Lecture 4: Integer Math

ME/AE 6705

Introduction to Mechatronics

Dr. Jonathan Rogers

Georgia Tech

Lesson Objectives

- Understand various numbering systems used in microcontroller computation (decimal, binary, hexadecimal)
- Be able to convert between each form
- Understand representation of negative numbers
- Understand and be able to perform calculations on integer binary and hexadecimal numbers
- Understand integer types used in microcontroller programming

Georgia Tech

Numbering Systems

- There are many ways to represent a number in written form
  - Decimal
  - Binary
  - Hexadecimal
  - Binary Coded Decimal
- Digital devices represent numbers differently than humans due to the two-state (on-off) nature of their computation scheme

Decimal Numbering System

- Decimal numbers are represented using Base 10
  - Each placeholder represents a constant times a power of 10

$$235 = \mathbf{2} \times 10^2 + \mathbf{3} \times 10^1 + \mathbf{5} \times 10^0$$

  - Digits (constants) in each placeholder range from 0 to 9
  - Works for fractions too:

$$27.48 = \mathbf{2} \times 10^1 + \mathbf{7} \times 10^0 + \mathbf{4} \times 10^{-1} + \mathbf{8} \times 10^{-2}$$

Georgia Tech

Decimal Numbers (Base 10)

- When adding two numbers, if sum of the digits are greater than or equal to the base, then carry a digit to the next place holder

$$\begin{array}{r} \,1 \\ \,2\;3\;5 \\ +\;1\;4\;6 \\ \hline 3\;8\;1 \end{array} \qquad \begin{array}{r} \,1 \\ \,5 \\ +\;6 \\ \hline 1\;1 \end{array}$$

Georgia Tech

Binary Numbers (Base 2)

- In binary numbers, each placeholder represents a constant times a power of 2
- Valid digits for each placeholder are 0 and 1

Number	Decimal	Binary
2^0	1	00000001
2^1	2	00000010
2^2	4	00000100
2^3	8	00001000
2^4	16	00010000
2^5	32	00100000
2^6	64	01000000
2^7	128	10000000

Georgia Tech

Binary Numbers (Base 2)

- Like decimal, we can represent any number using binary
- Some examples:

$$5 = \mathbf{1}\times 2^2 + \mathbf{0}\times 2^1 + \mathbf{1}\times 2^0 \quad \rightarrow 101 \quad \text{in binary}$$

$$27 = \mathbf{1}\times 2^4 + \mathbf{1}\times 2^3 + \mathbf{0}\times 2^2 + \mathbf{1}\times 2^1 + \mathbf{1}\times 2^0 \quad \rightarrow 11011 \quad \text{in binary}$$

$$255 = \mathbf{1}\times 2^7 + \mathbf{1}\times 2^6 + \ldots + \mathbf{1}\times 2^1 + \mathbf{1}\times 2^0 \quad \rightarrow 11111111 \quad \text{in binary}$$

Georgia Tech

Binary Numbers and Bits

- Placeholders in binary numbers are called **bits**
  - Comes from combining words **b**inary and dig**it**
- Group of 8 bits is called a **byte**
- Group of two bytes is a **word**, group of four bytes is called a **double word**
- Most significant bit (**MSB**) is bit that represents highest power of 2
- Least significant bit (**LSB**) is bit that represents lowest power of two

8-bit binary number: 01101101

MSB LSB

Binary Numbers

- Number of values that an *n*-bit binary number can represent is 2^n
- Thus one byte of memory (8 bits) can hold values from 0 to 255 (which is $2^8 = 256$ total values)
- Likewise, a 16-bit memory location can hold values from 0 to 65535 (which is $2^{16} = 65536$ total values)

- These numbers will be very important when we start talking about types in C programming

Binary Numbers

- Adding numbers in binary follows the same principles as in decimal

Decimal

```
    1
    5
+   6
-----
   11
```

Binary

```
  1 1 1
  1 0 1
+ 1 1 1
-------
  1 1 0 0
```

Binary Numbers

- Subtraction works the same way as in decimal too

Decimal

$$\begin{array}{rr} & 7 \\ - & 5 \\ \hline & 2 \end{array}$$

Binary

$$\begin{array}{rrrr} & 1 & 1 & 1 \\ - & 1 & 0 & 1 \\ \hline & & 1 & 0 \end{array}$$

Georgia Tech

Binary Numbers

- Multiplication of binary numbers

			1	1	1	Decimal 7
		×	1	0	1	Decimal 5
			1	1	1	
		0	0	0	0	
+	1	1	1	0	0	
1	0	0	0	1	1	Decimal 35

Note: Division works in a similar manner.

Example: Binary Numbers

- Multiply binary numbers 1010 and 1101 using the procedure on the previous slide. What is result in binary and decimal?

Fractions in Binary

- So far we have only considered whole numbers. Like in decimal, fractions work by multiplying each place holder to the right of the decimal point by increasing negative powers of 2

Number	Decimal	Binary
2^{-1}	0.5	0.1000000
2^{-2}	0.25	0.0100000
2^{-3}	0.125	0.0010000
2^{-4}	0.0625	0.0001000
2^{-5}	0.03125	0.0000100
2^{-6}	0.015625	0.0000010
2^{-7}	0.0078125	0.0000001

Georgia Tech

Fractions in Binary

- Consider 1/3 in binary. To find its representation, divide 1 by 3.

```
          0 . 0  1  0  1
       ________________
 1 1  |  1 . 0  0  0  0
       -      1  1
       ___________
              1  0  0
          -      1  1
          ___________
                 1  0  0
```

Fractions in Binary

- So 1/3 = 0.010101… in binary

$$\frac{1}{3} = 0.010101\ldots = \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \ldots$$

$$= \sum_{n=1}^{\infty} \left(\frac{1}{4}\right)^n$$

- Recall the power series:

$$\sum_{n=0}^{\infty} a^n = \frac{1}{1-a} \quad \text{if} \quad a \le 1 \quad \Rightarrow$$

$$\sum_{n=1}^{\infty} \left(\frac{1}{4}\right)^n = \sum_{n=0}^{\infty} \left(\frac{1}{4}\right)^n - 1$$

$$= \frac{1}{1-(1/4)} - 1 = \frac{1}{(3/4)} - 1 = \frac{4}{3} - 1 = \frac{1}{3}$$

Negative Numbers in Binary

- Negative numbers represented using method called "two's complement"
- Two's complement may be found by
  - Flipping each digit
  - Adding 1 to final result
- Consider 5 represented by 8 binary digits: 00000101
  - Complement:

    11111010
  - Add 1:

    11111011 = -5 in binary

Negative Numbers in Binary

- Example: Add 12 to -5

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ + & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}$$

  - Truncate to 8 digits

$$00000111 = 7$$

- Subtraction on microprocessors is usually achieved by adding two's complement of subtracted value

Negative Numbers in Binary

- Number of ***signed*** values that an *n*-bit binary number can represent is still 2^n
  - But instead of ranging from 0 to 2^n -1, it ranges from -2^{n-1} to 2^{n-1}-1
  - So we can still represent same number of values, but range is moved and approximately centered at zero

- For example, an 8-bit signed binary number can represent values from -128 to 127 (or -2^7 to 2^7-1)

Hexadecimal Numbers (Base 16)

- When evaluating contents of large memory locations (32 bits), it is cumbersome to write out all 32 1's and 0's
- Instead, hexadecimal (hex) commonly used each 8-bit value using only two digits
- In hex, each placeholder represents a constant times a power of 16
- There are 16 valid digits. Counting is defined as:

0 1 2 3 4 5 6 7 8 9 A B C D E F

A = 10 B = 11 C = 12 D = 13 E = 14 F = 15

Hexadecimal Numbers

- Because hex uses some of the same digits as decimal (1, 2, 3, etc), hex numbers can be easily confused with decimal numbers
- Thus we usually write “0x” in front of hex numbers to denote that they represent hex, not decimal
- Example: Compute hex representation of decimal number 19

$$19 = \mathbf{1} \times 16^1 + \mathbf{3} \times 16^0 = 0\text{x}13$$

Comparison of Representations

Decimal	Binary	Hexadecimal
0	0	0x0
1	1	0x1
5	0101	0x5
13	1101	0xD
15	1111	0xF
17	10001	0x11
128	1000000	0x80
255	11111111	0xFF
65438	1111111110011110	0xFF9E
-12	11110100	0xF4
-394	111001110110	0xE76

Example: Number Conversions

- Write the number 336 and -336 decimal in binary and hexadecimal.

Example: Interpreting Binary Numbers

- You find that the value in a certain memory address is 11010011. You would like to find the decimal equivalent of this number.
  - Are there multiple ways to interpret what decimal value this number represents? If so, how many ways can you interpret it?
  - Find the decimal equivalent(s) of this number.

Assembly Code Math

- Every program that you write in C/C++ gets compiled into *assembly language*
  - Assembly is low level language consisting of simple instructions. Some examples:
    - *Add two numbers (ADD)*
    - *Subtract two numbers (SUB)*
    - *Multiply two numbers (unsigned, MUL) (signed, MULS)*
    - *Divide two numbers (unsigned UDIV) (signed, SDIV)*
    - *Logical AND of two numbers (AND)*
  - All C programs can be turned into (long) assembly program!

Georgia Tech

Integer Math

- What happens when we need to add 10 to 250 using one byte?
- Result:

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ + & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

- Truncate to one byte: 00000100 = 4
  - Wrong! This is called *overflow.*

Integer Math

- A single byte (interpreted as unsigned) can only represent numbers 0 to 255
- **Overflow** occurs when two numbers added together exceed the capacity of the defined storage
  - Also known as a “carry”
- **Underflow** occurs when numbers subtracted from one another exceed the capacity of the defined storage
  - Also known as a “borrow”

Integer Math

- How do you define appropriate storage when programming?
- Depends on the type you specify for a variable
- C99 (version of C used for MSP432) has built in definitions that:
  - Define how many bytes are used to store a variable
  - Define whether variable should be interpreted as unsigned or signed

Variable Types in C

C Data Type	C99 keyword	Precision & Interpretation	Range
unsigned char	uint8_t	8-bit (unsigned)	0 to 255
signed char	int8_t	8-bit (signed)	-128 to 127
unsigned short	uint16_t	16-bit (unsigned)	0 to 65535
short	int16_t	16-bit (signed)	-32768 to 32768
unsigned int	unsigned int	32-bit (unsigned)	0 to 4294967295
int	int	32-bit (signed)	-2147483648 to 2147483647
unsigned long	uint32_t	32-bit (unsigned)	0 to 4294967295
long	int32_t	32-bit (signed)	-2147483648 to 2147483647
float	float	32-bit float	$+/-10^{-38}$ to $+/-10^{38}$
double	double	64-bit float	$+/-10^{-305}$ to $+/-10^{305}$

Unsigned Variable Range

- Remember that an *n*-bit unsigned variable can represent numbers from 0 to 2^n-1
  - unsigned char (8 bit): 0 to 255
  - unsigned short (16 bit): 0 to 65535
  - unsigned long (32 bit): 0 to 4294967295

Signed Variable Range

- Remember that an *n*-bit signed variable can represent numbers from $-(2^{n-1})$ to $2^{n-1}-1$

  - char (8 bit): -128 to 127
  - short (16 bit): -32768 to 32767
  - long (32 bit): -2147483648 to 2147483647

Variable Types

- Question. Suppose you decide to just make all your variables have the largest storage possible (declare them all as longs) so you don't need to worry about overflow or underflow. Why might this be a bad idea?
  - Why is this especially important for microcontrollers programming (compared to standard PC programming)?

Variable Declarations

- In C programs, variables are defined by first specifying a variable type, and then a variable name
  - All declarations must end with a ";"
  - Multiple variables of the same time can be made on a single line
- Example: Define "count" to be a single byte, and "a", "b", and "c" as signed 16-bit integers

```
unsigned char   count ;
short            a, b, c ;
```


Variable Scope

- Where a variable is declared in the program determines when the variable can be accessed
- Programs are divided into functions
  - Every program must have a function called “main”
- Variables that are defined outside all functions (at top of file) can be accessed anywhere in program
  - These are called **global variables**

```
char    count ;
int     i, j ;

int main(){
        [your code]
        return 0 ;
}
```


Global variables

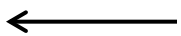
Variable Scope

- When variables are defined inside a function, they can only be used in that function. These are called **local variables.**
  - Undefined outside that function

```
int     i, j ;

void my_function(){

        char    count;
        [your code]
        return ;
}
```



Global variables

Local variable

Variable Scope

- What are the benefits of using local variables vs global variables from a memory usage standpoint?

- What are the benefits of using local variables vs global variables from a software development standpoint?

Georgia Tech

Math Operators in C

- C offers various operators to perform basic arithmetic

Operator	Name	Example
+	Addition	C = A + B
+=	Addition assignment (x+=y equiv. to x = x+y)	c += b
-	Subtraction	c = b – a
-=	Subtraction assignment (x-=y equiv. to x=x-y)	c -= b
*	Multiplication	j = count*4
*=	Multiplication assignment (x*=y equiv. to x=x*y)	i *= k
/	Division	f = r / 13
/=	Division assignment (x/=y equiv. to x=x/y)	f /= 13
++	Increment by one (++j is equiv. to j = j + 1)	i++
--	Decrement by one (--j is equiv. to j = j – 1)	--i

Bitwise Operations in C

- C also provides bitwise operators which compute logical bitwise operations on two binary numbers
  - Used extensively in microcontroller programming
- Some common examples

Operator	Name	Description
`&`	Bitwise AND	Performs AND operation on each corresponding bit of two arguments and returns result
`|`	Bitwise OR	Performs OR operation on each corresponding bit of two arguments and returns result
`^`	Bitwise Exclusive OR	Performs Exclusive OR operation on each corresponding bit of two arguments and returns result
`<< or >>`	Right shift or Left shift	Shift bits of argument 1 right/left by argument 2 places
`!`	Logical Negation	Returns 1 if input is 0, returns 0 otherwise
`&&`	Logical And	Returns 1 if inputs are all nonzero, 0 otherwise
`||`	Logical Or	Returns 0 if all inputs are zero, 1 otherwise
`<, >, ==`	Greater than, less than, logical equal	Returns 1 if argument one is (gt, lt, equal to) argument two, 0 otherwise

Example: Bitwise Operations

- Some lines from a given microcontroller program read:

```
unsigned char   count = 8 ;
unsigned char   mult = 3 ;

q = count & mult ;
r = count | mult ;
w = mult ^ count ;
x = count >> 2
```

- What are the numerical (decimal) values of q, r, w, and x?

Georgia Tech

Overflow

- Let's revisit the concept of overflow. Consider following program:

```
unsigned char   A, B, C ;

int main(){

        A = 255 ;
        B = 1 ;
        C = A + B ;

        return 0 ;
}
```

- We expect the result in C to be 256. What is the actual result?

Overflow

- We know that A + B = 256

$$\begin{array}{rcccccccc} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ + & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

- But since *C* is defined as having 8 bits of storage, processor will only store the first 8 bits
  - Thus ***C* = 0**
  - Similarly, if *B* = 2 result in *C* will be 1, not 257

Incorrect Variable Types

- Sometimes solution will work correctly despite incorrect variable types
  - This is dangerous: Just because incorrect definitions work for some numbers does not mean it will always work
- Example:

```
unsigned char   a, b, c, d ;

int main(){

        a = 2 ;
        b = 4 ;
        c = a - b ;
        d = c + 2 ;
        return 0 ;
}
```


Incorrect Variable Types

- Example continued…
  - Result of *a* – *b* or *a* + (-b) is:

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ + & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ \hline & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{array}$$

  - Taking two's complement of the answer we find -2
    - One's complement: 00000001
    - Add one: 00000010
  - Processor will correctly store the solution in *c* despite thinking it is 254 (since *c* was originally declared unsigned)

Incorrect Variable Types

- Example continued…
  - Now go to the next line, where *d* = *c* + 2

$$\begin{array}{rcccccccc} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ + & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

  - In this case, *d* is stored correctly as 0 despite an overflow because only the first 8 bits of the answer are stored
  - If *d* had been declared as an unsigned short, the overflow would have caused us to get the wrong answer for *d*

Integer Division

- Integer division will always truncate the result to an integer (always rounding down)
- Examples:

$$\frac{6}{3} = 2 \qquad \frac{3}{3} = 1$$

$$\frac{5}{3} = 1 \qquad \frac{2}{3} = 0$$

$$\frac{4}{3} = 1 \qquad \frac{53}{54} = 0$$

Georgia Tech