Array Basics

Array basics

- Arrays are dedicated blocks of memory for storing lists of values of the same type (ie size)
- Almost all programming languages support arrays:
  - E.G in C programming:

    ```
    int somearray[10];
    ```

    Defines a block of memory referred to as “somearray”, which holds 10 integer (32 bit) values

Array basics

```
int somearray[10];
```

Using memory in ARMlite, the above could translate to a block of memory like this.

Memory

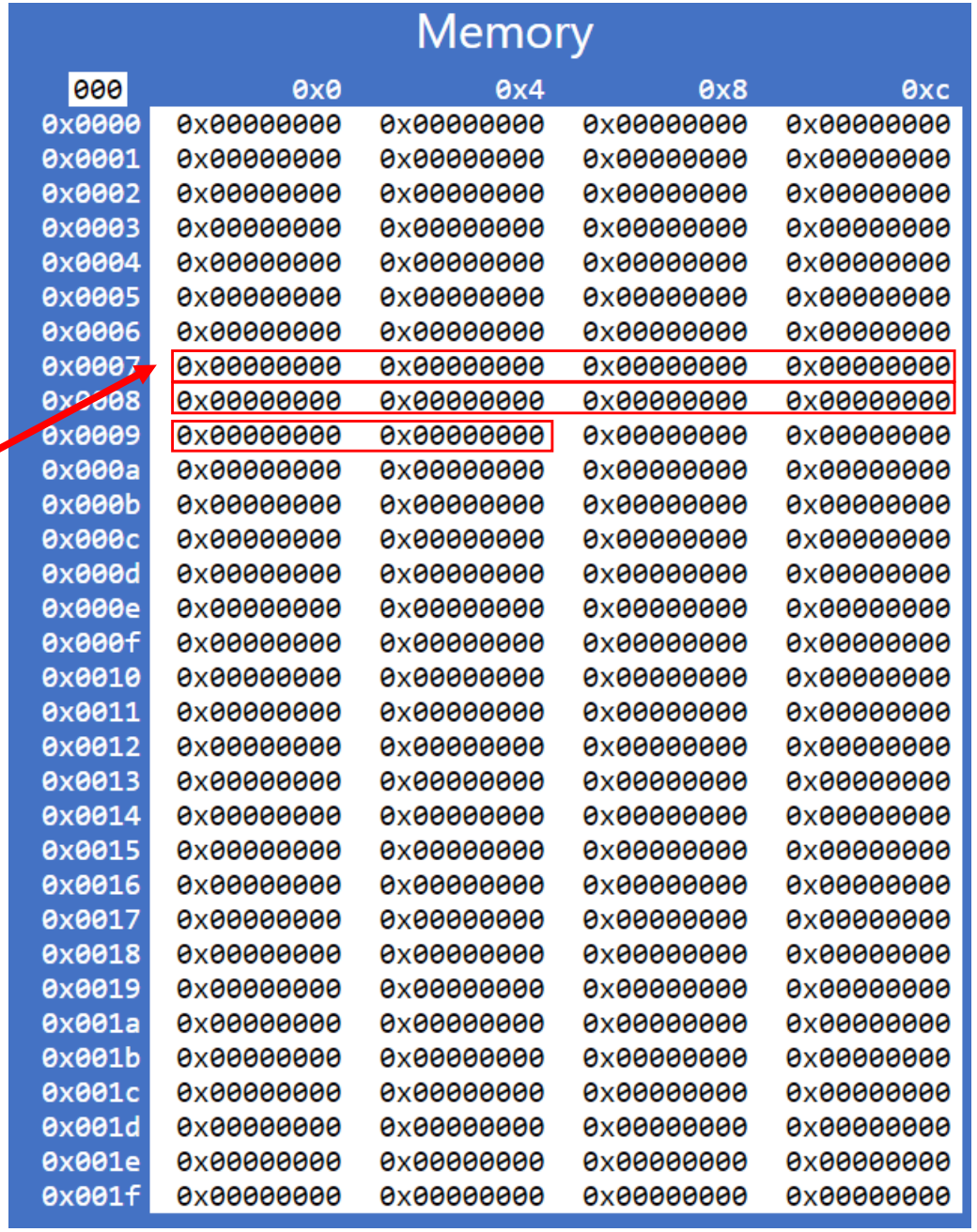
000	0x0	0x4	0x8	0xc
0x0000	0x00000000	0x00000000	0x00000000	0x00000000
0x0001	0x00000000	0x00000000	0x00000000	0x00000000
0x0002	0x00000000	0x00000000	0x00000000	0x00000000
0x0003	0x00000000	0x00000000	0x00000000	0x00000000
0x0004	0x00000000	0x00000000	0x00000000	0x00000000
0x0005	0x00000000	0x00000000	0x00000000	0x00000000
0x0006	0x00000000	0x00000000	0x00000000	0x00000000
0x0007	0x00000000	0x00000000	0x00000000	0x00000000
0x0008	0x00000000	0x00000000	0x00000000	0x00000000
0x0009	0x00000000	0x00000000	0x00000000	0x00000000
0x000a	0x00000000	0x00000000	0x00000000	0x00000000
0x000b	0x00000000	0x00000000	0x00000000	0x00000000
0x000c	0x00000000	0x00000000	0x00000000	0x00000000
0x000d	0x00000000	0x00000000	0x00000000	0x00000000
0x000e	0x00000000	0x00000000	0x00000000	0x00000000
0x000f	0x00000000	0x00000000	0x00000000	0x00000000
0x0010	0x00000000	0x00000000	0x00000000	0x00000000
0x0011	0x00000000	0x00000000	0x00000000	0x00000000
0x0012	0x00000000	0x00000000	0x00000000	0x00000000
0x0013	0x00000000	0x00000000	0x00000000	0x00000000
0x0014	0x00000000	0x00000000	0x00000000	0x00000000
0x0015	0x00000000	0x00000000	0x00000000	0x00000000
0x0016	0x00000000	0x00000000	0x00000000	0x00000000
0x0017	0x00000000	0x00000000	0x00000000	0x00000000
0x0018	0x00000000	0x00000000	0x00000000	0x00000000
0x0019	0x00000000	0x00000000	0x00000000	0x00000000
0x001a	0x00000000	0x00000000	0x00000000	0x00000000
0x001b	0x00000000	0x00000000	0x00000000	0x00000000
0x001c	0x00000000	0x00000000	0x00000000	0x00000000
0x001d	0x00000000	0x00000000	0x00000000	0x00000000
0x001e	0x00000000	0x00000000	0x00000000	0x00000000
0x001f	0x00000000	0x00000000	0x00000000	0x00000000

Array basics

```
int somearray[10];
```

Using memory in ARMlite, the above could translate to a block of memory like this.

“somearray” refers to an address, marking the start of the allocated block

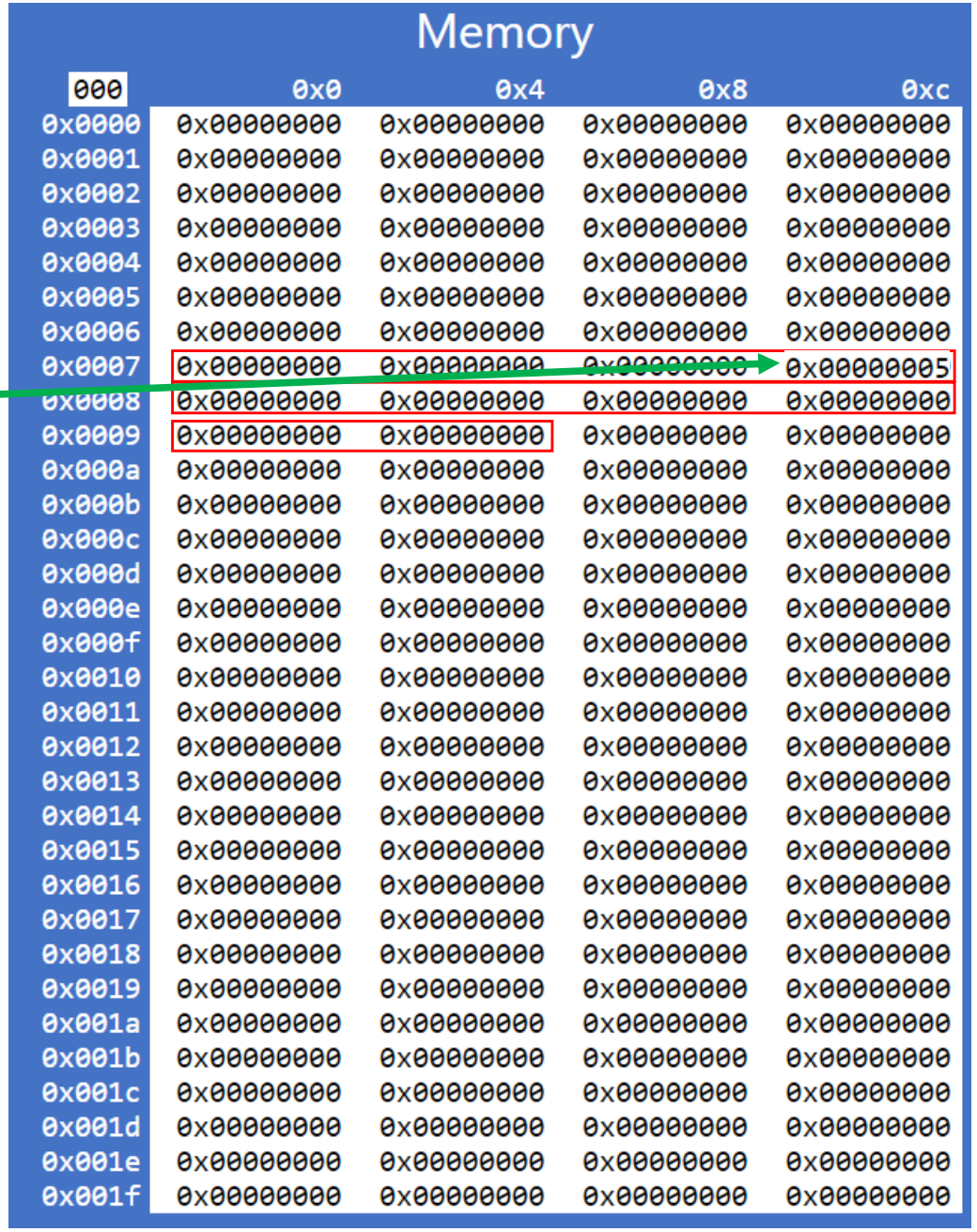


Array basics

```
int somearray[10];
somearray[3] = 5;
```

Once declared, somarray can be indexed to specify a particular 32 bit/4 byte) integer within the array.

In above case, it is assigning the value 5 to the cell indexed by 3



Array basics

```
int somearray[10];
somearray[3] = 5;
```

How does this translate to ARM assembly ?

Array basics

```
int somearray[10];
somearray[3] = 5;
```

How does this translate to ARM assembly ?

```
somearray: .word
0
0
0
0
0
0
0
0
0
0
```

We simply define a label!

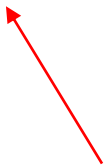
Array basics

```
int somearray[10];
somearray[3] = 5;
```

How does this translate to ARM assembly ?

We simply define a label!

```
somearray: .Word 0 0 0 0 0 0 0 0 0 0
```



The label “somearray” now refers to the base address of a block of 10 x 32 bit words

error correct: these values should be actually be one per line in actual code

Array basics

```
int somearray[10];
somearray[3] = 5;
```

How does this translate to ARM assembly ?

```
somearray: .Word 0 0 0 0 0 0 0 0 0 0
```

error correct: these values should be actually be one per line in actual code

Array basics

```
int somearray[10];
somearray[3] = 5;
```

How does this translate to ARM assembly ?

This involved a few steps so let's unpack it

```
somearray: .Word 0 0 0 0 0 0 0 0 0 0
```

error correct: these values should be one per line in actual code

Array basics

```
int somearray[10];
somearray[3] = 5;
```

We have the address of the array which needs to be stored in a register

```
MOV R0, #somearray
somearray: .Word 0 0 0 0 0 0 0 0 0 0
```

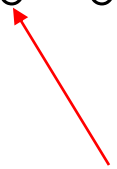
error correct: these values should be actually be one per line in actual code

Array basics

```
int somearray[10];
somearray[3] = 5;
```

We have an index to a 32 bit integer within the array.
Address of this value will be 3 x 4 = 12 bytes from start of array
We can store this in a register

```
MOV R0, #somearray
MOV R1, #12
somearray: .Word 0 0 0 0 0 0 0 0 0 0
```



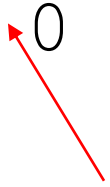
error correct: these values should be actually be one per line in actual code

Array basics

```
int somearray[10];
somearray[3] = 5;
```

We can keep the value to store in another register

```
MOV R0, #somearray
MOV R1, #12
MOV R2, #5
somearray: .Word 0 0 0 0 0 0 0 0 0 0
```



error correct: these values should be actually be one per line in actual code

Array basics

```
int somearray[10];
somearray[3] = 5;
```

Finally, we can write the value to the array using STR

```
MOV R0, #somearray
MOV R1, #12
MOV R2, #5
STR R2, [R0 + R1]
somearray: .Word 0 0 0 0 0 0 0 0 0 0
```

error correct: these values should be actually be one per line in actual code

Array basics

```
int somearray[10];
somearray[3] = 5;
```

```
MOV R0, #somearray
MOV R1, #12
MOV R2, #5
STR R2, [R0 + R1]
HALT
somearray: .Word 0 0 0 0 0 0 0 0 0 0
```

We make sure program is halted so it does not try and execute anything in the array

error correct: these values should be actually be one per line in actual code

Array Basics

- Lets have a look at the code in ARMLite

Some array definition for single byte arrays

- For a generic array of single bytes:

```
somebytearray:   .Byte    1 3 221 56
```

- For an array of ASCII characters (ie., a string):

```
somestring: .ASCIZ  “hello world\n”
```

Some array definition for single byte arrays

- For a generic array of single bytes:

```
somebytearray:   .Byte    1 3 221 56
```

- For an array of ASCII characters (ie., a string):

```
somestring: .ASCIZ  “hello world\n”
```

This directive ensures a character per byte, and the appending of a NULL character at the end of the string.

An alternative is .ASCII, which does not append the NULL character

Memory Alignment

- Memory Alignment is a common requirement:
  - hardware often imposed alignment requirements to ensure integrity
- When defining byte addressable arrays, precede the array definitions with .Align.
- For example:

```
.ALIGN 128
somestring: .ASCIZ  “hello world\n”
```

`.ALIGN N` ensures the next instruction is aligned with a word address divisible by `N` (in general, and multiple of 4 is generally fine).