CS 40: Machine Structure
and
Assembly Language Programming (Spring 2024)

Big Endian vs. Little Endian Storage of Numeric Data

Goals for this presentation

- **Explore two different conventions for storing numbers *in computer memory***
- **Learn the specifics of “Big-endian” and “Little-endian” representations**
- **Focus on “little-endian” – used by our AMD 64 computers**
- **Note: none of this affects the storage of characters or character strings! Here, we are discussing only multibyte numeric types.**

The Problem

What’s the issue?

- **We usually think of an integer variable as a single value:**

```
int myint = 0x1A2B3C4E;
```

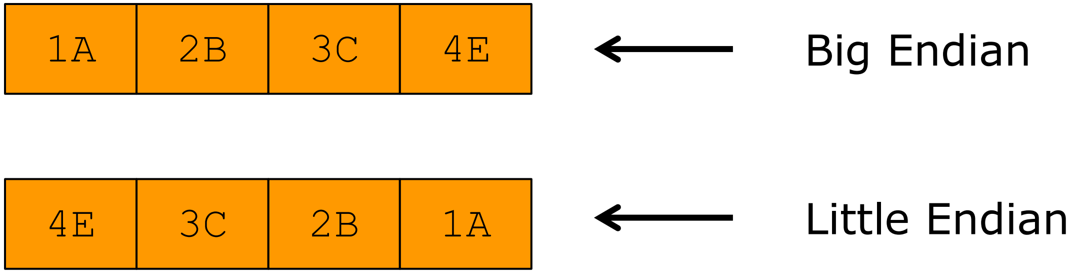
- **If we store it in memory, that takes 4 bytes, *each of which is addressable*…which is stored first?**

What’s the issue?

- **We usually think of an integer variable as a single value:**

```
int myint = 0x1A2B3C4E;
```

- **If we store it in memory, that takes 4 bytes, *each of which is addressable*…which is stored first?**



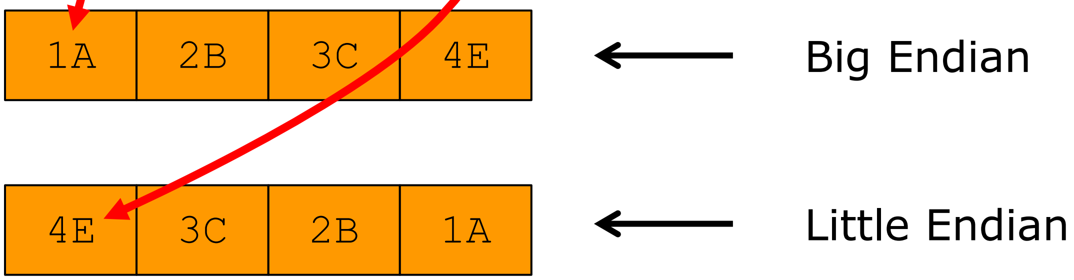
1A	2B	3C	4E	Big Endian
4E	3C	2B	1A	Little Endian

What’s the issue?

- **We usually think of an integer variable as a single value:**

```
int myint = 0x1A2B3C4E;
```

- **If we store it in memory, that takes 4 bytes, each of which is addressible…which byte of the int is stored first?**



1A	2B	3C	4E	Big Endian
4E	3C	2B	1A	Little Endian

What’s the issue?

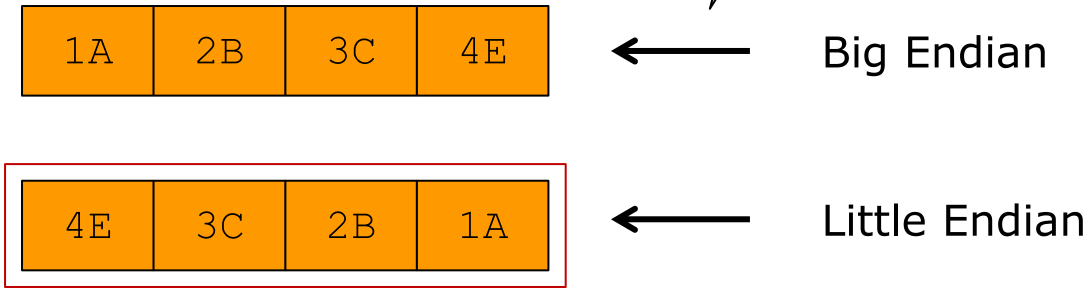
The choice depends on the model of computer you are using.

- We usually think of an integer variable as

Our AMD64 machines are *little endian!*

```
myint = 0x1A2B3C4E;
```

- If we st in memory, that takes 4 bytes ach of which is addressi .which byte of the int is store first?



1A 2B 3C 4E ← Big Endian

4E 3C 2B 1A ← Little Endian

Can your program tell the difference?

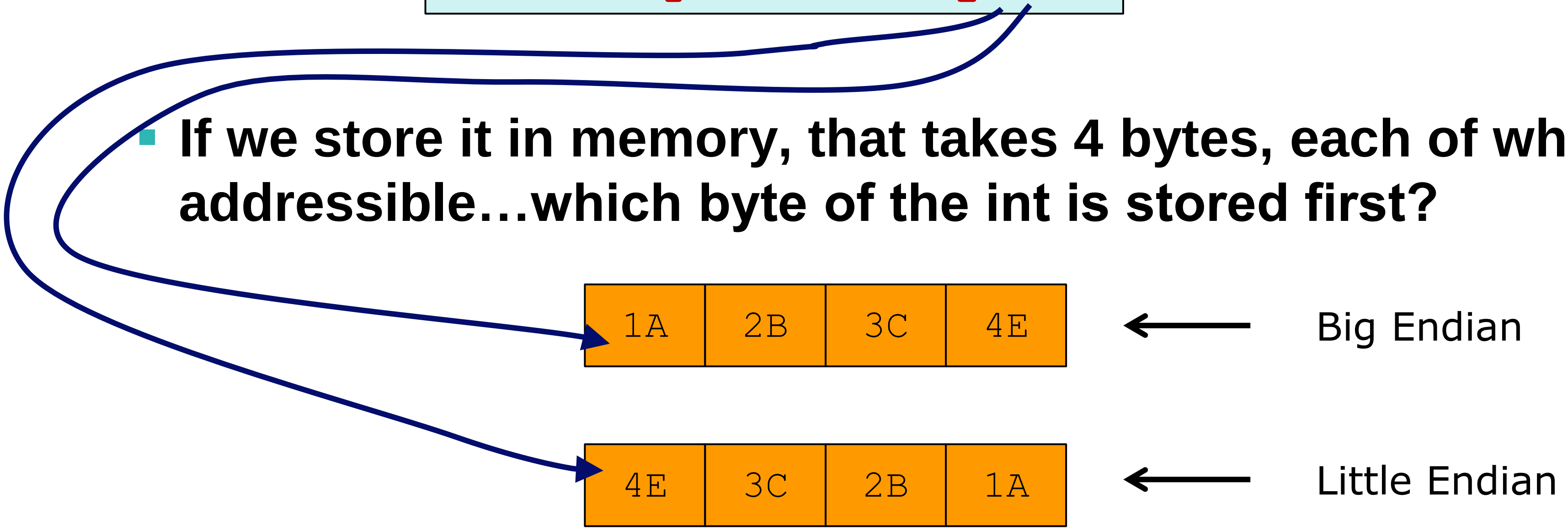
Pointing to integers in memory

Pointer is always address of *first* byte.

- We usually think of an integer variable as a value.

```
  int   myint = 0x1A2B3C4E;
char *ip      = &myint;
```

- If we store it in memory, that takes 4 bytes, each of which is addressible…which byte of the int is stored first?



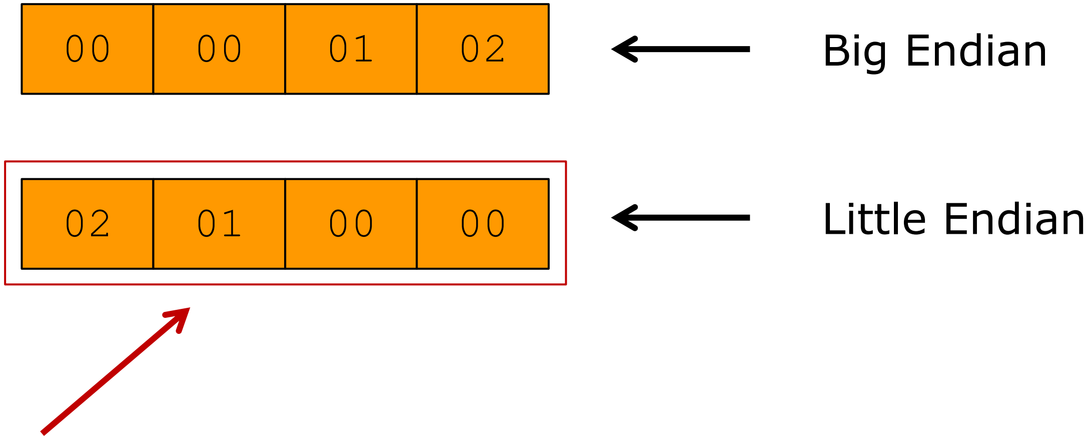
1A	2B	3C	4E	←	Big Endian
4E	3C	2B	1A	←	Little Endian

If you print `*ip` in hex on our AMD 64 machines you will get `4E`
…on other computers you may get `1A` from the same program!

Example: positive number

- **We usually think of an integer variable as a single value:**

```
int myint = 258;
```



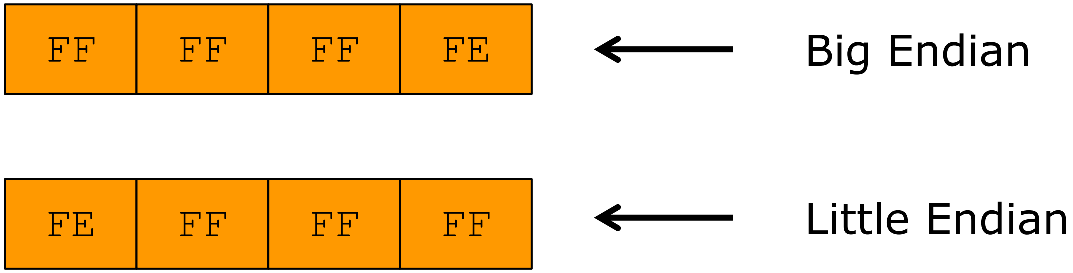
00	00	01	02	Big Endian
02	01	00	00	Little Endian

REMEMBER: Our AMD 64 machines are *little endian!*

Example: negative number

- **We usually think of an integer variable as a single value:**

```
int myint = (-2);
```



FF FF FF FE ← Big Endian

FE FF FF FF ← Little Endian

Can we ever observe the difference?

```
int
main(int argc, char *argv[])
{
  (void) argc;
  (void) argv;

  int pos = 258;
  int neg = (-2);
  float float12 = 12.0;
  float floatneg12 = (-12.0);

  printf("The bytes in memory for signed integer %d are ", pos);
  printbytes(&pos, sizeof(pos));
  printf("\n");

  printf("The bytes in memory for signed integer %d are ", neg);
  printbytes(&neg, sizeof(neg));
  printf("\n");
  printf("The bytes in memory for float %f are ", float12);
  printbytes(&float12, sizeof(float12));
  printf("\n");

  printf("The bytes in memory for float %f are ", floatneg12);
  printbytes(&floatneg12, sizeof(floatneg12));
  printf("\n");
}
```

```
/*
 * Print bytes in memory in hex
 */
void
printbytes(void *p, unsigned int len)
{
  unsigned int i;
  unsigned char *cp = (unsigned char *)p;
  for (i = 0; i < len; i++) {
    printf("%02X", *cp++);
  }
}
```

RUN THIS PROGRAM
ON OUR MACHINES!!

Output:

```
The bytes in memory for signed integer 258 are 02010000
The bytes in memory for signed integer -2 are FEFFFFFF
The bytes in memory for float 12.000000 are 00004041
The bytes in memory for float -12.000000 are 000040C1
```

Summary

Do we care about "endianness"?

- **Mostly, we don’t worry about it…variables generally work as you would expect**
- **When we store data *in memory* or *externally* (on disk, in a network packet), the *endianness* matters**
- **Times you care most:**
  - When writing numeric variables or arrays from memory *to files*
  - When writing numeric variables or arrays from memory *to a network*
  - *In these cases, you and the reader must agree on byte order*
- **Note that HW4 specifies the endianness of the output file you must produce!**

- **When we store data *in memory* or *externally* (on disk, in a network packet), the *endianness* matters**

How did this happen?

- **Both ways work**
- **Many people feel big-endian is most natural, but…**
- **There are some advantages for little-endian:**
  - Regardless of int, long ,etc, you always consistently address the low order byte with pointers.
  - *A simple addition circuit can work from low addresses to high, doing addition or subtraction in the natural way.*
- ***Imagine writing a BigNum package…you would have to manage the storage of the digits in some order***