Lecture #16 – March 10, 2004

Arrays and the Stack

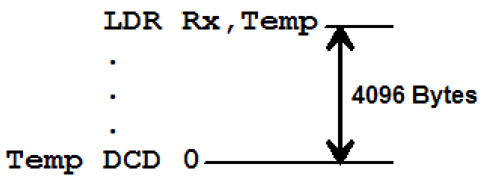
In this lecture we start thinking about using arrays of numbers in memory. By extension this allows us to start working with the stack, which is no more than a special type of memory array supported by a little extra hardware.

Simple Arrays

In a few processors such as the 6502, instructions that reference primary memory have unlimited access to every byte in that memory. The 6502 has a 64K address space, requiring 16 bits per ***absolute address***, and many instructions that reference memory can include all 16 bits. For example, the `LDA` (load accumulator) instruction requires three bytes of memory; one byte for the op code and two for the 16-bit address. This makes the creation and manipulation of arrays fairly straightforward, as any memory-reference instruction may also add the contents of an ***index register*** to the ***base address*** encoded in that instruction. (While addresses on the 6502 are 16 bits wide, index registers X and Y are only 8 bits wide, so arrays are limited to a maximum of 256 bytes.)

On the 8088, where an absolute address is generated by adding a 16-bit offset value to a 16-bit segment value (shifted left by 4 bits), segment registers effectively partition the 1 megabyte address space into movable 64K arrays, and addresses are 16-bit offsets into these 64K arrays. For example, instructions are always referenced relative to the `CS` code segment register, data values are referenced relative to the `DS` data segment or the `ES` extra segment registers, and items on the stack are referenced relative to the `SS` stack segment register. Without special programming, code blocks, data blocks, and the stack are all limited to 64K bytes maximum. Register `BX` is used most often as an index register into an array; since `BX` is 16 bits wide it can generate any address within the 64K segment.

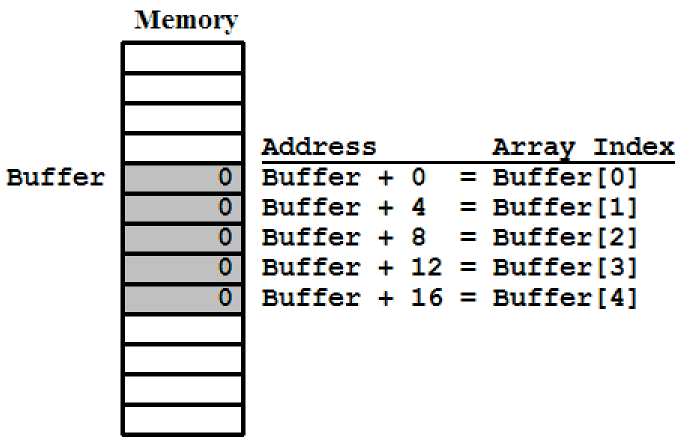
On the ARM, any of the sixteen 32-bit integer registers may be used as a base address register, *including the program counter*. We use this fact without really thinking about it when we set up "simple" `LDR` or `STR` instructions; in reality such memory references use the program counter `R15` as the 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. (The 8088 can do this, but only by tacking more bytes onto an instruction; most instructions range between 1 and 11 bytes in length. ARM instructions are always exactly 32 bits wide.) 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).



Now, let's attempt to 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 below:

```
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 follows:



Address	Array Index
Buffer + 0	= Buffer[0]
Buffer + 4	= Buffer[1]
Buffer + 8	= Buffer[2]
Buffer + 12	= Buffer[3]
Buffer + 16	= Buffer[4]

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 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
```

we are giving the assembler permission to do whatever is necessary to get the address of `Buffer` into register `Rx`. If the address is a value that can be generated by something as simple as a `MOV` instruction, it will generate a `MOV`, but in most cases it will reserve a "hidden" memory word to contain the address and then generate the appropriate `LDR` instruction. 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`, but we will discuss those later.)

```
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]
```

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 follows:

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

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 shows how we would store zero into every element of the array using this approach.

```
          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
```

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]    R0 := Buffer[R1]
          STR R0,[R5,R1,LSL #2]    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 decrement R1 by 1, not by 4. Here's the final example of storing zero into every element of the array:

```
          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
```

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. There are few restrictions on which registers may be used in this manner. Also, since registers are 32 bits in length, modifying either the base address register or the array index register (or both) will allow us to reference any word in memory, regardless of where it is located. The ±4096 byte rule applies only to *constant offsets*, which must be embedded into a 32-bit instruction along with the op code, condition, destination register, etc. We could step through the elements of an array by modifying the base address register just as easily as by using an index register. Of course, we must take great care not to run beyond the end of the array, in either direction!

The Stack

In their simplest form stacks are nothing more than arrays of words in memory. In all common processors that support stacks (including the 6502, 8088, and ARM) there is an extra register called the ***stack pointer***, which decreases in value as items are pushed and increases in value as items are popped. Stacks tend to grow downwards, and shrink upwards.

On many processors there are explicit push and pop instructions. For example, on the 6502 the instructions to push and pop (or "pull") the accumulator are `PHA` and `PLA`, respectively. On the 8088, you must explicitly specify the register you wish to push or pop, as in `PUSH AX` or `POP BX`.

On the ARM there are no explicit push or pop instructions, but we can do much the same work with the stack as with any other array, with a couple of additional bits of notation. Register `R13` is the stack pointer, and has the special name `SP`.

To push some register `Rx` onto the stack, we must store the value in memory at an address relative to `SP`, but since the stack grows downwards the offset will be negative, and since registers are 32 bits wide the offset will be 4 bytes. Thus, pushing `Rx` onto the stack starts out with the instruction:

```
STR Rx,[SP,#-4]      Stack[SP-4] := Rx
```

Unfortunately, this does not change the value of `SP`, so the next time we execute this instruction the previous value is overwritten by the new value. So, we must also decrease the value of the stack pointer. A true push would be written as follows:

```
STR Rx,[SP,#-4]      Stack[SP-4] := Rx
SUB SP,SP,#4         SP := SP-4
```

A pop would reverse these actions:

```
LDR Rx,[SP]          Rx := Stack[SP]
ADD SP,SP,#4         SP := SP+4
```

Notice that in these fragments the stack point `SP` always points at the true top of the stack. The 8088 does this as well. The 6502, in contrast, points one byte *beyond* the end of the stack, at the next free byte instead of at the last filled byte. This becomes an issue of great concern when programming a wide variety of different processors.

In the ARM, however, we have additional notation that allows us to perform pushes and pops in one instruction each, instead of two. To combine the store into memory and the update of the stack pointer, we would write:

```
STR Rx,[SP,#-4]!     Push(Rx)
```

The addition of the exclamation point (or "bang") sets one bit in the `STR` instruction that tells the processor to update `SP` with the new value of `SP-4` after the store has occurred. This is called ***write-back***.

Similarly, to combine a load from memory with the update of the stack pointer, we would write:

```
LDR Rx,[SP],#4        Pop(Rx)
```

Notice that there is no exclamation point on this instruction. The presence of the `#4` *outside* of the square brackets says that the addition happens *after* the load has already occurred, so write-back happens by default.

There is no earthly reason for an ARM assembler to *not* contain `PUSH` and `POP` synonyms for pushing and popping registers, which would then be translated into the proper `LDR` and `STR` instructions by the assembler. For example, we *ought* to be able to write something like `PUSH R0` and have the assembler automatically build the instruction `STR R0,[SP,#-4]!` internally, but I do not think that the ARMulator does this. You must get into the habit of seeing past the complex memory reference instructions to their underlying meanings. From now on:

```
STR Rx,[SP,#-4]!   means  Push(Rx)

LDR Rx,[SP],#4     means  Pop(Rx)
```