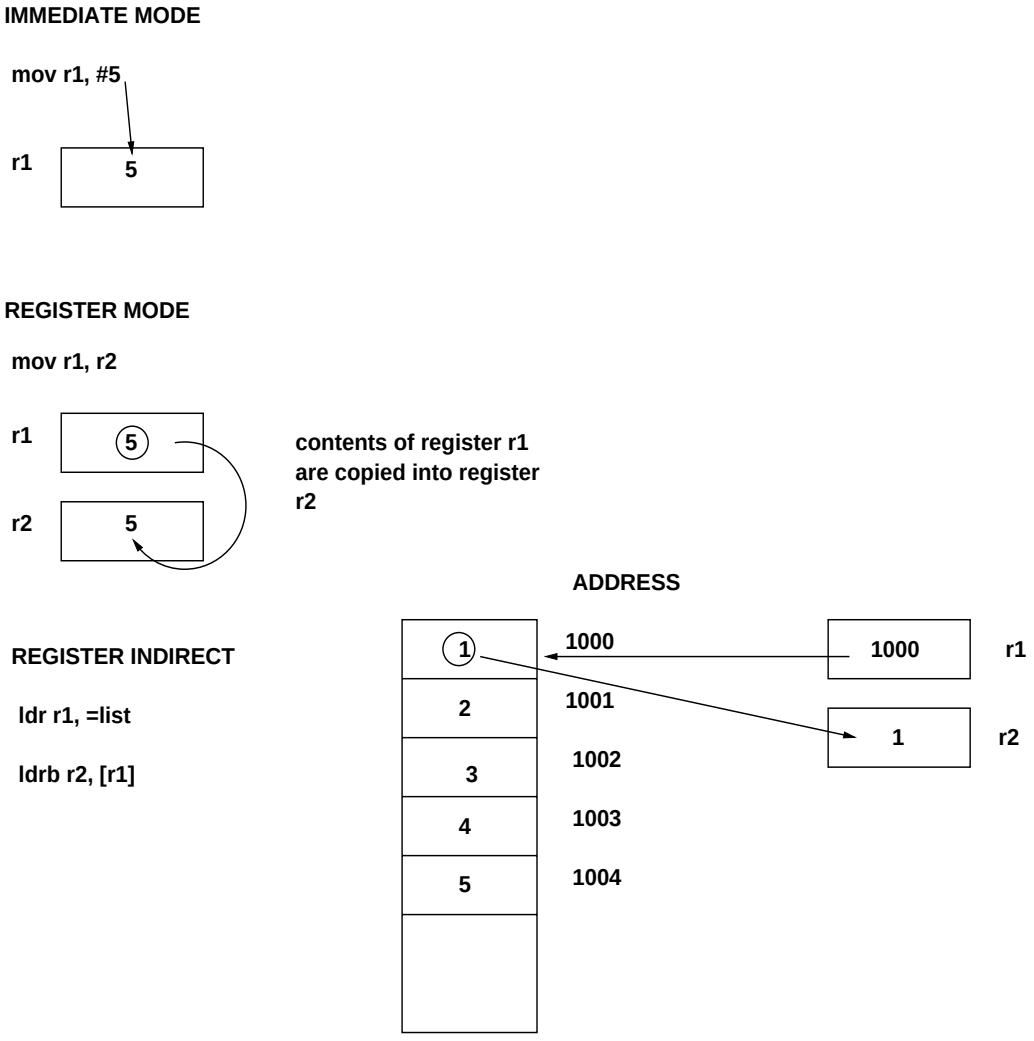

Figure 1: Immediate, register, and register indirect addressing modes.

The astute reader may have noted a connection between the register indirect addressing mode and a pointer deference operation in a high-level programming language (such as C++). Consider the following C++ pointer declaration and dereference operation:

```
int keith = 5 ;
int mick;
int* ptr;

ptr = &keith;
mick = *ptr;
```

A compiler would ultimately translate the pointer deference into a sequence of assembly language statements in which a register is loaded with the address of variable `keith`, and then register indirect mode is used to access the data at this address:

```
ldr r1, =keith  @ load address of keith into register r1
ldr r2, [r1]    @ load the data pointed to by r1 into r2
```

There are several other addressing modes which are variations of the basic register indirect mode.

Register indirect with immediate offset

A specified register holds the address of the memory operand. A specified offset value is added to the contents of the register to yield the *effective memory address*. It should be noted that the addition of the offset value to the register, does **not** change the final contents of the register.

Example:

```
list: .byte 1,2,3,4,5

ldr r1, =list     @ register r1 holds the starting address
                  @ of label 'list'
ldrb r2, [r1, #1] @ r2 will be loaded with the data
                  @ stored at address 'list + 1'
                  @ r2 loaded with 2
                  @ r1 still points to address 'list'
                  @ r1 is NOT MODIFIED
```

Figure 2 illustrate the register indirect with immediate offset addressing mode.

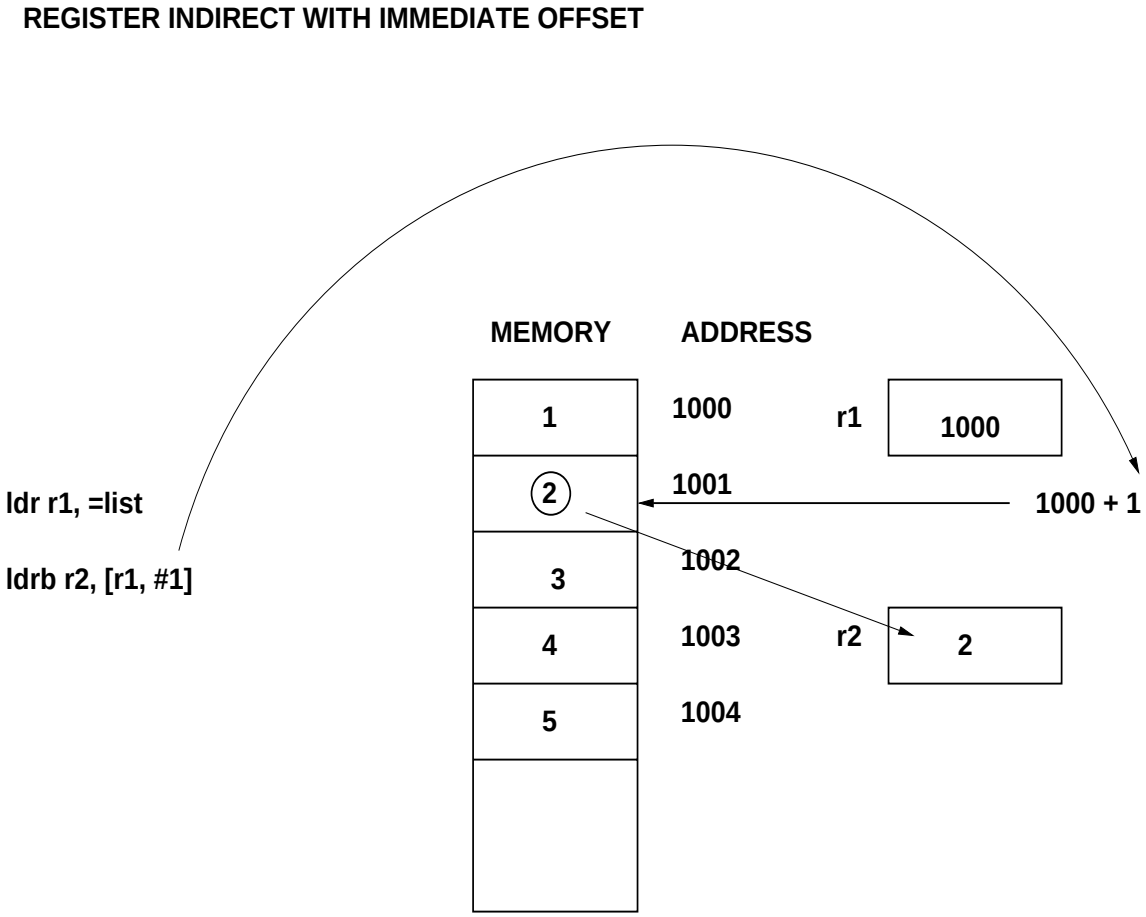

Figure 2: Register indirect with immediate offset mode.

Register indirect with register offset

In this mode, a second register holds the offset value to be added to the first register holding the address of the memory operand. The addition does not modify the value of the register.

Example:

```
list: .byte 1,2,3,4,5
ldr r1, =list     @ register r1 holds the starting address
                  @ of label 'list'
mov r2, #2        @ load r2 with some offset value
ldrb r3, [r1,r2]  @ register r3 loaded with data from
                  @ address 'list + 2'
                  @ r3 loaded with 3
                  @ neither r1 not r2 is modified
```

Figure 3 illustrates this mode.

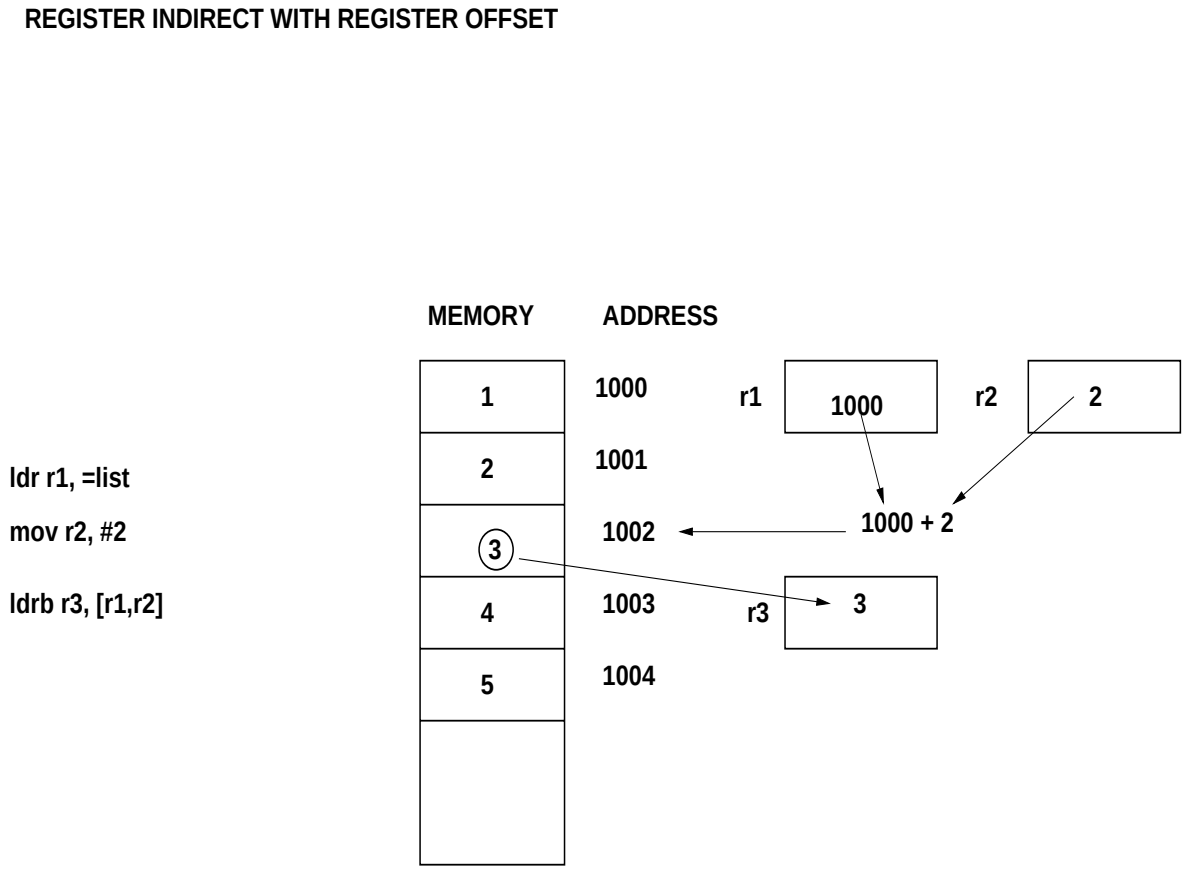

Figure 3: Register indirect with register offset.

Post increment mode with immediate offset

It is sometimes useful (when working with arrays for example), to have the register which points to a memory operand be automatically incremented to point to the next array element once the current element has been accessed. This is exactly what post increment mode with immediate offset mode performs. In the parlance of ARM, this mode is referred to as ‘’post-index with write back’. It uses a somewhat unwieldy syntax:

```
list: .byte 1,2,3,4,5
ldr r1, =list     @ register r1 holds the starting address
                  @of label ’list’
ldrb r2, [r1], #1 @ r2 loaded with data from address ’list’ (1)
                  @ AND THEN
                  @ the immediate #1 is ADDED to r1,
                  @ so r1 now points to the
```

```
                @ second item in the list (data value 2)
ldrb r3, [r1], #1 @ r3 is loaded with value 2, and then
                @ 1 is added to r1
```

Figure 4 gives the details of this mode.

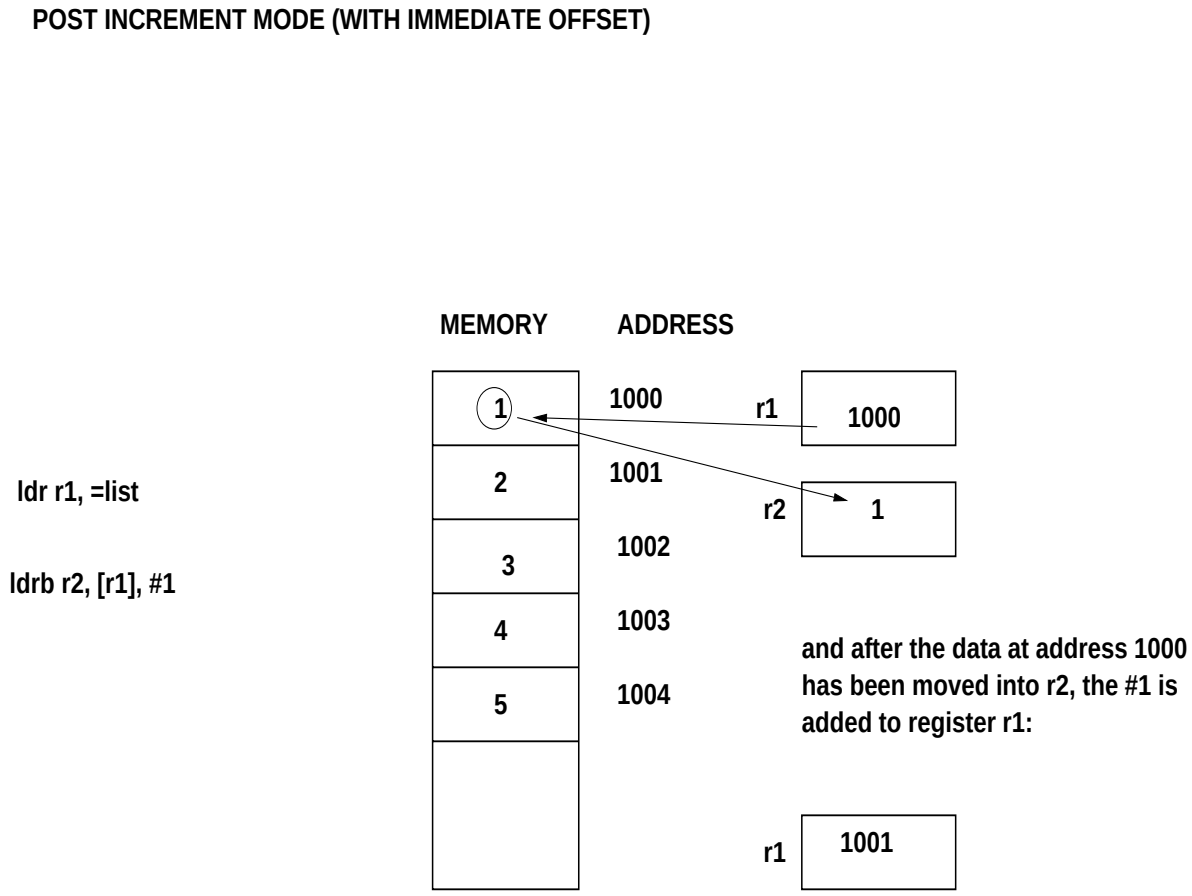

Figure 4: Post increment mode with immediate offset.

Predecrement mode with immediate offset

This mode is similar to postincrement mode, except that the offset is added FIRST to the specified register. The offset can be a negative value. This mode is useful for accessing the elements of an array in reverse order. It has an even more unwieldy syntax:

```
list: .byte 1,2,3,4,5
ldr r1, =list + 5     @ register r1 holds the address of
                      @ memory AFTER the last element
```

```
ldrb r2, [r1, #-1]!   @ r2 loaded with date from r1 + (-1),
                      @ (ie. value 5)
                      @ and then r1 becomes  r1 + (-1) ,
                      @( r1 now points to data value 4)
ldrb r3, [r1, #-1]!   @ r3 will be loaded with data value 4
                      @ and then r1  modified to point
                      @ to r1 + (-1)
```

This mode is illustrated in Figure 5.

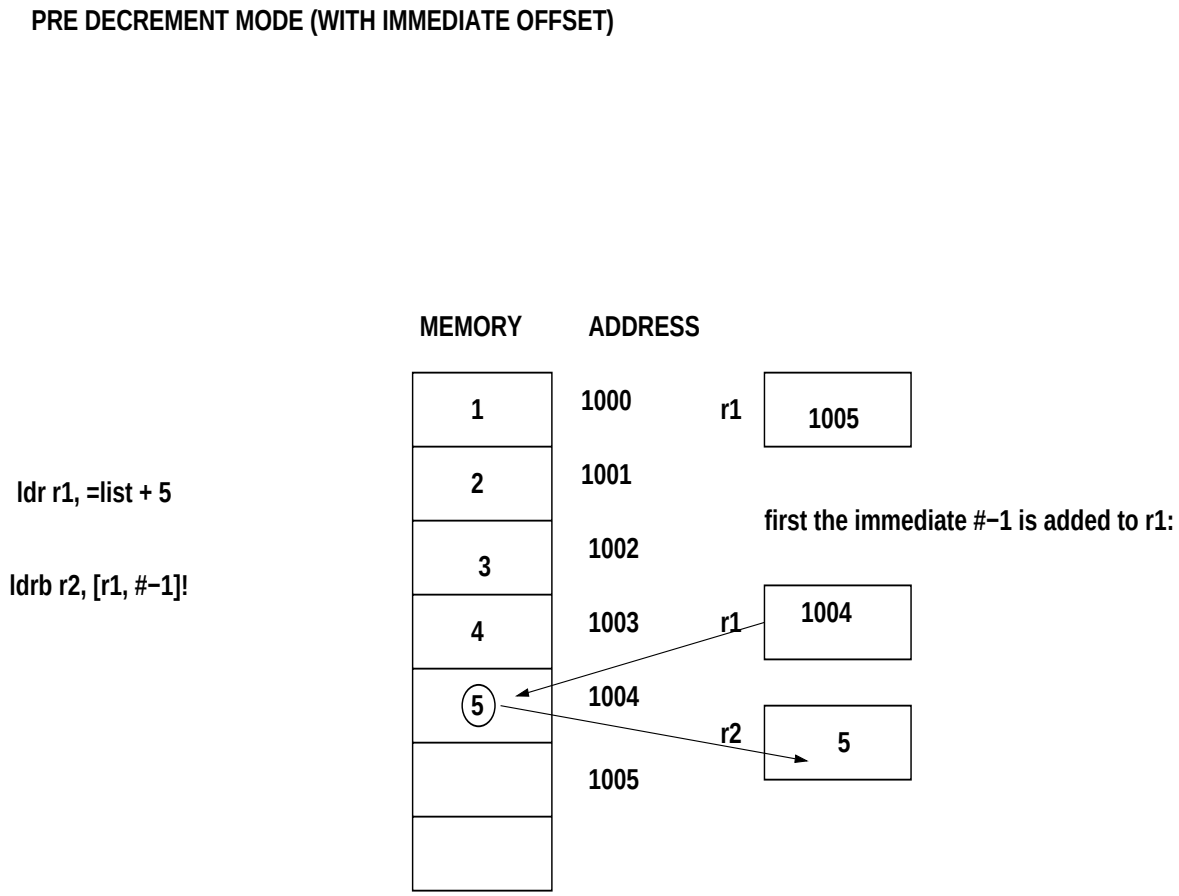

Figure 5: Predecrement mode with immediate offset.

Other variations of the post/preincrement modes exist, but they are not supported by the Thumb instruction set variant of the ARM assembly language which is used in the labs. The following is a complete ARM assembly language program which makes use of the aforementioned addressing modes. The interested reader is advised to single-step through this program using gdb to further ones's understanding.

```
@ Ted Obuchowicz
@ May 9, 2023
@ addressing_modes_thumb.s

.syntax unified
.cpu cortex-m4
.thumb

.word 0x20000400
.word 0x800000ed
.space 0xe4

.data
@ declare your data here
list: .byte 1,2,3,4,5
.text
start:

@ IMMEDIATE MODE
mov r1, #5   @ register r1 will be loaded with the immediate data 5

@ REGISTER MODE
@ the operand is contained in the specified register
@ sometimes referred to as “register direct”

mov r2, r1  @ register r1 contains the data operand 5 (from the
            @ previous instruction, the contents of r1
            @ will be copied into r2
            @ general form mov rdest , rsource


@ REGISTER INDIRECT MODE
@ a register is used to hold the ADDRESS of some data operand

ldr r1, =list   @ register r1 holds the starting address
                @ of label ‘list’
ldrb r2 , [r1]  @ register r2 will be loaded with 1
                @(first element of the list)

@ REGISTER INDIRECT WITH  IMMEDIATE OFFSET

ldr r1, =list     @ register r1 holds the starting address of
                  @label ‘list’
ldrb r2, [r1, #1] @ r2 will be loaded with the data stored at
                  @ address ‘list + 1’
                  @ r2 loaded with 2
```

```
                @ r1 still points to address ‘list’
                @ r1 is NOT MODIFIED

@ REGISTER INDIRECT WITH REGISTER OFFSET
@ similar to above, but a second register is used to hold
@ the offset value

ldr r1, =list     @ register r1 holds the starting address
                  @ of label ‘list’
mov r2, #2
ldrb r3, [r1,r2]  @ register r3 loaded with data from
                  @ address ‘list + 2’
                  @ r3 loaded with 3
                  @ neither r1 not r2 is modified

@ POST INCREMENT MODE (WITH IMMEDIATE OFFSET)
@ this is a variation of register indirect in which the register
@ which holds the address of the operand is MODIFIED after.
@ this is called ‘post-index with write back’ in ARM parlance

ldr r1, =list     @ register r1 holds the starting address
                  @ of label ‘list’
ldrb r2, [r1], #1 @ r2 loaded with data from address ‘list’ (1)
                  @ AND THEN
                  @ the immediate #1 is ADDED to r1, so r1
                  @ now points to the
                  @ second item in the list (data value 2)

ldrb r3, [r1], #1 @ r3 is loaded with value 2, and then
                  @ 1 is added to r1

@ this mode is useful for sequentially accessing the elements of
@ an array in a loop



@PREDECREMENT MODE (WITH IMMEDIATE OFFSET)
@ similar to postincrement mode, except that the
@ offset is added FIRST to the specified register
@ the offset can be a negative value
@ useful for accessing the elements of an array in reverse order

ldr r1, =list + 5     @ register r1 holds the address of
                      @ memory AFTER the
                      @ last element
ldrb r2, [r1, #-1]!   @ r2 loaded with date from r1 + (-1),
```

```
                    @(ie. value 5)
                    @ and then r1 <= r1 + (-1) ,
                    @ (r1 now points to data value 4)
ldrb r3, [r1, #-1]!   @ r3 will be loaded with data value 4 and then
                    @ r1  modified to point to r1 + (-1)


@ other variations of the post/preincrement modes exist,
@ but they are not supported by the Thumb instruction set
@ which is used in the labs.

stop:   b stop
.end
```

Consider the C++ declaration of a two-dimensional (2-D) array of integers:

```
   int my_array[4][3] = { 1,2,3,
                          4,5,6,
                          7,8,9,
                          10,11,12 } ;
```

In a high-level programming language such as C++, one makes use of array notation to access a particular array element. For example:

```
int my_element ;

my_element = my_array[3][2] // assign array element with value 12
                            //  to integer variable my_element
```

The use of the `[][]` array index operators represents a level of abstraction, which is also known as "let the compiler do the dirty work of figuring out where in main memory the particular array element is stored at". Recall, that a compiler first converts the high-level code into assembly language, then invokes the native assembler which converts the assembly code to machine code. Ultimately, all we have to work with is the CPU instruction set.

Most introductory programming courses introduce the concept of *array address translation* formulas. The elements of a (two dimensional or higher) array are stored row by row (row major form) in main memory. Main memory is abstracted as a 1-dimensional array of locations. The 2-dimensional array address translation formula is:

a[i][j] is stored at : (starting address) + [**(** (i * number of columns) + j **)** * sizeof(data_type)]

In C++, the name of an array is synonymous with the starting address in main memory where the first element of the array is stored. Thus, the above formula can be succinctly stated as:

a[i][j] is found at address : a + [**(** (i * number of columns) + j **)** * sizeof(data_type)]

Consider again the previous 2-D array C++ declaration (annotated with row and column numbers):

```
//                    Col 0  Col 1  Col 2
int my_array[4][3] = { 1,     2,     3,     // Row 0
                       4,     5,     6,     // Row 1
                       7,     8,     9,     // Row 2
                       10,    11,    12 } ; // Row 3
```

Figure 6 shows how this array would be stored row-by-row in main memory (assuming the array is stored at starting address 1000).

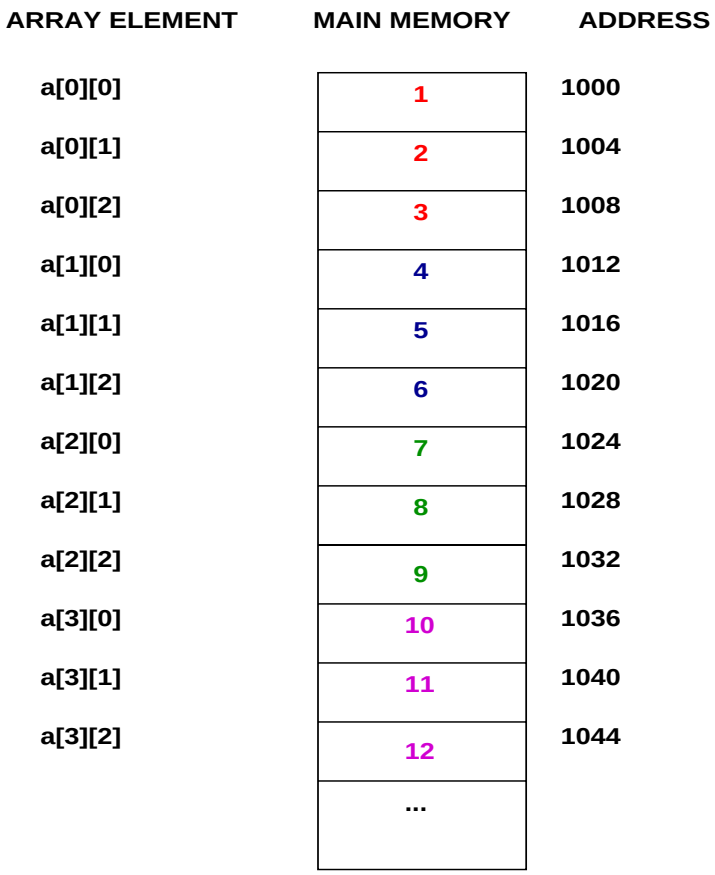

Figure 6: 2-D array stored in memory.

Note in Figure 6, since each element of the array is an integer, four consecutive memory bytes are used to hold each integer. The starting address of each array element is indicated in the figure (in multiples of 4 bytes). The array address translation formula used by the compiler whenever it encounter array notation in the program code is:

a[i][j] = a + (((i x 3) + j) x sizeof(int)) (where 3 is the number of columns per row)

Let us use this formula to show how the compiler translates a[1][2] into the memory address containing the first byte of the integer element in position a[1][2] of the array:

a[1][2] = a + (((1 x 3) + 2) x 4)
= 1000 + ((3 + 2) x 4)
= 1000 + 20
= 1020

Convince yourselves that the 2-D array address translation formula properly converts from a[i][j] array notation into a starting main memory address containing the array element. It is for the these reasons that when we pass a 2-D array to a C++ function, we need to **specify the number of columns in the array, but not the number of rows**. For example:

```
void some_function(int some_array[][3])
{
  // whenever in the function, the programmer uses array notation
 //  similar to some_array[i][j], the compiler translates this
 //  to address: some_array +  ( ( (i * 3 ) + j ) * sizeof(int) )
}
```

PROCEDURE

Write an ARM assembly language program which makes use of the 2-D array translation formula to access the array elements making use of the register indirect addressing mode (or more specifically "register indirect with register offset" mode which is just a variation of register indirect mode).

Use the following section `.data` in your program:

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

@ 3 rows and 2 columns
@ array =  3 2
@          4 1
@          5 6
```

Use one register to hold the starting address of the array, and use another register which holds the "offset" of the element. You may "hardwire" two registers 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
```

Make use of the `mul` and `add` instructions to compute the offset value. Load the desired array element into a register.

Test your program using ‘gdb’ for **several** typical values of row and column indices to ensure your program is working correctly. For the purposes of this lab, it is sufficient to merely edit your source code and put new values into the row and column indices , re-assemble and re-ld and re-run with gdb. It is not necesary to use loops to iterate over each array element.

QUESTIONS

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

```
@ Ted Obuchowicz
@ May 23, 2023
@ lab3_question.s

.syntax unified
.cpu cortex-m4
.thumb

.word 0x20000400
.word 0x800000ed
.space 0xe4

.data
@ declare your data here
mydata:   .word  0xdeadbeef
address_of_mydata: .word mydata

.text
start:
        @  assembly code goes here
        ldr r1, =address_of_mydata
        ldr r1, [r1]
        ldr r2, [r1]

stop:   b stop
.end
```

Express your answer as a 8 digit hexadecimal number.

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

```
15 0000 EFBEADDE      mydata:   .word  0xdeadbeef
```

Comment on the ordering of the bytes. Is the ARM processor contained within the STM32F334 Nucleo microcontroller board used in the labs an example of a *big-endian* or *little-endian* processor? The *endianness* of a processor refers to how multiple byte data is stored by a processor in a byte sized memory. Big-endian processors store the highest-order byte at address X, followed by the second-highest order byte at address X + 1, etc. Little endian processors strore the lowest order byte at address X, followed by the second lowest order byte at address X + 1, etc.

T. Obuchowicz
May 24, 2023