II. Representing and Manipulating Information

§2.1 Information Storage

- **Bytes**: blocks of 8 bits, the smallest addressable unit of memory
- **Virtual memory**: machine-level program views memory as a very large array of bytes
- **Virtual address space**: set of all possible addresses
  - conceptual image presented to the machine-level program
  - actual implementation=DRAM + flash memory + disk storage + special hardware + OS

¶2.1.1 Hexadecimal Notation

- 1 byte = 8 bits
  - 00000000_2 ~ 11111111_2
  - 0_{10} ~ 255_{10}
  - Base-16 or ***hexadecimal*** is commonly used for convenience
    - 00_{16} ~ FF_{16}
    - In C, numeric constants starting with **0x** or **0X** are interpreted as being in hexadecimal
    - characters 'A' ~ 'F' may be written in either upper or lowercase

¶2.1.2 Data Sizes

- **Word size**: nominal size of pointer data
  - virtual address is encoded by words
  - determines the **maximum size of the virtual address space**
  - machine with a w-bit word size gives the program access to at most 2^w bytes
- In recent years, widespread shift from 32-bit machines to ***64-bit machines***
  - 32-bit: virtual address space ≈ 4GB (4×10^9 bytes)
  - 64-bit: virtual address space ≈16EB (1.84×10^{19} bytes)
  - Most 64-bit machines can also run programs compiled for use on 32-bit machines (**backward compatibility**)
  - 64-bit programs will only run on a 64-bit machine

```
linux> gcc -m32 prog.c // runs on either a 32-bit or a 64-bit machine

linux> gcc -m64 prog.c // runs only on a 64-bit machine
```

- C language supports multiple data formats for both integer and floating point data
  - exact numbers of bytes for some data types depends on how the program is compiled
  - most of the data types **encode signed values**, unless prefixed by the keyword *unsigned*
  - ***char***: C standarad does not gurantee it to be encoded as signed
    - use ***signed char*** to guarantee a 1-byte signed value
  - **pointer**: uses the full word size of the program

C Declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char*		4	8
float		4	4
double		8	8

```
- For portability, make the program insensitive to the exact sizes of the different data types
```

H5 ¶2.1.3 Addressing and Byte Ordering

- For program objects that span mutliple bytes, need to establish two conventions:
  1. what the address of the object will be
  2. how we will order the bytes in memory
     - Two common conventions
       1. Little endian
          - store the object in memory ordered from least significant byte to most
          - Most Intel-compatible machines
       2. Big endian
          - store the object in memory ordered from most significant byte to least
          - IBM, Oracle (Sun Microsystems)
       - E.g. variable x of type *int* at address **0x100** has a hexadecimal value of **0x1234567**
         - Big endian

0x100	0x101	0x102	0x103
01	23	45	67

         - Little endian

0x100	0x101	0x102	0x103
67	45	23	01

     - recent microprocessor chips are ***bi-endian*** (configurable)
     - no technological reason to choose one byte ordering convention over the other
- Byte ordering becomes an **issue** when...
  1. binary data are communicated over a **network** between different machines
     - make sure sending machine converts its internal representation to the network standard, while the receiving machine converts the network standard to its internal representation
  2. looking at the **byte sequences** representing integer data
     - insepecting machine-level programs

     ```
     4004d3:    01 05 43 0b 20 00      add %eax, 0x200b45(%rip)
     ```

     - line generated by a ***disassembler***
       - : tool that determines the instruction sequence represented by an executable program file
       - adds a word of data to the value stored at an address computed by adding

0x200b43 to the current value of the ***program counter***, the address of the next instruction to be executed

3. programs are written that circumvent the normal type system
    - using ***cast*** or a ***union*** to allow an object to be referenced according to a different data type from which it was created

```c
#include <stdio.h>

typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, size_t len) {
  int i;
  for(i = 0; i < len; i++)
    printf(" %.2x", start[i]);
  printf("\n");
}

void show_int(int x) {
  show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x) {
  show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x) {
  show_bytes((byte_pointer) &x, sizeof(void *));
}
```

    - in the code above, functions pass *show_bytes* a pointer &x to their argument x, casting the pointer to be of type **unsigned char ***
        - indicates to the compiler that the program should consider the pointer to be a sequence of bytes rather than to an object of the original data type

H5 ¶2.1.5 Representing Code

- **instruction codings are different**
    - different machine types use different and incompatible instructions and encodings
    - even identical processors running different OS have differences in their coding conventions. --> not binary compatible
- Fundamental concpet of computer system: **program**, from the perspective of the machine, **is simply a sequence of bytes**

¶2.1.6 Introduction to Boolean Algebra

- **George Boole** observed that by eoncoding logic values **TRUE** and **FALSE** as binary values 1 and 0, he could formulate an algebra that captures the basic principles of logical reasoning
- Operations of Boolean algebra

```
~           &   0  1       |   0  1       ^   0  1
-------     --------       --------       --------
0     1     0   0  0       0   0  1       0   0  1
1     0     1   0  1       1   1  1       1   1  0
```

  - ~: logical operation **NOT**
    - ~TRUE = FALSE / ~FALSE = TRUE
  - **&**: logical operation **AND**
    - p & q == 1 only when p =1 and q = 1
  - **|**: logical operation **OR**
    - p | q == 1 when either p = 1 or q = 1
  - **^**: logical operation **EXCLUSIVE-OR**
    - p ^ q == 1 when either p = 1 and q = 0 or p = 0 and q = 1
- Boolean algebra still plays a central role in the design and analysis of digital systenedms
- Boolean algebra can be extended on ***bit vectors***

```
  0110      0110      0110
& 1100    | 1100    ^ 1100      ~ 1100
  ----      ----      ----        ----
  0100      1110      1010        0011
```

¶2.1.7 Bit-Level Operations in C

- C supports **bitwise Boolean operations** applied to any "integral data type"

C expression	Binary expression	Binary result	Hexadecimal result
~0x41	~[0100 0001]	[1011 1110]	0xBE
~0x00	~[0000 0000]	[1111 1111]	0xFF
0x69 & 0x55	[0110 1001] & [0101 0101]	[0100 0001]	0x41
0x69 | 0x55	[0110 1001] & [0101 0101]	[0111 1101]	0x7D

- commonly used to implement ***masking*** operations
  - **mask**: bit pattern that indicates a selected set of bits within a word
    - e.g. **0xFF**: the lower-order byte of a word

      => **x & 0xFF** returns the least significant byte of x with all other bytes set to 0
      - if **x =** **0x89ABCDEF** , **x & 0xFF** = **0x000000EF**

¶2.1.8 Logical Operations in C

- C provides a set of ***logical* operators ||, &&, and !**
  - correspond to the OR, AND, and NOT operations
- Logical operations treat any **nonzero argument** as **TRUE** and argument **0** as **FALSE**, then return either **1 (TRUE)** or **0 (FALSE)**

¶2.1.9 Shift Operations in C

- C provides a set of ***shift*** operations for **shifting bit patterns** to the left and to the right
  - if **x = [**$x_{w-1}, x_{w-2}, \ldots, x_0$**]**, **x << k** yields **[**$x_{w-1-k}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0$**]**
    - x is shifted **k** bits to the left, dropping off the **k** most significant bits and filling the right end with **k** zeros
  - Shift operations associate from left to right
    - **x << j << k** is equivalent to **(x << j) << k**
- Note that machines support **two forms of right shift** [**x >> k**]
  - **Logical**: fills the left end with **k** zeros
    - [$0, \ldots, 0, x_{w-1}, x_{w-2}, \ldots, x_k$]
  - **Arithmetic**: fills the left end with **k** repetitions of the most significant bit
    - [$x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_k$]
    - useful for operating on signed integer data

- C standards do not precisely define which type of right shift should be used with signed numbers -- either arithmetic or logical shifts may be used
  - Almost all use **arithmetic right shifts for signed data**
  - **logical right shifts for unsigned data**

§2.2 Integer Representations

- two different ways bits can be used to encode integers
  1. only representing nonnegative numbers
  2. representing negative, zero, and positive numbers
- strongly related both in their mathematical properties and their machine-level implementations

¶2.2.1 Integral Data Types

- C supports a variety of ***integral*** data types -- ones that represent finite ranges of **integers**
  - each type can specify a size with keywords, **char, short, long**
  - can indicate whether the represented numbers are all **nonnegative** (declared as ***unsigned***) or **negative** (by **default**)
- different sizes allow different ranges of values to be represented
- ***long*** is the only **machine-depenedent range** indicator
- **Note that** ranges are **not symmetric**
  - range of negative numbers extends one further than the range of positive numbers

¶2.2.2 Unsigned Encodings

- consider an integer data type of **w** bits and write a bit vector as $\vec{x}$

  For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:

  $$B2U_w(\vec{x}) := \sum_{i=0}^{w-1} x_i 2^i$$

  where $B2U_w$ is a function that interprets **binary to unsigned**

for example,

$$B2U_4([0001]) = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1$$

$$B2U_4([0101]) = 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5$$

$$B2U_4([1011]) = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 0 + 2 + 1 = 11$$

$$B2U_4([1111]) = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15$$

- Function $B2U_w$ is a bijection
- $UMax_w := \sum_{i=0}^{w-1} 2^i = 2^w - 1$

H5 ¶2.2.3 Two's-Complement Encodings

- to represent negative values, ***Two's complement form*** uses the **most significant bit** of the word to have **negative weight**

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:

$$B2T_w(\vec{x}) := -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

where $B2T_w$ is a function that interprets **binary to two's complement**

for example,

$$B2T_4([0001] = -0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1)$$

$$B2T_4([0101] = -0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5)$$

$$B2T_4([1011] = -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 0 + 2 + 1 = -5)$$

$$B2T_4([1111] = -1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 4 + 2 + 1 = -1)$$

- Function $B2T_w$ is a bijection
- $TMin_w := -2^{w-1}$
- $TMax_w : \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$

- Note that...
  1. **Two's complement range is asymmetric**: $|TMin| = |TMax| + 1$
     - half the bit patterns represent negative numbers, while half represent nonnegative numbers (including 0)
  2. **Maximum unsigned value is just over twice the maximum two's complement value**: $UMax = 2TMax + 1$
  3. **-1 has the same bit representation as** $UMax$

¶2.2.4 Conversions between Signed and Unsigned

- C allows casting between different numeric data types
  - suppose **x** is declared as **int** and **u** as **unsigned**
    - **(unsigned) x** : converts x to an unsigned value
    - **(int) u:** converts u to signed integer
  - Conversions are done based on a **bit-level perspective**
    - with the same word size, the n **umeric values might change** , but the **bit patterns stay the same**

given an integer x in the range $0 \leq x < UMax_w$, the function $U2B_w(x)$ gives the unique w-bit unsigned representation of x.

Similarly, when x is in the range $TMin_w \leq x \leq TMax_w$, the function $T2B_w(x)$ gives the unique w-bit two's-complement representation of x.

Now, define the function $T2U_w(x) := B2U_w(T2B_w(x))$.

this function takes a number between $TMin_w$ and $TMax_w$ and yields a number between 0 and $UMax_w$

the two numbers have identical bit representations, except that the argument has a two's-complement representation while the result is unsigned.

Similar for the function $U2T_w(x) := B2T_w(U2B_w(x))$

Hence, **conversion** from **two's-complement to unsigned**

For x such that $TMin_w \leq x \leq TMax_w$:

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases}$$

For example, $T2U_{16}(-12,345) = -12,345 + 2^{16} = 53,191$

$$T2U_w(-1) = -1 + 2^w = UMax_w$$

Similarly, **conversion** from **unsigned to two's-complement**

For u such that $0 \leq u \leq UMax_w$:

$$U2T_w(u) = \begin{cases} u, & u \leq TMax_w \\ u - 2^w, & u > TMax_w \end{cases}$$

H5 ¶2.2.5 Signed vs. Unsigned in C

- almost all machines use two's-complement
  - most numbers are **signed by default**
  - needs to add 'U' or 'u' as a suffix to create unsigned constants
    - 12345U or 0x1A2Bu
- ***explicit casting***

```c
int tx, ty;
unsigned ux, uy;

tx = (int) ux;
uy = (unsigned) ty;
```

- ***implicit casting***

```c
int tx, ty;
unsigned ux, uy;

tx = ux;    // cast to signed
uy = ty;    // cast to unsigned
```

- **using printf**

```c
int x = -1;
unsigned u = 2147483648; // 2^31

printf("x = %u = %d\n", x, x);
printf("u = %u = %d\n", u, u);

/* On 32-bit machine, it will print
x = 4294967295 = -1
y = 2147483648 = -2147483648 /*
```

- When an operation is performed where **one operand is signed and the other is unsigned**,
  - **C implicitly casts the signed argument to unsigned** and performs the operations assuming the numbers are **nonnegative**
  - quite accurate for standard arithmetic operations, but..

- **weird results** for relational operators **<** and **>**
  - e.g. $-1 < 0U$ returns **False**
    - because C casts -1 to 4294967295U

¶2.2.6 Expanding the Bit Representation of a Number

- Conversion between integers **having different word sizes** while retaining **the same numeric value**
  - **may not be possible** when the **destination data type is too small** to represent the desired value
  - **smaller to larger data type** should **always be possible**

1. **Zero extension**: for converting an unsigned number to a larger data type
   - **add leading zeros**

   Define bit vectors $\vec{u} = [u_{w-1}, u_{w-2}, \ldots, u_0]$ of width w and $\vec{u}' = [0, \ldots, 0, u_{w-1}, u_{w-2}, \ldots, u_0]$ of width w', where $w' > w$. Then $B2T_w(\vec{x}) = B2T_{w'}(\vec{x}')$.
2. **Sign extension**: for converting a two's-complement number to a larger data type
   - **add copies of the most significant bit**

   Define bit vectors $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ of width w and $\vec{x}' = [x_{w-1}, x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0]$ of width w', where $w' > w$. Then $B2T_w(\vec{x}) = B2T_{w'}(\vec{x}')$.

```
// When run as a 32-bit program on a big-endian machine that uses a two's complement representation,

sx = -12345:     cf c7
usx = 53191:     cf c7
x = -12345:      ff ff cf c7    // ff ff = 1111..1111
ux = 53191:      00 00 cf c7
```

¶2.2.7 Truncating Numbers

- Used to reduce the number of bits representing a number

```
int x = 53191;
short sx = (short) x;      // -12345
int y = sx;                // -12345
```

- casting x to be short will truncate a 32-bit int to a 16-bit short
- When truncating a w-bit number $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ to a k-bit number, **drop the high-order $w-k$ bits**
  - can alter its value -- ***<u>OVERFLOW!</u>***

- **Truncation of an <u>unsigned number</u>**

  Let $\vec{x}$ be the bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and let $\vec{x}'$ be the result of truncating it to k bits: $\vec{x}' = [x_{k-1}, x_{k-2}, \ldots, x_0]$.

  Let $x = B2U_w(\vec{x})$ and $x' = B2U_k(\vec{x}')$. Then $x' = x mod 2^k$
  - all of the bits that were truncated have weights of the form 2^i , where $i \geq k$

- **Truncation of a <u>two's-complement number</u>**

  Let $\vec{x}$ be the bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and let $\vec{x}'$ be the result of truncating it to k bits: $\vec{x}' = [x_{k-1}, x_{k-2}, \ldots, x_0]$.

  Let $x = B2T_w(\vec{x})$ and $x' = B2T_k(\vec{x}')$. Then $x' = U2T_k(x mod 2^k)$
  - Applying function $U2T_k$ will have the effect of converting the most significant bit x_{k-1} from having weight 2^{k-1} to having weight -2^{k-1}
  - For example,

    Converting $x = 53,191$ from int to $short$.

    Since $2^{16} = 65,536 \geq x$, we have $x\ mod 2^{16} = x$.

    But since we need to convert it to a 16-bit two's complement number, we get $x' = 53,191 - 65,536 = -12,345$

H5 ¶2.2.8 Advice on Signed versus Unsigned

- Implicit casting of signed to unsigned leads to some non-intuitive behavior
  - program bugs & difficult to identify it
- To avoid errors or vulnerabilities...
  1. **NEVER use unsigned numbers**
     - few languages other than C support unsigned integers
       - other language designers viewed unsigned integers as more trouble than

they are worth

2. or use it for **collections of bits with no numeric interpretation**
   - ***flags*** describing various Boolean conditions
   - implementing mathematical packages for modular arithmetic & multiprecision arithmetic

§2.3 Integer Arithmetic

- adding two positive numbers can yield a negative result
- x-y can yield something other than x-y < 0
- **NEED to understand Computer arithmetic** to write more reliable codes

¶2.3.1 Unsigned Addition

Consider two nonnegative integers x and y, such that $0 \leq x, y < 2^w$.

Each of these values can be represented by a w-bit unsigned number.

However, if we compute their sum, $0 \leq x + y \leq 2^{w+1} - 2$.

Representing this sum could require $w + 1$ bits

- **"Word size inflation"** : cannot place any bound on the word size required to fully represent the results of arithmetic operations

Now, let $+^u_w$ for arguments x and y, where $0 \leq x, y < 2^w$, be the result of truncating the integer sum $x + y$ to be w bits long and viewing the result as an unsigned number

- form of modular arithmetic: computing the sum modulo 2^w by discarding

  any bits with weight greater than 2^{w-1}

Then, ***Unsigned Addition*** can be formularized as...

For x and y such that $0 \leq x, y < 2^w$:

$$x +^u_w y = \begin{cases} x + y, & x + y < 2^w \quad\quad Normal \\ x + y - 2^w, & 2^w \leq x + y < 2^{w+1} \quad\quad Overflow \end{cases}$$

In addition, to **detect** **overflow** of unsigned additions,

For x and y in the range $0 \le x, y \le UMax_w$, let $s := x +_w^u y$.

Then the computation of s overflowed if and only if $s < x$ (or equivalently, $s < y$)

E.g. $9 +_4^u 12 = 5$. ==> **OVERFLOW!** ($\because 5 < 9$)

$(1001_2 + 1100_2 = 10101_2 => 0101_2)$ ($\because$ $word\ size = 4$)

Similarly, for ***Unsigned negation***,

For any number x such that $0 \le x < 2^w$, its w-bit unsigned negation $-_w^u x$ is given by the following:

$$-_w^u x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases}$$

E.g. $-_4^u 4 = 12$

($-0100_2 = 1100_2 = (-1) \cdot 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = -8 + 4 + 0 + 0 = -4$)

But since we are taking unsigned negation, $1100_2 = 8 + 4 + 0 + 0 = 12$

H5 ¶2.3.2 Two's-Complement Addition

- for two's complement addition, results can be either **too large** (positive) or **too small** (negative) to represent

Let us define $x +_w^t y$ be the result of **truncating** the integer sum $x + y$ to be w bits long.

For integer values x and y in the range $-2^{w-1} \le x, y \le 2^{w-1} - 1$:

$$x +_w^t = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y & Positive\ overflow \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} & Normal \\ x + y + 2^w, & x + y < -2^{w-1} & Negative\ overflow \end{cases}$$

- when $x + y$ exceeds $TMax_w$, => **positive overflow**!

  e.g. $x = 5 = [0101], y = [0101] => x + y = 10 = [01010], \ x +_4^t y = -6 = [1010]$

  $10 - 2^4 = -6$

- when $x - y$ is less than $TMin_w$ => **negative overflow**!

  e.g. $x = -8 = [1000]$, $y = -5 = [1011]$ => $x + y = -13 = [10011]$, $\ x +_4^t y = 3 = [0011]$

  $-13 + 2^4 = 3$

To **detect overflow** in two's-complement addition

For x and y in the range $TMin_w \le x, y \le TMax_w$, let $s := x +_w^t y$.

Then the computation of s has had positive overflow if and only if $x > 0$ and $y > 0$ but $s \le 0$.

The computation has had negative overflow if and only if $x < 0$ and $y < 0$ but $s \ge 0$.

¶2.3.3 Two's-Complement Negation

- Every number x in the range $TMin_w \le x \le TMax_w$ has an additive inverse under $+_w^t$, which we denote $-_w^t x$ as follows:

For x in the range $TMin_w \le x \le TMax_w$, its two's-complement negation $-_w^t x$ is given by the formula

$$-_w^t x = \begin{cases} TMin_w, & x = TMin_w \\ -x, & x > TMin_w \end{cases}$$

Note that $TMin_w + TMin_w = -2^{w-1} + -2^{w-1} = -2^w$ ==> **Negative overflow!**

- Bit-level representation can be used to find two's-complement negation

  examples with a 4-bit word size:

$\vec{x}$	~$\vec{x}$	incr(~ $\vec{x}$)
[0101]=5	[1010]=-6	[1011]=-5
[0111]=7	[1000]=-8	[1001]=-7
[0000]=0	[1111]=-1	[0000]=0
[1000]=-8	[0111]=7	[1000]=-8

¶2.3.4 Unsigned Multiplication

Integers x and y in the range $0 \leq x, y \leq 2^w - 1$ can be represented as w-bit unsigned numbers, but their product $x \cdot y$ can range between 0 and $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$, requiring $2w$ bits to represent

Let's define $x *_w^u y$ be the w-bit value given by the low-order w bits of the $2w$-bit integer product.

Then, **Unsigned multiplication** can be formularized as:

For x and y such that $0 \leq x, y \leq UMax_w$:

$$x *_w^u y = (x \cdot y)\ mod\ 2^w$$

¶2.3.5 Two's-Complement Multiplication

Integers x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$ can be represented as w-bit two's complement numbers, but their product $x \cdot y$ can range between $-2^{w-1} \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$ and $-2^{w-1} \cdot -2^{w-1} = 2^{2w-2}$, requiring as many as $2w$ bits to represent in two's complement form

Let's define $x *_w^t y$ be the w bit result after trancating the $2w$-bit product.

Then, **Two's-complement multiplication** can be formularized as:

For x and y such that $TMin_w \leq x, y \leq TMax_w$:

$$x *^t = U2T_w((x \cdot y)\ mod\ 2^w)$$

- However, note that the **bit-level representation** of the product operation is **identical** for both unsigned and two's-complement multiplication

¶2.3.6 Multiplying by Constants

- Historically, the **integer multiply instructions** on many machines was fairly slow
  - Other integer operations (+, -, bitwise, shifting) require ≈ 1 clock cycle, while multiplication takes ≈ 3 clock cycles even on the Intel Core i7 Haswell.
- To **optimize** multiplications, compilers replace multiplications by **constant factors** with **shift & addition** operations
  - e.g. $x * 14$ can be rewritten as $(x << 3) + (x << 2) + (x << 1)$ since $14 = 2^3 + 2^2 + 2^1$

or, $(x << 4) - (x << 1)$ $\ (\because 14 = 2^4 - 2^1)$

Multiplication by a power of 2

Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$.

Then for any $k \geq 0$, the $w + k$-bit unsigned representation of $x2^k$ is given by $[x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]$, where k zeros have been added to the right

When shifting left by k for a fixed word size, the high-order k bits are discarded, yielding

$$[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$$

Unsigned multiplication by a power of 2

For C variables x and k with unsigned values x and k, such that $0 \leq k < w$, the C expression $x << k$ yields the value $x *^u_w 2^k$

Two's-complement multiplication by a power of 2

For C variables x and k with two's-complement value x and unsigned value k, such that $0 \leq k < w$, the C expression $x << k$ yields the value $x *^t_w 2^k$

¶2.3.7 Dividing by Powers of 2

- **Integer division** is even slower than integer multiplication --> ≈30 or more clock cycels
  - Use **Right shifts!**
    - logical right shifts -- unsigned
    - arithmetic right shifts -- two's-complement
- Integer division always **rounds toward zero**

Unsigned division by a power of 2

For C variables x and k with unsigned values x and k, such that $0 \leq k < w$, the C expression $x >> k$ yields the value $\lfloor x/2^k \rfloor$.

-- Note that for unsigned divisions, use **Logical Right Shifts!**

Two's-complement division by a power of 2, rounding down

For C variables x and k have two's-complement value x and unsigned value k, respectively, such that $0 \leq k < w$. The C expression $x >> k$, when the shift is performed **arithmetically**, yields the value $\lfloor x/2^k \rfloor$

Two's complement division by a power of 2, rounding up

Let C variables x and k have two's-complement value x and unsigned value k, respectively, wuch that $0 \leq k < w$. The C expression $(x + (1 << k) - 1) >> k$, when the shift is performed **arithmetically**, yields the value $\lceil x/2^k \rceil$

$(\because \lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor)$