Lecture 3: Integer representation

David Hovemeyer

September 2, 2022

601.229 Computer Systems Fundamentals

Integer representation

Representing integers

- We've seen how to represent unsigned (nonnegative) integers
  - Bit string intrepreted as a binary (base 2) number
- How to represent signed integers?
  - Sign magnitude
  - Ones' complement
  - Two's complement
- In examples that follow, we'll use 4-bit words
  - Ideas will generalize to larger word sizes

Desired features for signed representation

What we want in a representation for signed integers:

- About half of encoding space used for negative values
- Each represented integer has a unique encoding as bit string
- Straightforward way to do arithmetic

Sign magnitude representation

Let most significant bit be a sign bit: $\mathbf{0}\rightarrow$positive, $\mathbf{1}\rightarrow$negative

Bit string	value	Bit string	value
0000	0	**1**000	-0
0001	1	**1**001	-1
0010	2	**1**010	-2
0011	3	**1**011	-3
0100	4	**1**100	-4
0101	5	**1**101	-5
0110	6	**1**110	-6
0111	7	**1**111	-7

Downsides: two representations of 0, arithmetic complicated by sign bit

Ones' complement

Ones' complement: to represent `-x`, invert all of the bits of `x`

Bit string	value	Bit string	value
0000	0	1000	-7
0001	1	1001	-6
0010	2	1010	-5
0011	3	1011	-4
0100	4	1100	-3
0101	5	1101	-2
0110	6	1110	-1
0111	7	1111	-0

Downsides: two representations of 0, slightly complicated arithmetic

Sign magnitude and ones' complement are obsolete

- Sign magnitude and ones' complement representations are not used for integer representation by modern computers
  - But, sign magnitude is used in floating point representation
- The rest of this lecture will discuss *two's complement*

Two's complement

Two's complement: in w-bit word, the most significant bit represents -2^{w-1}

E.g., when $w = 4$,

Representation	Bit 3	Bit 2	Bit 1	Bit 0
Unsigned	8	4	2	1
Two's complement	-8	4	2	1

Given bit string 1011,

- Unsigned, 1011 is $8 + 2 + 1 = 11$
- Two's complement, 1011 is $-8 + 2 + 1 = -5$

Two's complement

Two's complement: in w-bit word, the most significant bit represents -2^{w-1}

Bit string	value	Bit string	value
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

Note asymmetry of negative and positive ranges: -8 is represented, 8 isn't

Thinking about two's complement

Useful way to think about a w-bit two's complement representation:

- Bit $w-1$ is the sign bit, $0 \rightarrow$positive, $1 \rightarrow$negative
- If sign bit is 0, usual unsigned interpretation
- If sign bit is 1, bits $w-2 \ldots 0$ indicate the "offset" from -2^{w-1}

Two's complement example

Given $w = 4$, example bit string is $\boxed{1\underline{011}}$

- Sign bit is 1
- Offset from -2^3 is 011, which is 3 (2+1)
- -8 + 3 = -5

So, 1011 represents -5

Clicker quiz

Clicker quiz omitted from public slides

Why two's complement?

The most important advantage of two's complement:

Why two's complement?

The most important advantage of two's complement:

Unsigned addition yields correct result for signed values!

Why two's complement?

The most important advantage of two's complement:

Unsigned addition yields correct result for signed values!

Wow!

Trying it out

Add two 8 bit integer values:

$$00101101$$

Trying it out

Add two 8 bit integer values:

$$
\begin{array}{rr}
 & 00101101 \\
+ & 11111100 \\
\hline
\end{array}
$$

Trying it out

Add two 8 bit integer values:

$$
\begin{array}{rr}
 & 00101101 \\
+ & 11111100 \\
\hline
 & 100101001
\end{array}
$$

Trying it out

As unsigned values:

$$
\begin{array}{rrrl}
 & 00101101 & 45 & \\
+ & 11111100 & 252 & \\
\hline
 & 100101001 & 297 & \text{(truncated to 41)}
\end{array}
$$

Trying it out

As signed two's complement values:

$$
\begin{array}{rrr}
 & 00101101 & 45 \\
+ & 11111100 & -4 \\
\hline
 & 100101001 & 41
\end{array}
$$

Subtraction via addition

- Two's complement negation: invert all bits, then add 1
- Example, negating 5
  - Original value: 00000101
  - Invert bits: 11111010
  - Add one: 11111011
  - Value is -128 + 64 + 32 + 16 + 8 + 2 + 1 = -5
- $a - b$ can be computed as $a + -b$
  - I.e., invert b, then add to a

Sign extension

- Sometimes it is necessary to increase the number of bits in the representation of a signed integer
  - E.g., type cast or implicit conversion of a 16 bit `short` value to a 32 bit `int` value
- In two's complement, this can be accomplished by *sign extension*: replicate the original sign bit as many times as necessary
  - This preserves the numeric value!
  - Processors typically have dedicated instructions to perform sign extension

Sign extension example

Example: extend 4 bit two's complement values 1011 and 0011 to 8 bits

Number of bits	Bit string	Meaning
4	<u>1</u>011	-8 + 2 + 1 = -5
8	**1111**<u>1</u>011	-128 + 64 + 32 + 16 + 8 + 2 + 1 = -5
4	<u>0</u>011	2 + 1 = 3
8	**0000**<u>0</u>011	2 + 1 = 3

Sign extension example program

```
#include <stdio.h>
void printbits(int x, int n) {
  for (int i = n-1; i >= 0; i--) {
    putchar(x & (1 << i) ? '1' : '0');
  }
  putchar('\n');
}

int main(void) {
  short s = -27987;
  int i = (int) s;            // <-- sign extension occurs here
  printf("%*c", 16, ' ');
  printbits(s, 16);
  printbits(i, 32);
  return 0;
}
```

Sign extension example program (output)

```
$ gcc signext.c
$ ./a.out
                1001001010101101
11111111111111111001001010101101
```

Clicker quiz!

Clicker quiz omitted from public slides

Extending unsigned values

Extending the representation of an unsigned value is straightforward: unconditionally pad with 0 bits

Example: 4 bit unsigned value $1011 = 8 + 2 + 1 = 11$

As an 8 bit unsigned value, $\mathbf{0000}1011 = 8 + 2 + 1 = 11$

General observation

In general, increasing the number of bits in the representation of an integer (signed or unsigned) will preserve its value

Truncation

- Truncation: *reducing* the number of bits in the representation of an integer
  - In general, this will lose information and potentially change the value
- Truncation is done by chopping off bits from the left side of the bit string
  - Whatever remains is the new representation

Truncation example

Example: convert signed 8 bit integer -14 to a 4 bit signed integer

Number of bits	Bit string	Meaning
8	11110010	-128 + 64 + 32 + 16 + 2 = -14
4	0010	2

Truncation example program

```
#include <stdio.h>
void printbits(int x, int n) {
  for (int i = n-1; i >= 0; i--) {
    putchar(x & (1 << i) ? '1' : '0');
  }
  putchar('\n');
}

int main(void) {
  short s = -129;
  char c = s;        // <-- truncation occurs here
  printf("s=%d, c=%d\n", s, c);
  printbits(s, 16);
  printf("%*c", 8, ' ');
  printbits(c, 8);
  return 0;
}
```

Truncation example program (output)

```
$ gcc truncate.c
$ ./a.out
s=-129, c=127
1111111101111111
        01111111
```

Explanation:

- `short` is a 16 bit signed type, `char`[1] is a signed 8 bit type
- After truncation from 16 to 8 bits, the sign bit was 0, so the resulting value became positive
- Look at the bit representations — convince yourself the values output by `printf` make sense!

[1] Compiler-dependent, tested with gcc 7.4.0 on x86-64 Linux

Conversions between signed and unsigned

- Another important type of conversion is between signed and unsigned values
- Fundamentally, data in the computer's memory has *no inherent meaning*
- It is up to the *program* to decide how to interpret data
- Conversions between signed and unsigned (without changing the number of bits) *do not change the underlying representation as bits*

Signed/unsigned conversion examples

Example: bit pattern 10010110 as signed and unsigned 8 bit integer values

Signed: -128 + 16 + 4 + 2 = -106

Unsigned: 128 + 16 + 4 + 2 = 150

Signed/unsigned conversion example program

```
#include <stdio.h>
unsigned char parsebits(const char *s) {
  unsigned char val = 0;
  char c;
  while ((c = *s++)) {
    val <<= 1;
    if (c == '1') { val |= 1; }
  }
  return val;
}

int main(void) {
  unsigned char uc = parsebits("10010110");
  char c = (char) uc;     // <-- conversion from unsigned to signed
  printf("%u %d\n", uc, c);
  return 0;
}
```

Signed/unsigned conversion example program (output)

```
$ gcc convert.c
$ ./a.out
150 -106
```

Considerations for writing programs

Programming considerations

- Semantics of integer values and data types can be surprisingly subtle
- C and C++ further complicate matters in several ways:
  - Data type sizes vary
  - Integer representation not actually specified by the language!
  - Some operations the program could perform have semantics that are implementation-defined or (worse) *undefined*
- Recommendation: be very careful!

Implicit conversions

- In C, there are many contexts in which *implicit conversions* will occur
  - Including ones where information can be lost!
- It's important to know where implicit conversions happen and to understand their effects
- It's not a bad idea to use explicit type casts so that conversions are *explicit*, even if they aren't strictly necessary
  - Semantics of program are more obvious, avoid unintended behaviors

Sign extension

- Sign extension can sometimes have surprising consequences (bits that you thought would be 0 become 1)
- Values belonging to unsigned types (`unsigned char`, `unsigned short`, etc.) are never sign extended