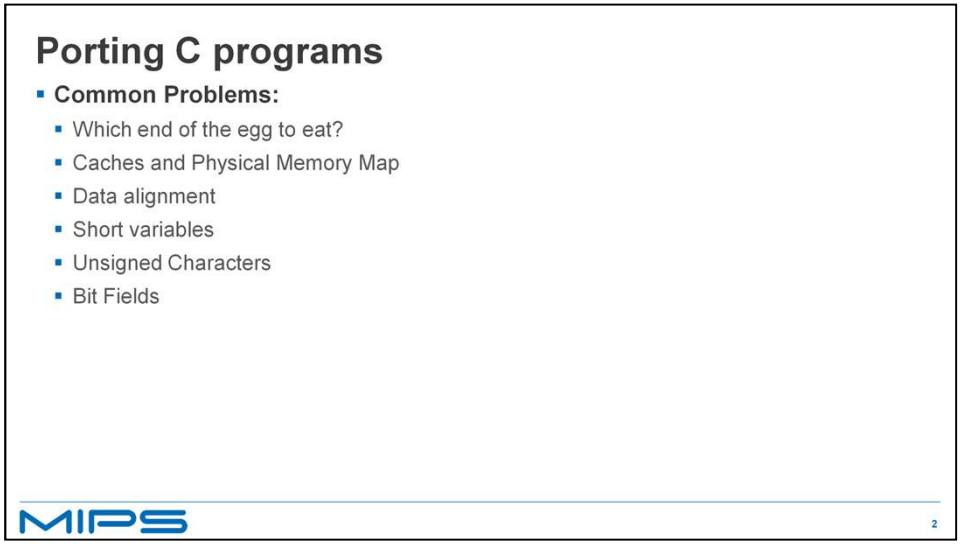
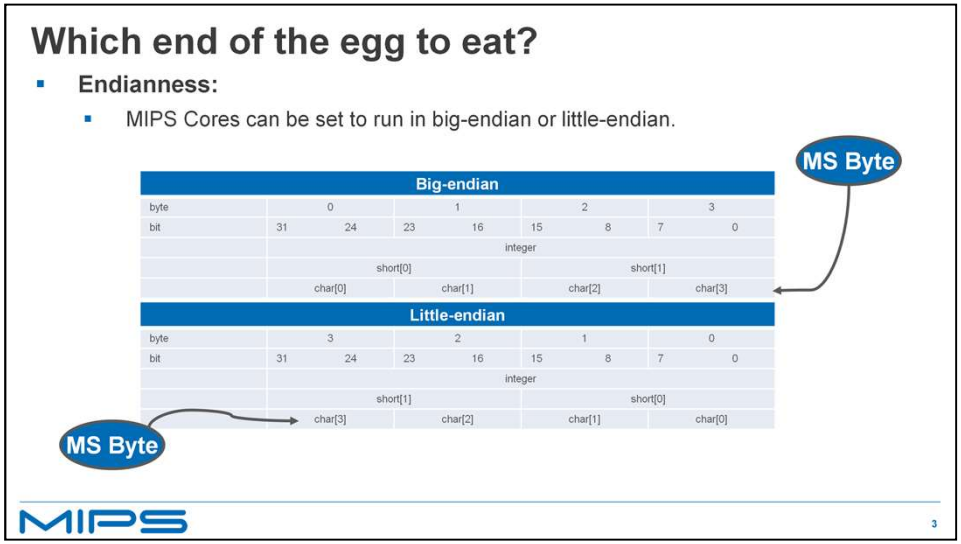
This section covers some of the differences that you can encounter when you are porting software to a MIPS processor.



Here is what I'm going to cover:

+ Endianness, what is it?
+ I'm going to review of Caching issues
+ talk about Odd data alignment
+ discuss the Use of short variables
+ Warning about the use of the type character in your code
+ And the Use of bit fields



In Jonathan Swift's Gulliver's Travels, the "little-endians" and "bigendians" fought a war over the correct end at which to start eating a boiled egg.

Danny Cohen used this as an analogy for a paper he wrote in 1980 entitled "On Holy Wars and a Plea for Peace". The paper covers the topic of byte ordering in computer systems. While it didn't stop a war it did coin the terms Little-endian and Big-Endian.

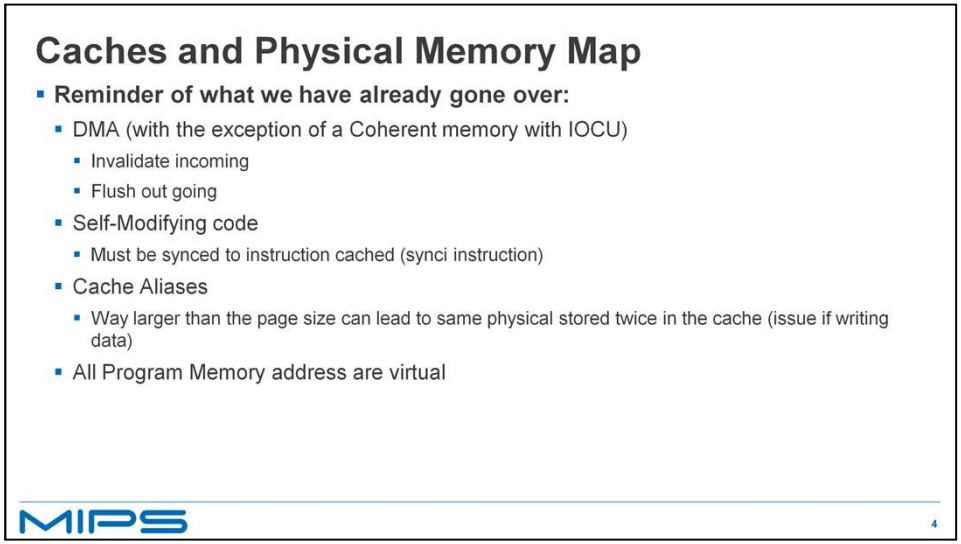
These two tables show the differences in the two methods of ordering. Integers or words don't come out any differently but shorts and characters are arranged within the words in a different order.

A MIPS CPU can be set to work in either endian and there are instructions that can be used to swap bytes with in words efficiently. Those instructions Load word right and load word left are covered in the Instruction Set section of this course.

So long as binary data is never imported into your application from elsewhere, and you avoid accessing

the same piece of data under two different endians, your code is portable.

Beyond that, the subject is covered in detail in the Danny Cohen paper you can find this on the I triple E web site. It is also covered in Chapter 10 of the “See MIPS Run Linux” book.



On caches, virtual, and physical memory here are reminders of some of the things covered in various sections of this course:

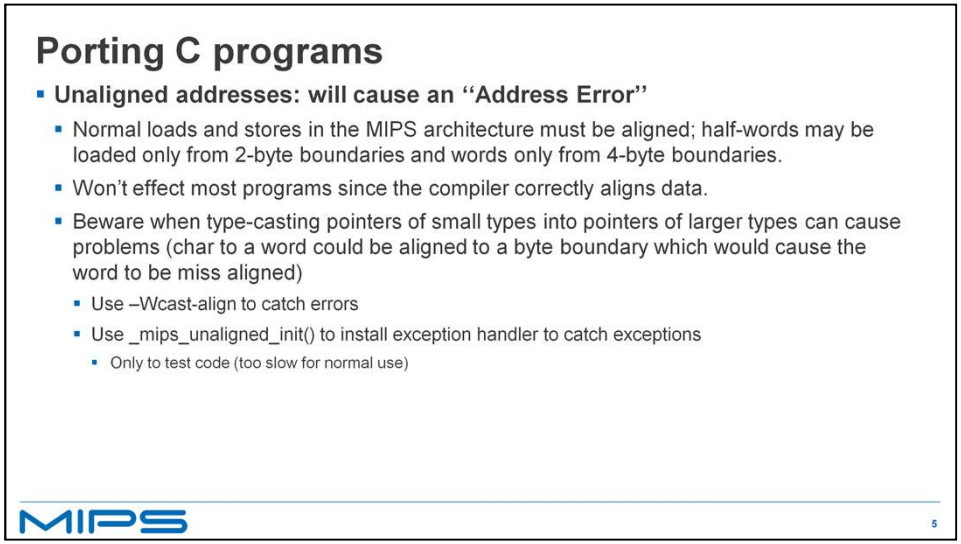
+ In regards to DMA the caches are not coherent so you must invalidate cache lines for an incoming DMA and flush cache lines to memory for outgoing DMA. Coherent Processing Cores with an IOCU do not need to do this. This is covered in the Coherent Processing section for Core that support it.

+ If you are loading code into memory or modifying code on the fly to cached address you must use the synci instruction to sync the instruction cache with the data cache and memory.

+ Remember, the page size needs to be as large or larger than a way of your cache to avoid Cache Aliases.

+ All memory accesses are virtual address. Some are direct mapped and some are mapped through the

use of the TLB.



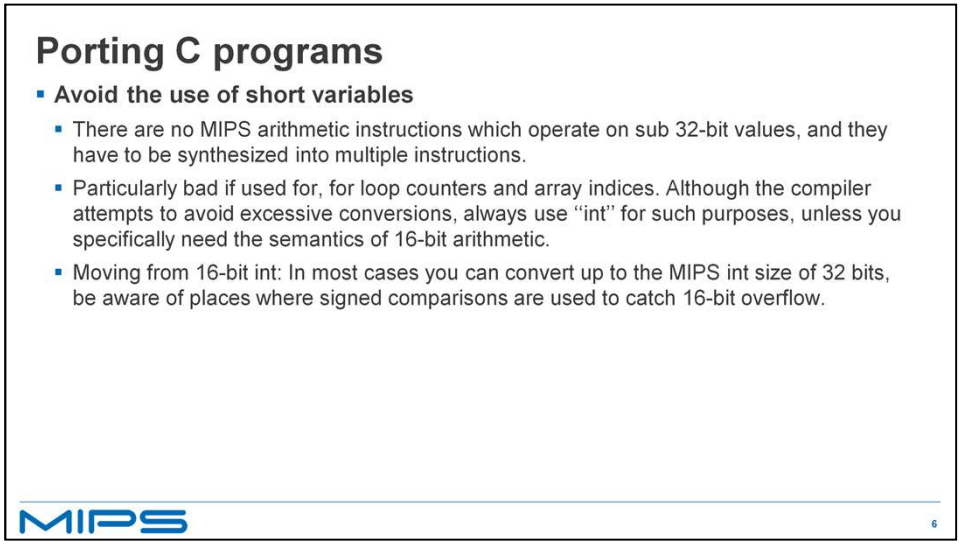
Accesses to unaligned addresses cause an address error exception.

+ the address accessed must be aligned to the instruction type. The Load or Store word instructions need to address on a four byte boundary. The Load or Store half word instructions need to address a double byte boundary. Only Load or Store byte instructions can access any byte address.

+ The compiler takes care of this for you unless you are type casting.

+ type casting a small type to a larger type can lead to a unaligned access. For example casting a byte that is aligned in the middle of a word to a word will cause an address error. To get the compiler to warn you about possible unaligned type casts, use the –Wcast align option.

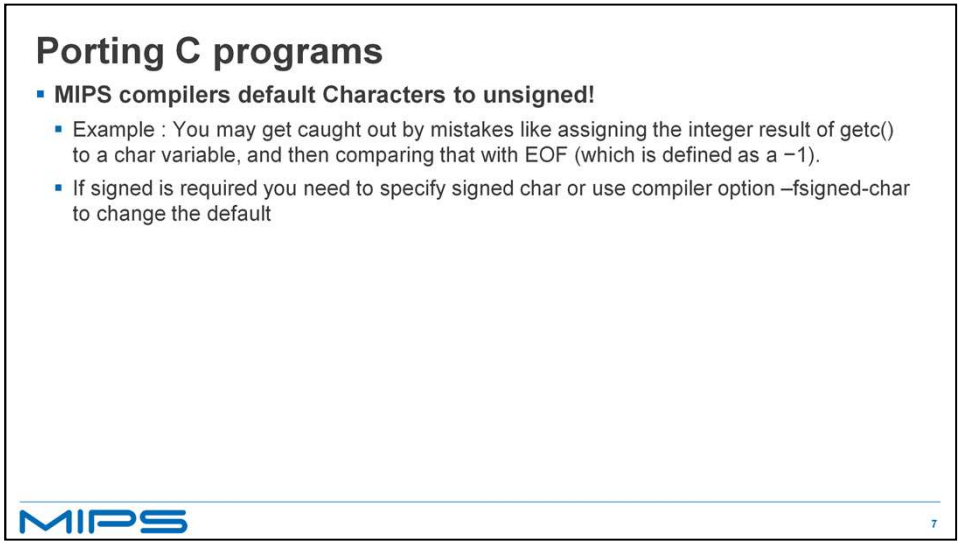
There is a supplied exception handler that you can use to catch the address error exception that can fix up errors but to be more efficient you should really try to fix the alignment in your code.



All arithmetic in a MIPS CPU is done on 32 bit values. Any instructions that do arithmetic on values less that 32 bits are synthesized instructions that will translate into multiple machine instructions.

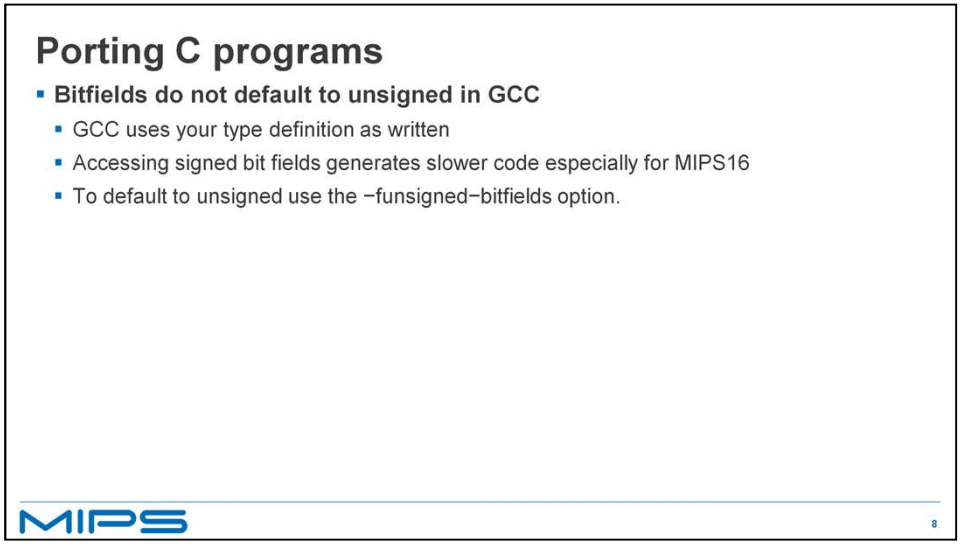
+ You may think you are saving space by using a short for a loop counter or an index into a small array but in fact you are adding at least one 32 bit instruction to do a conversion from a short to a integer and taking a performance hit.

+ When you are porting code from a 16 bit cpu to MIPS you need to use integers that are 32 bit. If your code relied on the sign bit being bit 15 to catch over flow you will have to change your code to check bit 31 instead.



MIPS compilers default the char type to unsigned. This has caught some programmers out because they used a char variable to hold the result of functions that only return zero or negative one and the default char type will never look like a negative one.

+ The compiler default can be changed by using the –fsigned-char option.



Bitfields often very from one compiler to another. When you define bits in a bit field you should always implicitly define each field as signed or unsigned. The GCC compiler will default any non implicitly defined fields as signed. For MIPS it harder to process signed fields then unsigned. You can switch the default to unsigned by using the –funsigned-bitfields option.