Microprocessor and Assembly Language Lab: Lab 6

June 17, 2025

1 Introduction

In high-level languages we have any number of techniques available for constructing data structures. These include single-, double-, and higher-dimensional arrays, records (or structs), bit sets, and pointers for linked lists, trees, etc. In this lab we start thinking about using arrays of numbers in memory. On the ARM, any of the sixteen 32-bit integer registers may be used as a base address register, including the program counter. We use this fact without really thinking about it when we set up "simple" LDR or STR instructions; in reality such memory references use the program counter R15 as the hidden base address to an array. Since ARM instructions are 32 bits wide, and so are absolute addresses, it is impossible to embed a complete address into an instruction. Memory reference instructions on the ARM reserve only 12 bits for the constant address offset, plus one more bit to determine whether to add or subtract the offset from the specified base register. Thus, all memory references are limited to a constant range of ±4096 bytes relative to some base register (for "simple" loads and stores this means that data words can't be more than ±4096 bytes away from the instructions that reference them).

- Let's build and use an array of five 32-bit words. Declaring the data structure is pretty easy; all that is required is the defined symbol for the first word of the array and enough data declarations to reserve all the space we need, as shown in **figure** 1.

  Buffer DCD 0,0,0,0,0

  The DCD directive reserves one word for each of the cells and initializes it to zero. We think of the first word as the initial element of the array, or Buffer[0], followed four bytes later by Buffer[1]. The five words are laid out in memory as **figure** 2:

  Now we need to know the address of where Buffer actually starts, and it can be anywhere in memory. Since we don't know where it is in memory, we cannot simply write LDR Buffer to get Buffer[0] and LDR Buffer+4 to get Buffer[1], because in the general case the starting address of Buffer isn't within 4K bytes of our instructions. What we can do, however, is ask the assembler to place the base address of Buffer into a memory

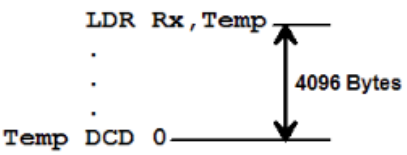

Figure 1:

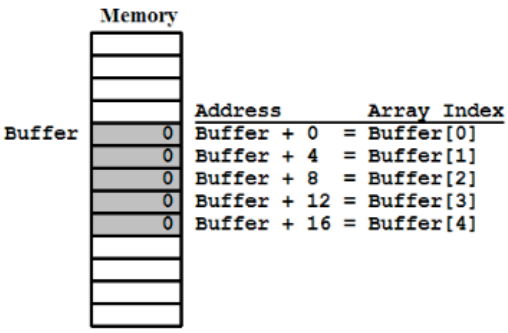

Figure 2:

slot which is close by, and then we write the code to load the contents of that word into the desired register. This is done as follows:

LDR Rx,Temp

.

.

.

Temp DCD Buffer

Variable Temp is initialized through the DCD directive to contain the complete 32-bit absolute address of Buffer, which will be loaded into the desired register as long as Temp is within 4K bytes of the LDR instruction. In the ARM assembler, this combination happens so frequently that there is a special directive to help the process. By typing:

ADR Rx,Buffer

Here, the assembler is getting permission to do whatever is necessary to get the address of Buffer into register Rx. For the following examples, we will put the address of Buffer into register R5, as shown:

ADR R5,Buffer

To reference memory relative to a given register other than the program counter you specify the name of the register in square brackets. Since Buffer[0] is at address Buffer+0, loading the contents of Buffer[0] into R0 is done by the instruction:

LDR R0,[R5]

Each of the other array elements is accessed by including its offset in bytes inside the square brackets. Remember that the offsets must be multiples of 4 since we are loading 32-bit words, and that addresses for the LDR and STR instructions must be divisible by four as well. (There are instructions for getting individual bytes out of memory, LDRB and STRB.)

While this is very useful, we still need to specify a variable offset so that a program can step through the individual elements of an array regardless of how many there are. A register may be used instead of a constant, as **figure** 4:

```
LDR R0,[R5]          R0 := Buffer[0]
LDR R0,[R5,#0]       R0 := Buffer[0]
LDR R0,[R5,#4]       R0 := Buffer[1]
LDR R0,[R5,#8]       R0 := Buffer[2]
LDR R0,[R5,#12]      R0 := Buffer[3]
LDR R0,[R5,#16]      R0 := Buffer[4]
```

Figure 3:

```
LDR R0,[R5,R1]       R0 := Buffer[R1÷4]
STR R0,[R5,R1]       Buffer[R1÷4] := R0
```

Figure 4:

In this approach, any value in R1 must be a multiple of 4 (the data storage size). This means that R1 does not contain a true array index, but instead it contains the memory offset of the desired array element, which is the array index multiplied by the data storage size. The following code **figure** 5 shows how we would store zero into every element of the array using this approach.

This approach does work, but we must use "strange" constants 16 and 4 for the initial offset and decrement values, respectively, and we keep track of the fact that R1 contains a value four times the value of the array index. Rather than force R1 to contain memory offsets, we can write the code so that R1 contains a true array index, but only if we multiply R1 by the storage size each time we reference memory. By using the barrel shifter, we can write the following:

LDR R0,[R5,R1,LSL #2]
STR R0,[R5,R1,LSL #2]
R0 := Buffer[R1]
Buffer[R1] := R0

These instructions form the effective address of the word to load or store by multiplying the array index in R1 by 4 (through the LSL #2 directive) to form the memory offset, and then adding that offset to the base address of the array in R5. Doing so does not change the contents of R1. Since R1 now contains a true array index, a loop that steps through every element of the array will increment or decrements R1 by 1, not by 4. Here's the final example of storing zero into every element of the array:

This more closely matches what we are used to seeing in high-level languages. In our examples we use R5 to contain the base address of the array and R1 as the array index.

```
        ADR  R5,Buffer        array base address
        MOV  R0,#0            value to store
        MOV  R1,#16           last array offset
Loop1                         Repeat
        STR  R0,[R5,R1]         Buffer[R1÷4] := 0
        SUBS R1,R1,#4           R1 := R1 - 4
        BPL  Loop1            Until R1 < 0
```

Figure 5:

```
        ADR  R5,Buffer          array base address
        MOV  R0,#0              value to store
        MOV  R1,#4              last array index
Loop1                           Repeat
        STR  R0,[R5,R1,LSL #2]    Buffer[R1] := 0
        SUBS R1,R1,#1             R1 := R1 - 1
        BPL  Loop1              Until R1 < 0
```

Figure 6:

2 Your Task

1. Write an Cortex-M4 assembly language to show the first and last element of an Array.
2. Write an Cortex-M4 assembly language to perform the summation of the elements of an Array.
3. Write an Cortex-M4 assembly language to reverse an Array.
4. Write an Cortex-M4 assembly language to find the largest element of an Array.
5. Write an Cortex-M4 assembly program that computes the element-wise sum of two equal length arrays and store the result in a third array.

3 Submission Guideline

Submission Guideline

1. Your Assembly code with proper comments. (*.s file)
2. A document (*.tex file) that contains:
3. Detail explanation of the code
   - Screenshot that shows the state of the system after the code has been loaded.
   - Screenshot that shows the situation after the code has been executed.
   - Submit as a .zip file. Example: your classroll_lab#.zip (12_lab6.zip)