Word alignment

32-bit word: 4 bytes

Suppose we want to store the word 0123ABCD$_{hex}$

Start at address 1000

big-endian	**data**	**01**	**23**	**AB**	**CD**
	address	**1000**	**1001**	**1002**	**1003**
little-endian	**data**	**CD**	**AB**	**23**	**01**
	address	**1000**	**1001**	**1002**	**1003**

We say the word is stored at address 1000, meaning it's stored beginning at address 1000

Could we store these same 4 bytes starting at address 1001, for example?

Yes, but the hardware for accessing the data in memory is simpler if the data is aligned

> **A word begins on a word boundary (address divisible by 4)**
>
> **What's a good way to tell if an address is a word boundary?**
>
> **If its address in binary ends in 00**
>
> **A halfword is aligned on an address divisible by 2**

What is the effect on high-level language?

Consider the structure

```
struct Foo {
   char x ;   // 1 byte
   int y ;    // 4 bytes
   char z ;   // 1 byte
   int w ;    // 4 bytes
} ;
```

What is the size of a variable of type struct Foo?

1 + 4 + 1 + 4 = 10 bytes

Not necessarily! If the ints are aligned on word boundaries, there must be 3 bytes between the chars and the ints.

This means that the size of the struct is 16 bytes, if alignment is required.

The extra bytes are called padding or holes.

This is the main reason struct variables can't be directly compared in C, but they can be assigned directly.

The efficient way to compare would be to compare all bits in each struct, but the pad bytes, if any, are undefined, and may be any value.

Why is assignment OK?

Try sizeof operator on this struct

Summary

	address divisible by	binary address ends in
byte	1	anything
halfword	2	0
word	4	00
doubleword	8	000