Module 3. Of Integers and Endians & Floating Point Representations

Topics

- Memory organization and endianness
- More data representations: overview of floating point

Last Time

- C programming basics
- Data representations for characters

Warmup: try the following…

```
int z = 0x4007;

// a. What is the size of z (in bytes)?
// b. In C, how is z stored (unsigned, sign-magnitude, 2's comp)?

if (z & 0x8000) {
    alpha();
} else {
    beta();
}

// c. Based on the value of z, which function would get called?
```

Memory organization

What does it mean to type "`int a`" in C? This is called variable declaration, which allocates space in the program's memory to store an int.

What do we mean by memory? You can think of memory as a big table of "addresses" that each map to a certain piece of data. This data could be a variable (as above), or it could be a piece of code, a portion of the hardware, etc., but for now let's focus on variables.

On the MSP430, addresses are 16-bits long, and each address refers to one byte.

Recall that the MSP430 is a 16-bit architecture,

Unfortunately, this is no longer completely true! Newer MSP430 variants (like ours MSP430F5529) utilize 20-bit addresses. Why?

Laying out variables in memory

When you declare variables in your program, they are arranged in memory starting at a certain address. For now, it is sufficient to know that variables in `main` start at address 0x4400. We will discuss why in an upcoming lecture.

When variables are declared, they are (usually) arranged in order from this starting address. For example:

```
char a = 0x11;
char b = 0x22;
```

…can be arranged in memory as follows:

Address	Data	Variable

In our class, we will arrange memory in a table like the one above, with the starting address at the bottom. We use this convention because we are typically representing variables on the program *stack*, which starts at a fixed base address and grows up.

Endianness: Ordering bytes

In the previous example, we have left out an important detail. How do you store variables that are larger than a byte?

As declared on the MSP430, a long is has a size of four bytes:

```
long v = 0xAABBCCDD; // AAh is the  most significant byte (MSB), and
                     // DDh is the least significant byte (LSB)
```

For multi-byte variables, we have a choice–do we arrange the data with the least significant byte first, or with the most significant byte first? Which is correct? Does it matter?

This concept is known as *endianness*, which governs how a processor orders bytes in memory. There are two forms of endianness:

Little Endian (LE)

   Little Endian stores the *least significant byte first*, meaning that the memory in this example would be arranged as follows:

Address	Data	Variable
0x4403	AA	
0x4402	BB	V
0x4401	CC	
0x4400	DDh	

Big Endian (BE)

   Big Endian stores the *most significant byte first*, as follows:

Address	Data	Variable
0x4403	DD	
0x4402	CC	V
0x4401	BB	
0x4400	AAh	

Important points on endianness

- Endianness is a fundamental part of the architecture's design. When a processor is designed, it is designed to use a specific byte order–you cannot change this with a compiler setting.
- Is one endianness better than the other? No, they simply reflect different design choices.
- Big endian is read "left to right", which is intuitively easier to read for those accustomed to languages written left to right
- Little endian makes it easier to slice out small portions of a variable (eg, what if you only want the first byte of a long?)

When will you deal with endianness?

Endianness becomes especially important when you need to transfer data between different architectures. Examples include any stored data format or network protocol.

More memory layout: Arrays

How do arrays work, anyway?

In C, we can declare arrays and use them as follows:

```
// Declare an array of 5 bytes
char arr[5];
// Declare an array of 5 bytes, and initialize it (set it with some initial values)
char arr[5] = { 0xAA, 0xBB, 0xCC, 0xDD, 0xEE };

// You can access elements of an array by "indexing" into it
// In C, array indexes start at 0
char c = arr[0]; // The first element
char d = arr[4]; // The last element (arr has size of 5, so last index is 5 - 1 = 4
```

You can think of the elements of the array laid out like this:

Index	0	1	2	3	4
Element					
Value					

Why is it important that array elements are contiguous?   (And must contain elements of the same type?)

What would happen if we tried to get the 6th element of `arr`?

How endianness affects arrays (or rather, how it does not)

A fundamental property of arrays is that their elements are stored contiguously in memory in order of their index (as discussed above).
Endianness does not change the order of array <u>elements</u>.

For example, if we laid out the array from the above example on a Big Endian (BE) and a Little Endian (LE) system, it would look like this:

Address	BE	LE
0x4404		
0x4403		
0x4402		
0x4401		
0x4400		

However, endianness *does* affect the ordering of the bytes in each element of the array! In the previous example, the elements were just 1 byte each!

Example: an array of `int`s:

```
int iarr[2] = {0x1122, 0x3344};
```

Here, the memory would be organized as follows:

Address	BE	LE
0x4403		
0x4402		
0x4401		
0x4400		

Using addresses as data

We can also have variables that contain memory addresses. These are called *pointers*.

You can get the address of a variable with the *"address-of" operator* (&):

```
long v = 0x11223344;
long *pv = &v;
```

In this example, we say that `pv` is declared as the type "pointer to long," which is indicated by the "*" before the name `pv`.

How big is `pv`?

,

What is the value of `pv`?

We can lay out these variables in memory as follows:

Address	BE	LE
0x4405		
0x4404		
0x4403		
0x4402		
0x4401		
0x4400		

How big is a pointer?

A pointer is the size of a memory address for a given architecture. On the MSP430, an address has a size of 2 bytes (16 bits).

Type	Size (bytes)
int	
long	
char	
long long	

Type	Size (bytes)
int *	
long *	
char *	
long long *	

This is one way in which pointers are powerful: a pointer can represent a larger data structure in the program—by passing around the pointer, we can avoid copying or moving the larger data structure.

How are pointers used with arrays?

Whenever you use arrays, you use pointers. Consider the following example:

```
int iarr[10];
int i = iarr[5];
```

When you index into an array, the program actually does the following:

```
int i = *(iarr + 5); // Equivalent to writing iarr[5]
```

Here, the * is the *dereference operator*, which **gets the value at the given address**. This is called *dereferencing* the pointer—it is the opposite of the *address-of* (&) operator.

Working with Pointers

Pointer math: When performing arithmetic operations on pointers, the address changes in increments based on the type of the pointer.

```
// Example 1: array of char
char carr[4];
// How big is the array?

// Say the starting address is 0x4400, what is the address of carr[3]?



```

```
// Example 1: array of int
int carr[4];
// How big is the array?

// Say the starting address is 0x4400, what is the address of iarr[3]?



```

Passing arrays: Further, when you use the name of an array (either to store or pass to a function), you are *passing a pointer to the first element of the array*. This is the "starting point" of the array used as input to calculate the index.

```
int *ptr = iarr;      // Could also write &iarr[0]
do_thing(iarr, 10); // Same here

void do_thing(int* arr, int size) { // Function takes pointer to array (+ size)
    // . . .
}
```

Memory organization example

Here's a larger example of memory organization. How would we organize the following variables?

```
unsigned int a = 0x00FF;
long int b[2] = { 65540, -5 };
char c = 'c'; // 'c' = 0x43
```

How many bytes of memory do we require?

So using the above information, we can make our table:

Address	BE	LE
0x440C		
0x440B		
0x440A		
0x4409		
0x4408		
0x4407		
0x4406		
0x4405		
0x4404		
0x4403		
0x4402		
0x4401		
0x4400		

How do you represent fractional numbers in binary form?

So far, we have only expressed integer values in binary. There are two conventions for representing fractions: fixed point and floating point.

Fixed Point

In a given data type, we can define a binary "radix point", which is a fixed point that denotes fractional bits.

In this format, the precision of the number is defined by the number of fractional bits.

For example, 4 fractional bits $= 2^{(-4)} = 0.0625$ is the smallest fraction you can represent

Often, fixed-point representations are stored in scaled form as integers–it's up to you (the programmer) to treat them as fixed-point values.

Floating Point

Floating point is an IEEE standard used to approximate real-valued numbers to a certain number of decimal places.

There are two forms, *single precision* (32 bits) and *double precision* (64 bits). Each representation has three components:

- A sign bit (S)
- An exponent (E)
- A fractional part (F), which is also called the "mantissa" or "significand"

Format for single precision: S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
Exponent is 8 bits, fractional part is the remainder (23 bits)

$$\text{Value} = (-1)^S * 2^{(E-127)} * (1.F)$$

Example: Floating point to Decimal
What is the decimal equivalent of the floating point variable CAAA0000h?

Some features of floating point:

- Effective range: approximately +/- 10^{38}
- Single precision has ~7 decimal digits of precision
  - Double precision and others have more
- Special representations for +/- infinity, NaN
- Standard has conventions for rounding, normalization, etc.

Example: Decimal to floating point

Represent -5.375 as a single-precision floating-point number.