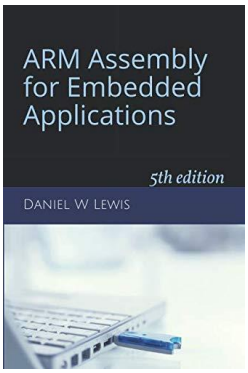


Programming Lab 4B

Address Alignment

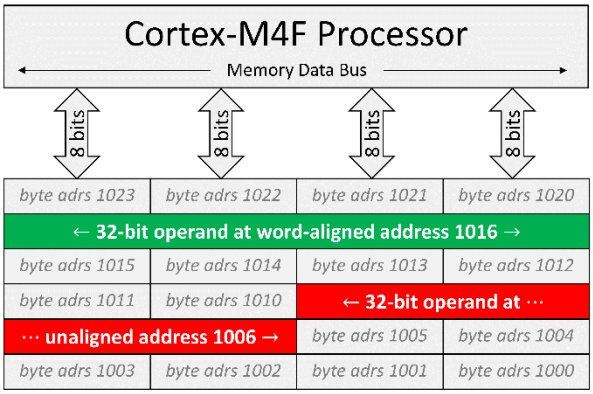
Topics: Address alignment, execution time, and the .REPT and .ENDR directives.

Click to download Lab4B-Main.c

Prerequisite Reading: Chapters 1-4

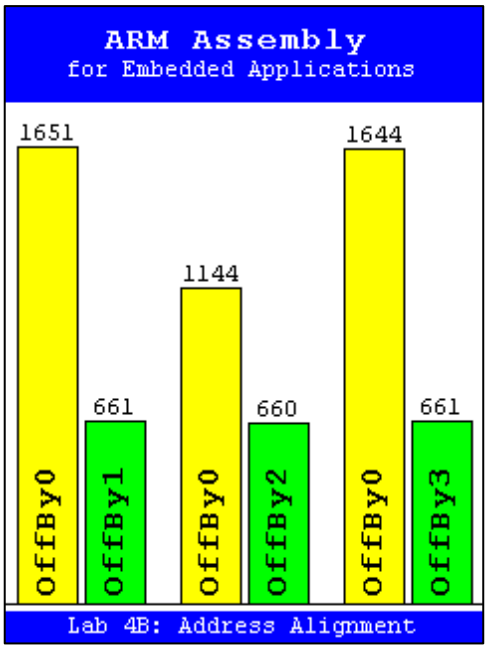
Revised: January 24, 2021

Background: *Logically*, we think of memory as a collection of bytes, each having its own address, with larger operands stored as a contiguous group of bytes. However, the memory of a 32-bit processor is *physically* implemented as words of four bytes each to optimize performance, which allows a native 32-bit operand to be read or written in a single memory cycle. Retrieving an 8-bit operand simply requires the processor to read a full 32-bit physical word from memory and then select the appropriate byte to use. When writing to memory, the processor selectively enables or disables each of the four data bus bytes depending on the size of the operand.



Address alignment can have an adverse effect on execution time. To read or write a 32-bit operand in a single memory cycle requires that its location start at an address that is a multiple of four, such as the word shown in green in the figure. (A similar restriction applies to 16-bit operands.) Otherwise, the operand will be split across two physical words (e.g., the example shown in red), which forces the processor to perform additional memory cycles to transfer all 32-bits. Thus, to optimize performance, compilers always place 16, 32 and 64-bit operands at locations whose addresses are multiples of two, four and four respectively.

Assignment: The main program may be compiled and executed without writing any assembly. However, your task is to create faster assembly language replacements for the four C functions shown below using their C versions to guide your implementation. Each function copies 1000 bytes of data between source and destination regions of memory. The original C functions are defined as "weak", so that the linker will automatically replace them in the executable image by those you create in assembly; you do not need to remove the C version.



The objective is that functions `OffBy1`, `OffBy2` and `OffBy3` are to be optimized for source and destination regions that are not word-aligned as implied by their names. The main program compares their execution time to that of calling `OffBy0` with the same unaligned source and destination addresses. The latter should always be slower.

You may assume that the source and destination do not overlap. To minimize execution time, your assembly versions should copy as much data as possible using a sequence of `LDR/STR` instructions with word-aligned addresses and any other bytes as necessary using `LDRB/STRB` instructions. To avoid loops, use the `.REPT` and `.ENDR` directives similar to what is shown in Listing 4-1 of the textbook to create repeated sequences of `LDR/STR` pairs (and `LDRB/STRB` pairs as needed) with post-indexed addressing as described in Table 4-6.

```
void OffBy0(void *dst, const void *src) ;   // These functions each copy 1000 bytes of data. Each should
void OffBy1(void *dst, const void *src) ;   // be optimized for source and destination regions that begin
void OffBy2(void *dst, const void *src) ;   // at addresses that are 0, 1, 2 or 3 bytes beyond a word-aligned
void OffBy3(void *dst, const void *src) ;   // location as indicated by the digit in their function name.
```

If your code works correctly, the display should look similar to the image shown at right with each function's execution time shown in clock cycles at the top of each bar graph. (Your numbers may differ, and the bar graph of an incorrect copy will be displayed in solid red.)