Inf2C Computer Systems

Tutorial 1, Week 3

Solutions

Boris Grot, Paul Jackson, Stratis Viglas

1. **Two's complement.** What decimal number does the two's complement binary number
`1111 1111 1111 1111 1111 1111 1110 0101`
represent?

   Answer.

   The conversion algorithm:

   - *Look at leftmost bit to see if it is positive or negative;*
   - *If positive, convert number from binary to decimal;*
   - *If negative, determine magnitude by:*
     - *Complementing the bits*
     - *Adding 1*
     - *The decimal number is the negative of this number*

   So, in the current example, the number is negative. The complement is:

   `0000 0000 0000 0000 0000 0000 0001 1010`

   which is 26 *in decimal; plus* 1 *equals* 27 *so the number is* -27.

2. **Number representation and addition.** Using 8 bits represent the numbers $+13$ and -4, using both 2's complement and sign-magnitude binary representation. Perform the common binary addition of the numbers, in both representations. What are the results? Are they correct? Based on the above results, comment on the advantages and disadvantages of two forms of representation.

   Answer.

   $+13$ *in* 8*-bit binary is* `0000 1101`; -4 *in signed-magnitude is* `1000 0100` *and in* 2*'s complement is* `1111 1100`. *Adding in signed-mignitude we have* `1001 0001`; *this is* -17 *and incorrect. Two's complement addition gives us* `0000 1001`, *which is* 9 *and the correct result.*

3. **Floating point representation.**

   (a) Compute the equivalent normalised binary number and the corresponding IEEE 754 32-bit representation for the decimal value of 61.

   (b) What decimal number is represented by this single precision float?
   `1 10000001 00100000000000000000000`

   Answer.

   (a) The floating point conversion algorithm:

   - *Convert the absolute value of the decimal number to a binary integer plus a binary fraction;*
   - *Normalize the number in binary scientific notation to obtain the mantissa and the exponent;*
   - *Set the sign bit to* 0 *for a positive number and to* 1 *for a negative number.*

   In this case, 61 *is binary* `111101`*. The number has no fractional part. This needs to be converted into a* $1.x$ *notation, which is* `1.11101` $\times 2^5$*. Therefore the mantissa is* `11101` *(note that the full mantissa is* `1.11101`*, but* `1.` *is implicit in the IEEE 754 representation) and the exponent in excess-*127 *notation is* $127 + 5 = (132)_{10} = (\texttt{10000100})_2$ *and the final representation is:*
   `0 10000100 11101000000000000000000`

   (b) Apply the same conversion algorithm in reverse:

   $$(-1)^{sign} \times 1.mantissa \times 2^{exponent-127}$$

   In this case the number is negative ($sign = 1$*), the exponent is equal to* 129 *and the mantissa is equal to* 0.125*. So the result is* $-1 \times 1.125 \times 2^2 = -4.5$.

4. **Overflow.** Using only 8 bits, perform the arithmetic operation: $115 + 33$ using 2's complement. What is wrong with the result? Discuss what are the conditions that can cause overflow when adding or subtracting two 2's complement numbers. Without looking at the values of the operands, is there a way of detecting overflow while performing the addition? Does it work for both positive and negative overflow? (Hint: consider the carries at the 2 most significant bit positions.)

   Answer.

   The 8*-bit binary representations are:*

   - $115 =$ `0111 0011`
   - $33 =$ `0010 0001`

Adding them, we get `1001 0100`*. This is an incorrect negative result caused by overflow. To detect overflow for 2's complement addition we can look at the carry values coming into and out of the most significant bit. If they are equal (both zero or both one), there is no overflow. If they are different overflow has occurred.*

How does one see that this overflow detection works in general? Consider sign-extending two 8-bit 2's complement words to 9 bits and then adding to get a 9-bit result. In this case we never expect to get an overflow or underflow. The addition looks like

$$\begin{array}{cccc} & c & d & \\ a & a & XXX & XXXX \\ b & b & XXX & XXXX \\ \hline f & e & XXX & XXXX \end{array} \quad +$$

where c and d are the inter-column carries and the Xs are bits we don't care about the values of.

Now, this 9-bit result can only be reduced back to equal 8-bit number when bits e and f are the same. If $f = 1$ and $e = 0$ we have and underflow situation – from the 8-bit addition point of view, and if $f = 0$ and $e = 1$ we have and overflow situation. Observe that the bits f and e are different exactly when the bits c and d are different. Hence we can detect overflow or underflow by looking for these carry bits being different and not actually doing the computation for the addition in the leftmost column of these 9-bit words.

5. **Bit-masking and hexadecimal notation.** Bit masking is an operation used for extracting specific bits from a binary variable. Perform the following operation and give the result as a hex number: `0x5e` AND `0x30` (AND is the bitwise logical "and" operation, *i.e.*, result is 1 only if both bits are 1.) How can you set the fifth bit (from the right hand side) of a binary variable w?

   Answer. *Converting hex numbers to binary is quite simple. Four bits are required for each hex digit and there is a standard mapping between hex digits and binary representation. In this case the conversion is:*

   - `0x5e` = `0101 1110`
   - `0x30` = `0011 0000`

   AND-ing the two numbers gives `0001 0000`

   To set a bit of a variable w you need to create a mask m where only the required bit is set (e.g., by setting the mask to 2^{i-1} – assuming bit numbering starts from the right and an index of 1) and then OR-ing w with m.

   In general, we use:

- *bitwise AND-ing with a mask word to reset (set to 0) selected bits in a word,*
- *bitwise OR-ing with a mask word to set (set to 1) selected bits in a word,*
- *bitwise XOR-ing (exclusive or-ing) with a mask word to invert selected bits in a word.*