Representation of Integers (lecture)

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Integer Representation

- A computer needs to store integers in memory/registers
- Stored using different numbers of bytes (1 byte = 8 bits):
  - 1-byte: “byte”
  - 2-byte: “half word” (or “word”)
  - 4-byte: “word” (or “double word”)
  - 8-byte: “double word” (or “paragraph”, or “quadword”)
  - Different computers have used different word sizes, so it’s always a bit confusing to just talk about a “word” without any context
- Regardless of the number of bytes, integers are stored in binary
- Integers come in two flavors:
  - Unsigned: values from 0 to 2^b-1
  - Signed: negatives values, with about the same number of negative values as the number of positive values
- You can actually declare variables as signed or unsigned in some high-level programming languages, like C/C++
  - Java doesn’t, but since Java8 there is an API for it!

Sign-Magnitude

- **Storing unsigned integers is easy:** just store the bits of the integer's binary representation
- **Storing signed integer raises a question:** how to store the sign?
- One approach is called sign-magnitude: reserve the leftmost bit to represent the sign

  00100101 denotes $+ 0100101_2$

  10100101 denotes $- 0100101_2$

- It's very easy to negate a number: just flip the leftmost bit
- Unfortunately, sign-magnitude complicates the logic of the CPU (i.e., ICS331-type stuff)
  - There are two representations for zero: 10000000 and 00000000
  - Some operations are thus more complicated to implement in hardware

One’s complement

- Another idea to store a negative number is to take the complement (i.e., flip all bits) of its positive counterpart
- Example: I want to store integer -87
  - $87_{10} = 01010111_2$
  - $-87_{10} = 10101000$
- Simple, but still two representations for zero: 00000000 and 11111111
- It turns out that computer logic to deal with 1’s complement arithmetic is complicated
- Note: it’s easy to compute the 1’s complement of a number represented in hexadecimal
  - let’s consider: 57_{16}
  - Subtract each hex digit from F: F-5=A, F-7=8
  - 1’s complement of 57_{16} is $A8_{16}$

Two's complement

- While sign-magnitude and 1's complement were used in older computers, nowadays all computers use 2's complement
- Computing the 2's complement is done in **two steps**:
  - Compute the 1's complement of the positive number
  - Add 1 to the result
  - The gives the representation of the negative number
- Example: Let's represent -87_{10}
  - First, start with the >0 version of the number: $87_{10} = 57_{16}$
  - "Flip" the bits or hex digits to compute the one's complement: $A8_{16}$
  - Add one: $A9_{16}$
- Let's invert again to check we get back to the positive
  - We start with: $A9_{16}$
  - Flip the digits (one's complement): 56_{16}
  - Add one: 57_{16}, which represents 87_{10}

Two’s complement

- Note that when adding 1 in the second step a carry may be generated but is ignored!
  - Difference between arithmetic and computer arithmetic
  - When adding two X-bit quantities in a computer one **always** obtains another X-bit quantity (X=8, 16, 32, …)
- Example: Computing 2’s complement of 00000000
  - Take the invert: 11111111
  - Add one: 00000000   with a carry generated but it’s dropped!
    - Should be a 9-bit quantity: 100000000
- Therefore 0 has only one representation: a signed byte can store values from -128 to +127 (128 <0 values, and 128 >=0 values)
- It turns out that 2’s complement makes for very simple arithmetic logic when building ALUs
- **From now on we always assume 2’s complement representation**
- Important: The leftmost bit indicates the sign of the number (0: positive, 1: negative)
  - In hex, if the left-most “digit” is 8, 9, A, B, C, D, E, or F, then the number is negative, otherwise it is positive

Ranges of Numbers

- For 1-byte values
  - Unsigned
    - Smallest value: 00_{16} (0_{10})
    - Largest value: FF_{16} (255_{10})
  - Signed
    - Smallest value: 80_{16} (-128_{10}) (note that if you subtract 1 you get 79_{16}, which is positive and likely is a bug in your program - you should use more bits if you want numbers smaller than -128_{10})
    - Largest value: $7F_{16}$ ($+127_{10}$)
- For 2-byte values
  - Unsigned
    - Smallest value: 0000_{16} (0_{10})
    - Largest value: $FFFF_{16}$ ($65{,}535_{10}$)
  - Signed
    - Smallest value: 8000_{16} ($-32{,}768_{10}$)
    - Largest value: $7FFFn_{16}$ ($+32{,}767_{10}$)
- etc.

The Magic of 2's Complement

- Say I have two 1-byte values, A3 and 17, and I add them together:

  $A3_{16} + 17_{16} = BA_{16}$

- If my interpretation of the numbers is unsigned:
  - $A3_{16} = 163_{10}$
  - $17_{16} = 23_{10}$
  - $BA_{16} = 186_{10}$
  - and indeed, $163_{10} + 23_{10} = 186_{10}$
- If my interpretation of the numbers is signed:
  - $A3_{16} = -93_{10}$
  - $17_{16} = 23_{10}$
  - $BA_{16} = -70_{10}$
  - and indeed, $-93_{10} + 23_{10} = -70_{10}$
- So, as long as I stick to my interpretation, the binary addition does the right thing assuming 2's complement representation!!!
  - Same thing for the subtraction

The Task of the (Assembly) Programmer

- The computer simply stores data as bits
- **The computer internally has no idea what the data means**
  - **It doesn't know whether numbers are signed or unsigned**
- We, as programmers, have precise interpretations of what bits mean
  - "I store a 4-byte signed integer", "I store a 1-byte integer which is an ASCII code"
- When using a high-level language like C/C++, we say what data means
  - "I declare x as an int and y as an unsigned char"
- When writing assembly code, we don't have "data types"
- **But** we have many instructions that operate on all types of data
- It's our responsibility to use the instructions that correspond to the data
  - e.g., if you use the "signed multiplication" instruction on unsigned numbers, you'll just get a wrong results but no warning/error
- This is one of the difficulties of assembly programming

Conclusion

- We’ll come back to numbers and arithmetic when we use arithmetic assembly instructions
- But for now you must make sure you have solid mastery of the material in this module
- We’ll do in-class practices to make sure we’re all on the same page