ASSIGNMENT NO. 2
MUHAMMAD HAMAD – MS-IC-21354

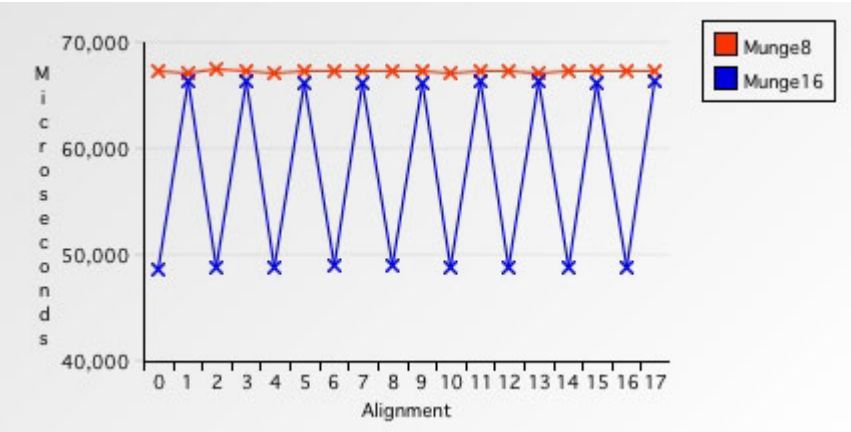
Question 1. Define Aligned and Mis-aligned access using diagrams? How does access granularity affect alignment?

Ans : Before discussing Aligned and Misaligned access we must know about **memory access granularity.** Our computer's processor does not read from and write to memory in byte-sized chunks. Instead, it accesses memory in two-, four-, eight-16- or even 32-byte chunks. We'll call the size in which a processor accesses memory its **memory access granularity.**

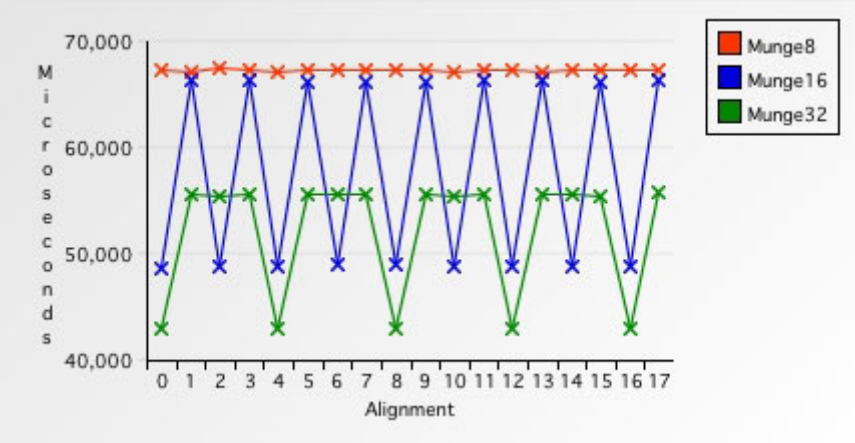
<u>Aligned access</u> : In the above figure observe what happens when reading from address 0. Because the **address fall evenly on the processor's memory access boundary,** the processor has no extra work to do in this case. Such an address is known as an **aligned** address.

<u>Mis-aligned access</u> :
In the above figure observe what happens when reading from address 1. Because the **address doesn't fall evenly on the processor's memory access boundary**, the processor has extra work to do. Such an address is known as an **unaligned** address. Because **address 1 is unaligned,** a processor with **two-byte granularity** must perform an extra memory access, slowing down the operation.

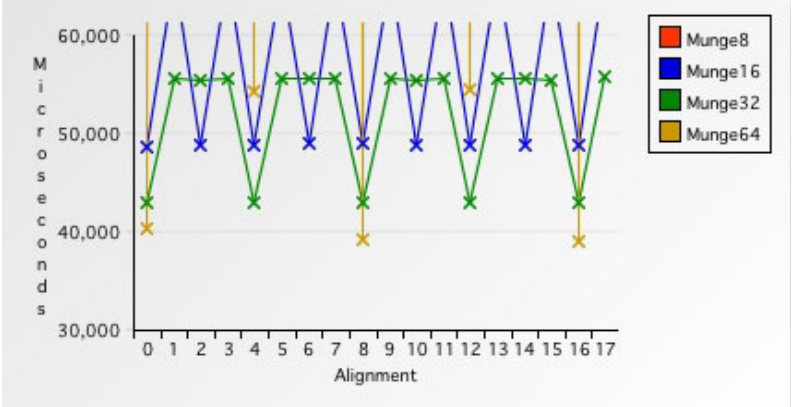
Now we will try to answer the second portion of the question i.e. **How does access granularity affect alignment ?**



Observe the image above that is showing the relationship between Single-byte access versus double-byte access , The first thing you notice is that **accessing memory one byte at a time is uniformly slow**. The second item of interest is that when accessing memory two bytes at a time, whenever the **address is not evenly divisible by two**, that **27% speed penalty** rears its ugly head.
This function took **48,765 microseconds** to process the same ten-megabyte buffer **38% faster** than Munge8. However, that buffer was aligned. If the buffer is unaligned, the time required increases to **66,385 microseconds** about a **27% speed penalty.**



Observe the image above that is showing the relationship between Single-byte access versus double-byte access **Single- versus double versus quad-byte access.** This function processes an aligned buffer in **43,043 microseconds** and an unaligned buffer in **55,775 microseconds**, respectively. Thus, on this test machine, accessing unaligned memory four bytes at a time is **slower** than accessing aligned memory two bytes at a time.



Munging data **eight bytes at a time** shows that it processes an aligned buffer in **39,085 microseconds** about **10% faster** than processing the buffer four bytes at a time. However, processing an unaligned buffer takes an amazing **1,841,155 microseconds –** two orders of magnitude slower than aligned access, an outstanding 4**,610% performance penalty.** Also Notice accessing memory eight bytes at a time on four- and twelve- byte boundaries is slower than reading the same memory four or even two bytes at a time.

Questions 2. What is the purpose of structure padding? Is it possible to use padded memory?

Ans : Padding is the act of adding otherwise unused space to a structure to make fields line up in a desired way. There's **two reasons** for this: **backwards compatibility** and **efficiency**.
First, backwards compatibility. Remember the 68000 was a processor with two-byte memory access granularity, and would throw an exception upon encountering an odd address and if a debugger weren't installed, the old Mac OS would throw up a System Error dialog box with one button: Restart. Yikes!

So, instead of laying out your fields just the way you wrote them, the compiler padded the structure so that b and c would reside at even addresses:

```
void Munge64( void *data, uint32_t size ) {
typedef struct {
        char a;
        long b;
        char c;
}       struct;
```

Field Type	Field Name	Field Offset	Field Size	Field End
`char`	a	0	1	1
	padding	1	1	2
`long`	b	2	4	6
`char`	c	6	1	7
	padding	7	1	8
Total Size in Bytes:				8

Structure with compiler padding

The second reason is **efficiency**. Nowadays, on Power PC machines, two-byte alignment is nice,but four-byte or eight-byte is better. You probably don't care anymore that the original 68000 choked on unaligned structures, but you probably care about potential 4,610% performance

penalties, which can happen if a double field doesn't sit aligned in a structure of your devising.

Now coming towards **second portion of question i.e. Is it possible to use padded memory?**

Uptill now, we've seen that modern compilers try to optimize data structures for maximum performance using padding unless specified otherwise. This comes at the trade-off of bigger structures in memory but given the abundance of memory nowadays it seems negligible in comparison to the potential speedups this optimization may have. We've also seen that even in case code is deliberately misaligned,modern processors will still not take a hit, though older processors seem to be greatly affected by memory misalignment.

The only thing the programmer still needs to take manual care of is creating efficient data structures by ordering members so that memory waste by padding is minimized.
