Lecture #17 – March 22, 2004

Arrays

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 some languages arrays start with index 0 (C, C++, Java), in others arrays start with index 1 (BASIC, FORTRAN), and in others still arrays may start at any index chosen by the programmer (Pascal, Delphi). At the assembly language level all of these approaches map onto single-dimension arrays alone. If we wish to emulate one of the other structures, then we must figure out how to create a ***mapping function*** which will convert the appropriate high-level access method into a simple offset into a linear list of memory bytes.

ONE-DIMENSIONAL

The simplest case is that of the single-dimension array, as such arrays in high-level languages closely match their equivalent structures in assembly language. For example, the ARM assembly language declaration `Buffer DCD 0,0,0,0,0` allocates five 32-bit words in memory and defines the symbol `Buffer` to have as its value the starting address of the first word. In a zero-based array system the first element is `Buffer[0]`, and the address of that first element is at address `Buffer+0`. For a representation where each memory word is four bytes in length, the second element `Buffer[1]` is at address `Buffer+4`, the third element `Buffer[2]` is at `Buffer+8`, and so on. This is shown below:

Array Index	**Memory Offset**
`Buffer[0]`	`Buffer+0`
`Buffer[1]`	`Buffer+4`
`Buffer[2]`	`Buffer+8`
`Buffer[3]`	`Buffer+12`
`Buffer[4]`	`Buffer+16`

To reference any element in an array we need to have both the starting address of the array (the ***base address***) and the index of the desired element. On the ARM, the base address of an array must be in a register. The easiest method for getting the address into a register in ARM assembly language is to use the `ADR` pseudo-instruction. In the simplest case the assembler will generate a `MOV` instruction with the correct (program counter relative) offset to the array, as long as the memory for the array is relatively close to the point of reference. For more distant offsets there are other approaches, but in each case the base address of the array will be placed into the specified register. For example, the (pseudo) instruction `ADR R5,Buffer` places the address of `Buffer` into register `R5`.

The array index value must be multiplied by the size of the array elements before it is added to the base address; if the array element size is a power of two, then we can combine the multiplication and the addition into a single instruction. In our example, array elements are four bytes in size; if the array index (a number between 0 and 4) is in register `R1` then loading some register `Rx` from `Buffer[R1]` is accomplished by the following code:

```
ADR  R5,Buffer
LDR  Rx,[R5,R1,LSL #2]
```

Note that the contents of `R1` are multiplied by four (the `LSL #2` attribute) before being added to the base address in `R5`, but `R1` is not modified by the execution of this instruction. On the ARM, 32-bit words must be aligned to 4-byte memory addresses, and the assembler will insure that `DCD` directives start at 4-byte addresses as well. If you use a shift value of 0 or 1, such that the final effective address is not a multiple of four, then a run-time exception will be generated.

Note too there are no provisions in this code for range checking. It does not matter if the value in `R1` is less than zero or greater than four; the word at the computed offset will be loaded into `Rx` regardless of what it contains. Worse, still, is executing an `STR` instruction, as memory outside the bounds of the array will be written over.

For byte-oriented arrays declared as `Buffer DCB 0,0,0,0,0` the instruction to load an array element (using the opcode `LDRB` instead of `LDR`) does not contain any shifting. The equivalent code for referencing a byte array is as follows:

```
ADR  R5,Buffer
LDRB Rx,[R5,R1]
```

NON ZERO-BASED 1D

In languages such as Pascal, the first array element can be at any index value, and need not always start at zero. (In my own Pascal code, I use zero-based arrays only about 30% of the time.) Given defined constants `Low` and `High`, which can have any arbitrary integer values so long as `Low` is less than `High`, the following Pascal array definition is perfectly legal:

```
Buffer : Array [Low..High] Of Integer ;
```

It does not matter if `Low` and `High` are positive, negative, or zero, or any combination as long as `Low` ≤ `High`. (Pascal allows any index type to be used here as long as it maps onto the integers, including Booleans, characters, or user-defined scalar types.) To map this array onto the equivalent assembly language we need to know only the total number of elements (`High - Low + 1`), the starting index (`Low`), and the size of each array element (4 bytes each for simple integers). The total number of elements, as well as the overall array size, can be computed at assembly time.

In ARM assembly language an arbitrary number of memory bytes are reserved by using the `%` directive. For example, reserving 40 bytes for an array called `Buffer` is accomplished by the directive `Buffer % 40`, and those bytes are initialized to zero at assembly time. In 8088 assembly language the same task is accomplished by the directive `Buffer DB 40 DUP (0)`. Thus, the complete ARM assembly language definition for the Pascal array will be as follows:

```
Low       EQU       ____          pick any integer
High      EQU       ____          pick any integer
Elements  EQU       High-Low+1
Buffer    %         Elements*4
```

Any index between `Low` and `High` must be mapped into the appropriate offset between `0` and `Elements - 1` before it can be applied to the base address of the array. The mapping function need only subtract `Low` from the array index value to generate the appropriate zero-based offset.

Range checking, if desired, may appear either before or after the mapping takes place; if before, the index must be greater than or equal to `Low` and less than or equal to `High`, and if after it must be greater than or equal to `0` and less than or equal to `Elements - 1`. Because of the comparison to zero this second form may be easier to use than the first.

The code to reference any array value with the index in `R1` is as follows:

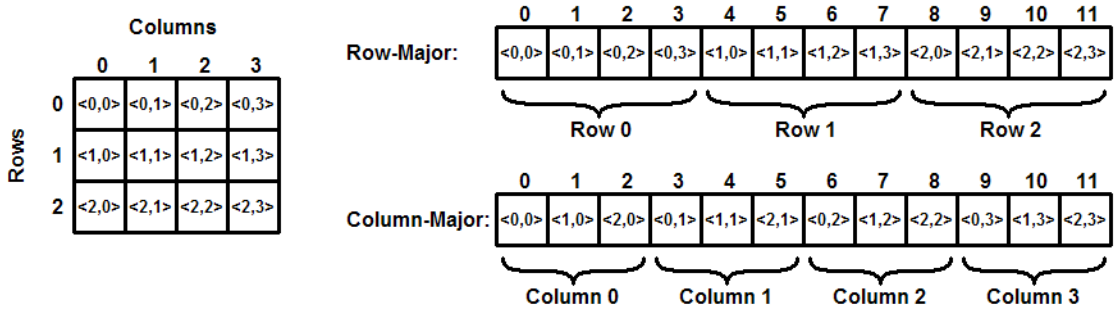
```
ADR  R5,Buffer           Base Address in R5
SUB  R2,R1,Low           Offset := R1 - Low
LDR  Rx,[R5,R2,LSL #2]   Rx := [Buffer+Offset*4]
```

If the code is to include range checking, it would be modified as follows:

```
ADR  R5,Buffer           Base Address in R5
SUBS R2,R1,Low           Offset := R1 - Low
BMI  BoundsError         If (Offset<0) OR
CMP  R2,Elements-1          (Offset>Elements-1)
BGT  BoundsError              Then BoundsError
LDR  Rx,[R5,R2,LSL #2]   Rx := [Buffer+Offset*4]
```

TWO-DIMENSIONAL

For two-dimensional arrays the same issues are present as before, but the mapping function is slightly more complicated, particularly if the array is not zero-based. The mapping function must also deal with how the rows and columns of the array are to be laid out in memory. For example, a two-dimensional array of three rows and four columns must map onto a linear one-dimensional array of twelve elements, but should the elements in each row or the elements in each column be kept together? Keeping the elements of each row together is called ***row-major form***, and keeping the elements of each column together is called ***column-major form***. Most modern high-level programming languages use row-major form, but a notable exception is FORTRAN which uses column major form. This can be a serious compatibility issue if data written out by an old FORTRAN program are to be read in to an array in a modern row-major language. These differences are illustrated below:



In the zero-based array shown above, the mapping function from row and column onto linear offset (using row-major form) is:

```
Offset := 4 * Row + Column
```

If the column value is in register `R2` and the row is in register `R3`, and the array contains four-byte integers, then the ARM code to load a random element of the array into register `Rx` is as follows:

```
ADR   R5,Buffer            R5 := Address of Buffer
MOV   R1,R3,LSL #2         R1 := R * 4
ADD   R1,R1,R2             R1 := R * 4 + C
LDR   Rx,[R5,R1,LSL #2]    Rx := Buffer[R1]
```

A code fragment in a high-level language to set all elements of the array to zero may be written as follows:

```
For R := 0 to 2 Do
     For C := 0 to 3 Do
          Buffer[R,C] := 0 ;
```

A "dumb compiler" might generate random access code for every access. A smart compiler would recognize that the inner loop accesses items in a row in linear order; a smarter compiler recognizes that all cells are being accessed in order, and can generate code that treats the array as a one-dimensional array *as long as the array is stored in row-major order*. The following code performs the same logical function as the earlier version (setting all elements of the array to zero), but because the array is stored in row-major form and the code accesses the array in column-major order, a compiler would have to be *really smart* to figure out how to optimize the corresponding assembly language:

```
For C := 0 to 3 Do
     For R := 0 to 2 Do
          Buffer[R,C] := 0 ;
```

Any programmer of a high-level language must take array order into account; depending on a compiler to be smart enough to compensate for poorly written code can result in less than optimal performance. Knowing something about the underlying assembly language model allows a programmer to write more efficient high-level code, to exploit the strengths of less-than-perfect compilers, and to avoid or reduce the effects of some of their shortcomings.

NON ZERO-BASED 2D

As with one-dimensional arrays, two-dimensional arrays in languages such as Pascal may start at indexes other than zero. For example, the following array declaration is perfectly legal, given the appropriate definitions of constants `LowR`, `HighR`, `LowC`, and `HighC` (where `LowR` ≤ `HighR` and `LowC` ≤ `HighC`):

```
Buffer : Array [LowR..HighR, LowC..HighC] Of Integer ;
```

The formula to compute the linear offset index (again, assuming row-major order) must first calculate the starting offset of each row by multiplying the size of each row by the zero-based row offset, and then add the zero-based column offset to the result, as follows:

```
Offset  := (HighC-LowC+1) * (R-LowR) + (C-LowC)
```

By precomputing the number of columns the expression is converted into:

```
Columns := HighC - LowC + 1
Offset  := Columns * (R-LowR) + (C-LowC)
```

By rearrangement of terms this becomes:

```
Columns := HighC - LowC + 1
Offset  := Columns * R + C + (-LowR * Columns - LowC)
```

Finally, by precomputing all possible constants the final result becomes:

```
Columns := HighC - LowC + 1
Adjust  := -LowR * Columns - LowC
Offset  := Columns * R + C + Adjust
```

The values of `Columns` and `Adjust` are computed once by the compiler, and the value of `Offset` is computed during run time at each individual reference to the array.

In ARM code, the declaration of the array is as follows:

```
LowR      EQU       ____          pick any integer
HighR     EQU       ____          pick any integer
LowC      EQU       ____          pick any integer
HighC     EQU       ____          pick any integer
Rows      EQU       HighR – LowR + 1
Columns   EQU       HighC – LowC + 1
Elements  EQU       Rows * Columns
Adjust    EQU       -LowR * Columns – LowC
Buffer    %         Elements*4
```

As before, if the column value is in register `R2` and the row is in register `R3`, and the array contains four-byte integers, then the ARM code to load a random element of the array into register `Rx` is as follows (assuming that `Columns` and `Adjust` are small enough constants to be embedded into instructions):

```
ADR  R5,Buffer           R5 := Address of Buffer
MOV  R4,Columns
MUL  R1,R3,R4            R1 := Columns*R
ADD  R1,R1,R2            R1 := Columns*R + C
ADD  R1,R1,Adjust        R1 := Columns*R + C + Adjust
LDR  Rx,[R5,R1,LSL #2]   Rx := Buffer[R1]
```

In cases where the number of columns is a power of two the `MUL` can be replaced with an instruction using a shift, perhaps in combination with the subsequent `ADD` instruction. Doing so reduces the number of registers required. For any particular array definition the reference code must be examined to determine if the mapping function may be optimized.

THREE DIMENSIONAL

Three-dimensional arrays have rows, columns, and planes, and the mapping function is correspondingly more complicated. Higher dimensional arrays are no different in concept; only the mapping function changes.

A SAMPLE PROBLEM

Here is a sample problem illustrating the techniques we've covered. In Pascal, I might declare an array as follows:

```
Buffer : Array [10..15, -3..3] Of Integer ;
```

In this array there are six rows and seven columns, and the low indices of the array do not start at zero. Laid out in memory, the 42 elements of the array are in a long linear list. We need a mapping function that converts a row index between 10 and 15 and a column index between -3 and +3 into a linear offset between 0 and 41. Plugging the constants from the array declaration into the formula to compute offsets, we get the following terms:

```
Columns := (3 - (-3) + 1) = 7
Offset  := 7 * (R - 10) + (C - (-3))
         = 7*R - 70 + C + 3
         = 7*R + C - 67
```

There is no need to compute the `Adjust` term explicitly, since its value (67) can be embedded as a constant into an `ADD` or `SUB` instruction. Multiplication by 7 can be done with a reverse subtract `RSB` instruction and the barrel shifter; the shifter multiplies the row value by 8, then the reverse subtract removes one copy. This same technique can be used whenever multiplying by one less than a power of two.

The entire array offset expression can be computed in just three ARM instructions. If the row `R` is in `R3` and the column `C` is in `R2`, then the code to load `Buffer[R,C]` into a register in ARM assembly language is as follows:

```
ADR  R5,Buffer           R5 := Address of Buffer
RSB  R1,R3,R3,LSL #3     R1 := 7*R
ADD  R1,R1,R2            R1 := 7*R+C
SUB  R1,R1,#67           R1 := 7*R+C-67
LDR  Rx,[R5,R1,LSL #2]   Rx := Buffer[R1]
```

Conclusions

As we have seen, it is a pretty straightforward matter to compute the array offsets for single and double dimension arrays, whether zero-based or not. Knowing how easy this is only contributes to my bafflement at why many modern high-level languages do not support non-zero-based arrays. In a language that only supports zero-based arrays, programmers must simulate non-zero-based arrays by writing their own mapping functions, much as what we have explored here. Why should they have to? Not only is it easy to generate assembly code to compute the mapping function, it is often true that the mapping function can be optimized into a very short, efficient form.