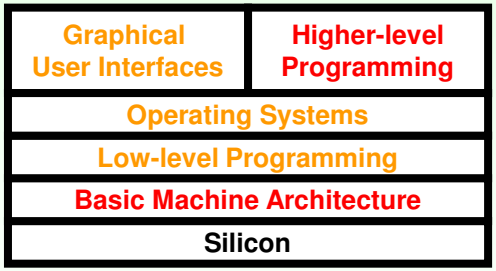
CSCU9V4 Systems

Systems Lecture 3

Integer Arithmetic



Integer Arithmetic

Representing and manipulating integers:

- Addition
- Subtraction
- Negative integers
- Twos complement

Integer Addition in Different Bases

- Addition just follows the familiar rules
- Example in decimal, binary, and hexadecimal
  - 106 + 27 = 133

106	0110 1010	**4Ax**
+27	+0001 1011	+**1Bx**
133	1000 0101	**65x**

- But what happens if the result is too large? E.g. suppose that we are holding numbers in one byte each, and we add (e.g.)
  - 128 + 128 = 256

1000 0000	128	**80x**
1000 0000	128	**80x**
10000 0000	256	**100x**

  - then the result is too big to hold in one byte.

Integer Overflow

- Only a limited range of integers can be stored in an N-bit word.
  - 0 to 2^N -1
- When such an operation is performed, where the result is too large to store, this is called "integer overflow".
- The machine does not treat this as an error, it just stores the truncated result.
- **AND** the machine "sets the **carry** flag" so that the programmer can test for integer overflow if they wish.
  - Enables coding with larger integer lengths
  - Good for Diophantine equations
  - And for developing prime-number based codes

We will look at this in more detail later in the course

Subtraction and Negative Integers

- Subtraction is straightforward to perform, as

$$x - y = x + (-y)$$

- But we need to be able to represent negative numbers

- We look at three schemes (the first two only very briefly, for comparison):
  - Sign magnitude
  - Excess notation
  - **Two's complement**

- First we decide on a word length for our numbers. We could use any sensible word length, but for the rest of the lecture we will use **one-byte** numbers.

Sign-magnitude

- We could use the left-hand-most bit (the "most significant bit" or *msb*) for the sign (say, 0 for + and 1 for -), and the remaining bits for the number
- e.g.
  - `0010 0001` represents +33
  - `1010 0001` represents -33

- We quickly see that this convention has undesirable properties.
  - For example, with addition as earlier, adding +33 and -33 gives -66 !
  - Also having `0 0000000` and `1 0000000` both represent zero is a bit weird.
    - +0 and -0 ? Neither useful nor efficient

Using Excess-k (or offset binary)

- We take the value to be stored, add an *excess* and store the result
- Working in one byte, the excess would be 128
  - (in general: working in *n* bits, the excess is 2^{n-1})

NOT 2^n -1

- So, working in one byte, we represent the numbers {-128 ... 127} using the unsigned numbers {0 ... 255}
  - for example
    - +33 is represented by unsigned 161 (1010 0001)
    - -33 is represented by unsigned 95 (0101 1111)

Addition doesn't work quite right…

- But addition still isn' t quite right: if we add two excess-notation numbers we double the excess, so we need to remember to subtract the excess from the answer

- e.g. (in decimal)
  - +33 -> 161
  - −33 -> + 95
  - 256

  - 128
    128 -> 0

- We can get round this problem by a clever trick that exploits integer overflow (Slide 3), giving ..

Two's complement notation

- This is how computers almost always represent integers
- We cover this by looking at it in several ways, to understand
  - how the method achieves its effects
  - how it is implemented
- We shall also see how it works in decimal, binary and hex

- Basic idea: for 8 bits, use an excess of **256**, not 128
- Storing a number in two's complement:
  - we take the value to be stored in *n* **(8)** bits, add an excess of 2^n **(256)**
  - and store the result (*which may be truncated*)

Two's complement (continued)

- Working in one byte, this has the effect that we store the numbers {0 ... 127} unchanged (because of the truncation)
  - $33 + 2^n = 33 + 256 = 289$ decimal
    $= 1\ 0010\ 0001$ binary
  - after truncation we store this as 0010 0001 (33 decimal)
- But negative numbers {-128 ... -1}, when 256 is added, give values in the range {128 ... 255}
  - which can be stored in one byte, so we just store them as they come as no truncation will occur
    - $-33 + 2^n = -33 + 256 = 223$ decimal
      $= 1101\ 1111$ binary

Two's complement (continued)

- The effects of this procedure are:
  - 127 -> (127 + 256) -> 127 (0111 1111)
    126 -> (126 + 256) -> 126 (0111 1110)
    ...
    33 -> (33 + 256) -> 33 (0010 0001)
    ..
    1 -> (1 + 256) -> 1 (0000 0001)
    0 -> (0 + 256) -> 0 (0000 0000)
  - -1 -> (-1 + 256) -> 255 (1111 1111)
    -2 -> (-2 + 256) -> 254 (1111 1110)
    -3 -> (-3 + 256) -> 253 (1111 1101)
    ...
    -33 -> (-33 + 256) -> 223 (1101 1111)
  - ...
  - -127 -> (-127 + 256) -> 129 (1000 0001)
  - -128 -> (-128 + 256) -> 128 (1000 0000)

Two's complement (continued)

- Things to notice about two's complement from the previous slide:
  - The most significant bit (msb), which is the bit at the left hand side, is equal to one (is set) if and only if the number is negative
    - This is very useful for testing whether a number is negative or not!
    - The msb effectively acts as the negative of the value you would expect it to be and the rest of the number is added to this value to give you the final result: 1000 0011 -> -128 +3 = -125
  - If a stored number n $\leq$127 it represents n
  - If a stored number n $\geq$128 it represents (n - 256)
  - Now we find that normal addition works perfectly provided we truncate the result when necessary

Addition in Two's Complement

- Addition examples:
  -

Problem	Represented as	Ans	Trunc	Really Means
-33 + 33	-> 223 + 33	-> 256	-> 0	-> 0
-1 + 127	-> 255 + 127	-> 382	-> 126	-> 126
-17 + 2	-> 239 + 2	-> 241	-> 241	-> -15

- The only remaining problem is the case where the result falls outside the range {-128 ... 127} (because we cannot store it)
  - -100 + (-100) -> 156 + 156 -> 312 -> 56 -> 56
    100 + 100 -> 100 + 100 -> 200 -> 200 -> -56

- Fortunately these cases can be detected:
- *Integer overflow occurs precisely when the inputs have the same sign and the result has a different sign*

Negation in Two's complement

- For implementation, we also need to be able to *negate* numbers (that is, make them negative, e.g. to obtain -33 we *negate* 33)
- Negating a positive number: *flip all the bits and add 1*
  - e.g.
    -

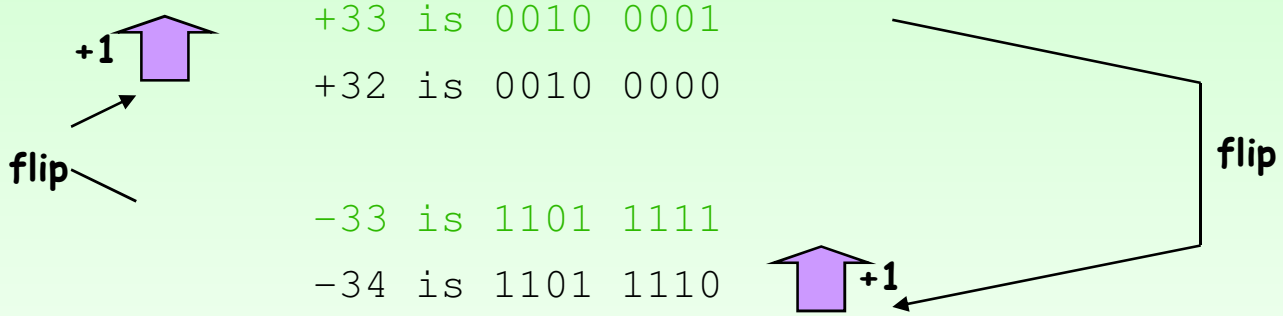
33 is	0010 0001	127 is	0111 1111
(flip)	1101 1110		1000 0000
(add 1)	0000 0001		0000 0001
-33 is	1101 1111	-127 is	1000 0001

  - (compare these results with Slide 10: we do get the same answers!)
- It is easy to see how to reverse the process: to find the positive equivalent of a negative number (one with its msb set) *we subtract 1 and flip all the bits……*BUT!

Negating (Positiving?!) Negative numbers

- In fact the same process (flip the bits and add one) will work both to convert a positive number to a negative one, *and* vice versa

e.g.



- Implementation is easier, as there is no need to do two different conversions for positive/negative numbers.

Two's complement using hexadecimal

- We can write these sums very succinctly in hex: to find the "flipped" hex value, subtract each hex digit from **Fx**
- e.g.
	FFx		**FFx**		**FFx**
33 is	**21x**	127 is	**7Fx**	1 is	**01x**
(flip)	**DEx**		**80x**		**FEx**
(add 1)	**01x**		**01x**		**01x**
-33 is	**DFx**	-127 is	**81x**	-1 is	**FFx**
    - once again, compare these results with Slide 11, we do get the same answers

How can we know?

- …and without knowing the internals of the machine, or those who designed the machine (or who implemented the software – particularly the compiler)
  - And the Java virtual machine if we're talking about Java…?

- The same way we answer many 'systems' questions… TEST for it!

Java example

- Demonstrating that Java uses two's complement:

```
import java.io.*;

public class Complement {
   public static void main (String args[]) {
    short i,j,k; // 16 bit integers from -32768 to 32767
    i = 30000;
    j = 30000;
    k = (short)(i+j);
    System.out.println("k is"+k);
   }
}
```

- Excess is 65536, 30000+30000 = 60000, which represents -5536

Note that in N bits we can hold from -2^{N-1} to $2^{N-1}-1$ values

e.g. -128 to 127 when N = 8 bits

…and in C

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {

    short i = 30000;
    short j = 30000;
    short k = 0;

    k = (short)(i+j) ;

    printf("%d\n", k);

    return (EXIT_SUCCESS);
}
```

- Again, excess is 65536, 30000+30000 = 60000, which represents -5536

End of Lecture