CS 24000 - Programming In C

Week Nine: File pointers , data format, floating point numbers

Zhiyuan Li
Department of Computer Science
Purdue University, USA

- We have been using *stdin*, *stdout*, and *stderr* for input and output
  - As we write more sophisticated programs, we need more versatile ways to
    - Import information from multiple files
    - Export information to multiple files
  - The C languages provide ways to
    - “Open” many different files for read and write files
    - These are done by making file system calls
- So far, we also have mainly dealt with text files
  - We will now discuss I/O for binary data
    - “dump” binary data w/o converting to text

File Systems

- A file may be defined as a physical entity that stores information
  - The file system is a part of the operating system
  - It specifies how files are organized
    - for retrieval and modification.
- When a file system is organized in a hierarchy (instead of being flat), we have
  - Directories (or folders)
    - Each directory may contain other directories (i.e. subdirectories)
  - Files (contained in directories)
- UNIX takes an extended view of files:
  - Peripheral devices (keyboard, screen, etc)
  - Pipes (inter process communication)
  - Sockets (communication via computer networks)

- The OS provides a number of file system calls for
  - Creating a file
  - Opening an existing file for read, write, append
  - Closing a file
  - Maintaining the open count for each file
    - How many activated programs have opened a specific file
  - Moving the next read/write position within a file
  - Setting access privilege for each file
  - Providing system level buffering
  - Etc

Files in C

- In Unix, a C program can directly make file system calls, but the C file library routines make it easier in many situations
  - Higher level operations than reading bytes
  - User-level buffering
  - Automatic data format transformation

- File abstraction by using the `FILE` type:
    - **`FILE *fp` // *fp is a pointer to a file .**

- To open a file, call
  FILE* fopen(const char* filename, const char* mode)
  - mode can be “r” (read), “w” (write), “a” (append)
  - Returns a file pointer
    - returns NULL on error (e.g., improper permissions)
  - filename is a string that holds the name of the file on disk
  - Automatically create a new file for write if not existing yet

- We will run a few examples to open files for
  - Write (to a new file)
  - Read
  - Re-write an existing file
  - Append to an existing file
  - Try to write a read-only file

Reading formatted text files

- `fscanf` requires a `FILE*` for the file to be read

  fscanf(ifp, “<format string>”, inputs)
- Returns the number of values read or EOF on an end of file

- Example: Suppose in.list contains

```
foo 70
bar 50
```

- To read elements from this file, we might write

  fscanf(ifp, “%s %d”, name, count)

- We can check against EOF:

  while(fscanf(ifp,“%s %d”,name,count)!=EOF);

Testing EOF

- Ill-formed input may confuse comparison with EOF

`fscanf` returns the number of successful matched items

```
while(fscanf(ifp,“%s  %d”, name,count)==2)
```

- We can use `feof`:

```
while (!feof(ifp))  {
   if (fscanf(ifp,“%s  %d”,name,count)!=2) break;
   fprintf(ofp, format, control);
}
```

Closing files

- **fclose(ifp);**
- Why do we need to close a file?
  - File systems typically buffer output
  - The buffer is flushed when the file is closed, or when full
    - This is called write-back, for efficiency
  - If the program aborts or terminates before the file is closed or explicitly flushed, then the buffered output might not be written back completely or at all

File pointers

- Three special file pointers:
  - **`stdin`** (standard input) /*corresponding to fd 0*/
  - **`stdout`** (standard output) /*corresponding to fd 1*/
  - **`stderr`** (standard error) /*corresponding to fd 2*/

Other file operations

- Remove file from the file system:

```
int remove (const char * filename)
```

- Rename file

```
int rename (const char * oldname,
                 const char * newname)
```

Binary file i/o at C level

- In Project 2, we will read and write binary data
- Often we can use fread() and fwrite() for such purposes
- Students should read the textbook and man page on Unix systems for their definitions
  - And do exercises for such uses
- The following is a brief summary

Binary I/O

- Read at most **`nobj`** items of size **`sz`** from **`stream`** into **`p`**
- **`feof`** and **`ferror`** used to test end of file

```
size_t fread(void* p,size_t sz,size_t nobj,FILE* stream)
```

- Write at most **`nobj`** items of size **`sz`** from **`p`** onto **`stream`**

```
size_t fwrite(void*p,size_t sz,size_t nobj,FILE* stream)
```

File position

`int fseek(FILE* stream, long offset, int origin)`

  - Set file position in the stream. Subsequent reads and writes begin at this location
  - Origin can be **SEEK_SET,SEEK_CUR,SEEK_END** for binary files

- **To find out the current position within the stream**

  `long ftell(FILE * stream)`

- **To set the file to the beginning of the file**

  `void rewind(FILE * stream)`

- **see page 247-248 in the text**

Example

```
#include <stdio.h>
int main() {
 long fsize;
 FILE *f;

 f = fopen(“./log”, “r”);

 fseek(f, 0, SEEK_END) ;
 fsize =  ftell(f) ;
 printf(“file size is: %d\n”, fsize);

 fclose(f);
}
```

Temp files

- Create temporary file (removed when program terminates)

```
FILE * tmpfile (void)
```

- We show a couple of examples of the use of tmpfiles

Text Stream I/O Read

- Read next char from stream and return it as an unsigned char cast to an int, or EOF
- `int fgetc(FILE * stream)`
- Reads in at most size-1 chars from the stream and stores them into null-terminated buffer pointed s. Stop on EOF or error
- `char* fgets(char *s, int size, FILE  *stream)`
- Writes c as an unsigned char to stream and returns the char
- `int fputc (int c, FILE * stream)`
- Writes string s without null termination; returns a non-negative number on success, or EOF on error
- `int fputs(const char *s, FILE *stream)`

UNIX File System Calls

- File descriptor
  - A handle to access a file, like the file pointer in streams
  - Small non-negative integer used in same open/read-write/ close ops
  - Returned by the `open call`; all opens have distinct file descriptors
  - Once a file is **closed**, fd can be reused
  - Same file can be opened several times, with different fd's

Management functions

- `#include <fnctl.h>`
- `int open(const char *path, int flags);`
- `int open(char *path, int flags, mode_t mode);`
- `int creat(const char *pathname, mode_t mode);`
  - All the above return a function descripter
  - creat is equivalent to open with flags equal to O_CREAT|O_WRONLY|O_TRUNC.
- **Flags:** O_RDONLY, O_WRONLY or O_RDWR bitwise OR with O_CREAT, O_EXCL, O_TRUNC, O_APPEND, O_NONBLOCK, O_NDELAY
- **Mode:** the permissions to use in case a new file is created.
- `int close(int fd);`

- `#include <unistd.h>`
- `ssize_t read(int fd, void *buf, size_t cnt);`
- `ssize_t write(int fd, void *buf, size_t cnt);`

- fd is a descriptor, _not_ FILE pointer

- Returns number of bytes transferred, or -1 on error
- Normally waits until operation is enabled (e.g., there are bytes to read), except under O_NONBLOCK and O_NDELAY (in which case, returns immediately with ‘‘try again’’ error condition)

Example

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

int main() {
    char buf[100];

    int f1 = open("log1", O_RDONLY);
    int f2 = open("log2", O_RDONLY);
    fprintf(stderr, "Log1 file descriptor is: %d\n", f1);
    fprintf(stderr, "Log2 file descriptor is: %d\n", f2);
    close(f1); close(f2);

    f2 = open("log2", O_RDONLY);
    fprintf(stderr, "Notice the new file descriptor: %d\n", f2);
    close(f2);
}
```

The issue of data endians

- When we perform binary I/O, it is important to understand the endian issue
- We first dump some integer data and then examine the layout of the output
  - Run two programs (see next pages)
    - dumpint | od –t x1 compare with
    - printint

```
#include <stdio.h>        /* dumpint.c */
#include <unistd.h>
int main() {
int a[4];
a[0] =0x0000ffff; a[1]=0xffff0000; a[2]=0x00000001; a[3] =0x10000001;

    if (write(1, a, 4) < 1) fprintf(stderr, "failed to write a[0]\n");
    if (write(1, &a[1], 4) < 1) fprintf(stderr, "failed to write a[1]\n");
    if (write(1, &a[2], 4) < 1) fprintf(stderr, "failed to write a[2]\n");
    if (write(1, &a[3], 4) < 1) fprintf(stderr, "failed to write a[3]\n");
 return 0;
}
```

```
 % dumpint | od -t x1
0000000 ff ff 00 00 00 00 ff ff 01 00 00 00 01 00 00 10
0000020
```

```
#include <stdio.h>    /* printint.c */
int main() {
 int i=0x0000ffff, j=0xffff0000, k=0x00000001,
h=0x10000001;
    printf("%d%d%d%d",i,j,k,h);
 return 0;
}
```

```
%printint
65535-655361268435457
```

Next, compare to fwrite() result

```
#include <stdio.h>                    /* fwriteint.c */
#include <unistd.h>
int main() {
 int a[4]; FILE *fp;
 a[0] =0x0000ffff; a[1]=0xffff0000; a[2]=0x00000001;
 a[3] =0x10000001;
     if ((fp = fopen("./fwriteout", "w")) == NULL) {
             fprintf(stderr, "failed to open file\n");
              return(1);
       }
     if (fwrite(&a, 4, 1, fp) < 1) {fprintf(stderr, "failed to write a[0]\n"); return(1);}
     if (fwrite(&a[1], 4, 1, fp) < 1) {fprintf(stderr, "failed to write a[1]\n");
return(1);}
     if (fwrite(&a[2], 4, 1, fp) < 1) {fprintf(stderr, "failed to write a[2]\n");
return(1);}
     if (fwrite(&a[3], 4, 1, fp) < 1) {fprintf(stderr, "failed to write a[3]\n");
return(1);}
 return 0;
}
```

```
 od -t x1 fwriteout
0000000 ff ff 00 00 00 00 ff ff 01 00 00 00 01 00 00 10
0000020
```

Same as the write() result

Data transfer between machines of different data representation

From the above result, we see that if we port data from a little-endian machine to a big-endian machine, we must convert the endian before using the data as operands on the big-endian machine

However, on a machine of the same endian, we won’t have this problem

Let us run a program to read back the dumped data

```
#include <stdio.h>
#include <unistd.h>
int main() {
 int a[4]; FILE *fp;
     if ((fp = fopen("./fwriteout", "r")) == NULL) {
             fprintf(stderr, "failed to open file\n");
              return(1);
       }
     if (fread(&a, 4, 1, fp) < 1) {fprintf(stderr, "failed to read a[0]\n"); return(1);}
     if (fread(&a[1], 4, 1, fp) < 1) {fprintf(stderr, "failed to read a[1]\n"); return(1);}
     if (fread(&a[2], 4, 1, fp) < 1) {fprintf(stderr, "failed to read a[2]\n"); return(1);}
     if (fread(&a[3], 4, 1, fp) < 1) {fprintf(stderr, "failed to read a[3]\n"); return(1);}
     printf("%d%d%d%d",a[0],a[1],a[2],a[3]);
 return 0;
}
```

```
 % freadint
65535-655361268435457
```

This is exactly the numbers we dumped earlier

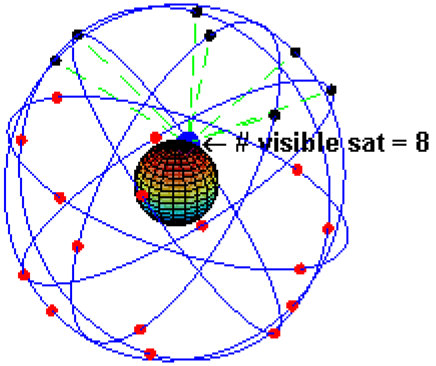
- Next we discuss floating point number representation, which will also be covered by Project 2

Floating point numbers

- Used extensively in scientific and engineering numerical computation & computer graphics (including game software)
- Graphics algorithms, e.g. ray tracing
- Global positioning systems (GPS)
- … …

RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing
Sven Woop, jörg Schmittler, Philipp Slusallek, ACM Transactions on Graphics (TOG)
 - Proceedings of ACM SIGGRAPH 2005

GPS algorithms



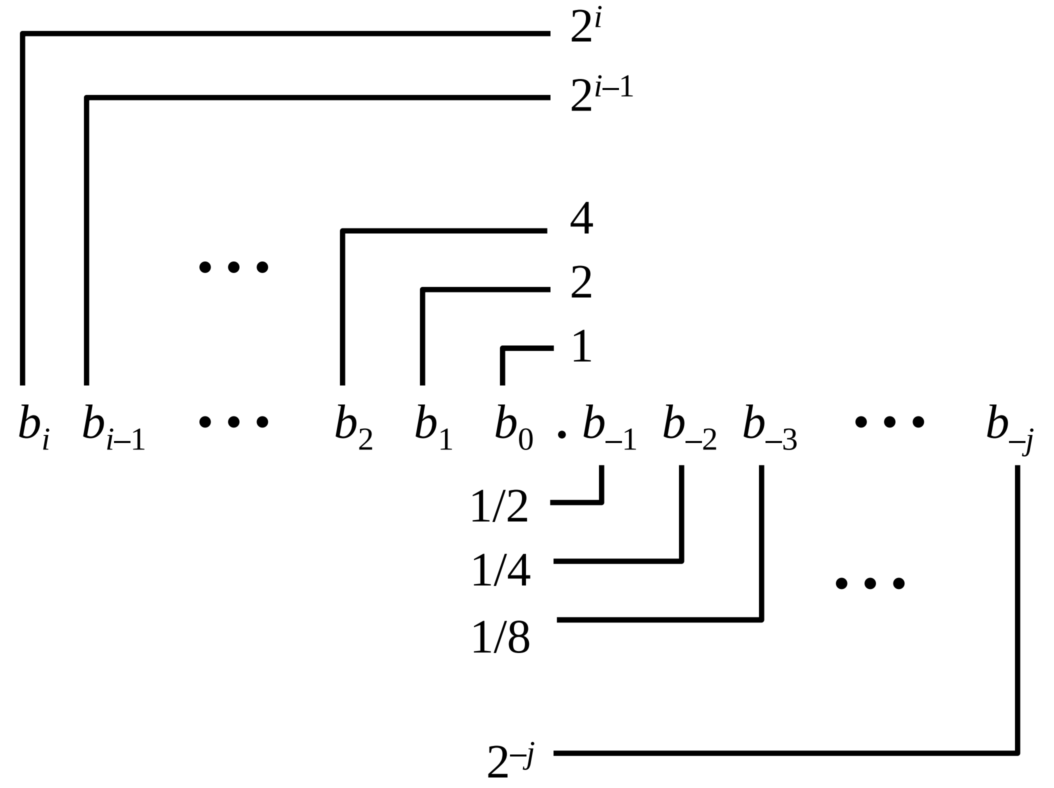
```
 * soe            Sun / Earth mass ratio
 * soem           Sun / (Earth + Moon) mass ratio
 * tropical_year    tropical year [day]
 * twopi          two pi
 * ut_to_st        conversion factor for UT to siderial time [UT s/sid s]

#define pi     ((double)3.14159265358979)
#define pio2    ((double)0.5*pi)
#define twopi   ((double)2.0*pi)
```

```
#define tropical_year   ((double)365.2421910)
#define ut_to_st       ((double)1.00273790934)
#define st_to_ut       ((double)0.9972695663399999)

#define emajor  ((double)6378137.0)
#define eflat   ((double)0.00335281068118 )
#define erate   ((double)7.292115855228083e-5)
#define soem    ((double)328900.550)
#define eom     ((double)81.3005870)
#define soe     (soem*((double)1.0 + (double)1.0/eom) )

#define L1_frequency    ((double)1575.420e+6)
#define L2_frequency    ((double)1227.600e+6)
#define L1_wavelength   ((double)cee/L1_frequency)
#define L2_wavelength   ((double)cee/L2_frequency)

#define ghadot  ((double) 7.292117855228083e-5 )
#define xmu     ((double) 3.986008e+14
```

IEEE Floating Point

- IEEE Standard 754
  - Estabilished in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs
- Driven by Numerical Concerns
  - Standards for rounding, overflow, underflow
  - Design principles: need both precisions and wide range

Fractional Binary Numbers



- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

Fractional Binary Number Examples

Value	Representation
5-3/4	101.11_2
2-7/8	10.111_2
63/64	0.111111_2

- Observation
  - Divide by 2 by shifting right
  - Numbers of form $0.111111\ldots_2$ just below 1.0
    - Use notation $1.0 - \varepsilon$
- Limitation
  - Can only exactly represent numbers of the form $x/2^k$
  - Other numbers have repeating bit representations

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

Floating Point Representation

s

exp

frac

- Numerical Form
  - $-1^{s}\ M\ 2^{E}$
    - Sign bit s determines whether number is negative or positive
    - Significand M normally a fractional value in range [1.0,2.0).
    - Exponent E weights value by power of two
- Encoding
  - MSB is sign bit
  - `exp` field encodes E
  - `frac` field encodes M
- Sizes
  - Single precision: 8 `exp` bits, 23 `frac` bits
    - 32 bits total
  - Double precision: 11 `exp` bits, 52 `frac` bits
    - 64 bits total

This is however not the end of the story

“Normalized” Numeric Values

- Under the condition
  - `exp` ≠ `000…0` and `exp` ≠ `111…1`
- Exponent coded as *biased* value

  $E = Exp - Bias$

  - *Exp* : unsigned value denoted by **`exp`**
  - *Bias* : Bias value
    - Single precision: 127 (*Exp*: from 1 to 254, *E*:from -126 to 127)
    - Double precision: 1023 (*Exp*: from 1 to 2046, *E*: from -1022 to 1023
    - in general: *Bias* = $2^{m-1} - 1$, where m is the number of exponent bits
- Significand coded with implied leading 1

  $m = 1.xxx…x_2$

  - **`xxx…x`**: bits of `frac`
  - Minimum when **`000…0`** ($M = 1.0$)
  - Maximum when **`111…1`** ($M = 2.0 - \varepsilon$)
  - Get extra leading bit for “free”

Normalized Encoding Example

- Value

  `Float F = 15213.0;`

  $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \text{ X } 2^{13}$

- Significand

  $M = 1.\underline{1101101101101}_2$

  $\mathtt{frac} = \underline{1101101101101}0000000000_2$

- Exponent

  $E = 13$

  $Bias = 127$

  $Exp = 140 = 10001100_2$

Floating Point Representation:

Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000
140:	100	0110	0					
15213:			*1*110	1101	1011	01		

Special (Denormalized) Values

- Under the condition
  - `exp` = `000…0`
- Value
  - Exponent value *E* = –*Bias* + 1
  - Significand value *m* = $0.xxx\ldots x_2$
    - **xxx…x:** bits of `frac`
- Cases
  - `exp` = `000…0`, `frac` = `000…0`
    - Represents value 0
    - Note that have distinct values +0 and –0
  - `exp` = `000…0`, `frac` ≠ `000…0`
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - “Gradual underflow”

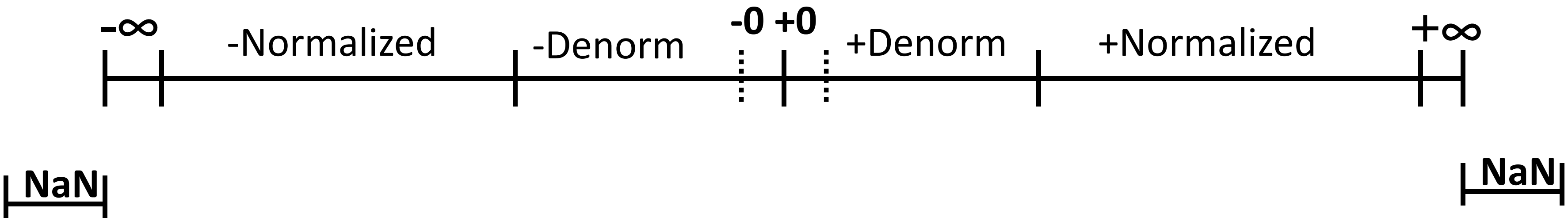
Interesting Numbers

Description	`exp`	`frac`	Numeric Value
Zero	00…00	00…00	0.0
Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}}$ X $2^{-\{126,1022\}}$
– Single ≈ 1.4 X 10^{-45}			
– Double ≈ 4.9 X 10^{-324}			
Largest Denormalized	00…00	11…11	(1.0 – ε) X $2^{-\{126,1022\}}$
– Single ≈ 1.18 X 10^{-38}			
– Double ≈ 2.2 X 10^{-308}			
Smallest Pos. Normalized	00…01	00…00	1.0 X $2^{-\{126,1022\}}$
– Just larger than largest denormalized			
One	01…11	00…00	1.0
Largest Normalized	11…10	11…11	(2.0 – ε) X $2^{\{127,1023\}}$
– Single ≈ 3.4 X 10^{38}			
– Double ≈ 1.8 X 10^{308}			

Special Values

- Condition
  - `exp` = `111…1`
- Cases
  - `exp` = `111…1`, `frac` = `000…0`
    - Represents value ∞ (infinity)
    - Operation that overflows
    - Both positive and negative
    - E.g., 1.0/0.0 = –1.0/–0.0 = +∞, 1.0/–0.0 = –∞
  - `exp` = `111…1`, `frac` ≠ `000…0`
    - Not-a-Number (NaN)
    - Represents case when no numeric value can be determined
    - E.g., sqrt(–1), ∞ – ∞

Summary of Floating Point Real Number Encodings



Floating Point Operations

- Conceptual View
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`
- Rounding Modes (illustrate with $ rounding)

	$1.40	$1.60	$1.50	$2.50	–$1.50
– Zero (truncate)	$1.00	$1.00	$1.00	$2.00	–$1.00
– Round down (-∞)	$1.00	$1.00	$1.00	$2.00	–$2.00
– Round up (+∞)	$2.00	$2.00	$2.00	$3.00	–$1.00
– Nearest Even (default)	$1.00	$2.00	$2.00	$2.00	–$2.00

Note:

1. Round down: rounded result is close to but no greater than true result.

2. Round up: rounded result is close to but no less than true result.

A Closer Look at Round-To-Even

- Default Rounding Mode
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated
- Applying to Other Decimal Places
  - When exactly halfway between two possible values
    - Round so that least signficant digit is even
  - E.g., round to nearest hundredth

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

Rounding Binary Numbers

- Binary Fractional Numbers
  - "Even" when least significant bit is `0`
  - Half way when bits to right of rounding position = $\texttt{100…}_2$
- Examples
  - Round to nearest 1/4 (2 bits right of binary point)

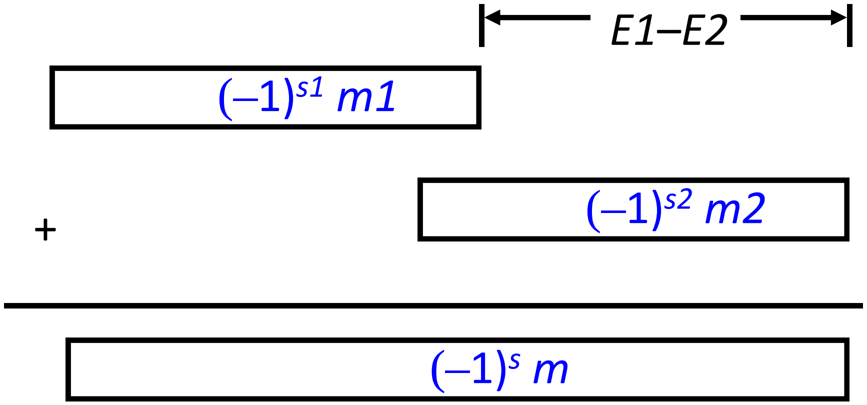
Value	Binary	Rounded	Action	Rounded Value
2-3/32	$\texttt{10.00011}_2$	$\texttt{10.00}_2$	(<1/2—down)	2
2-3/16	$\texttt{10.00110}_2$	$\texttt{10.01}_2$	(>1/2—up)	2-1/4
2-7/8	$\texttt{10.11100}_2$	$\texttt{11.00}_2$	(1/2—up)	3
2-5/8	$\texttt{10.10100}_2$	$\texttt{10.10}_2$	(1/2—down)	2-1/2

FP Multiplication

- Operands
  $(-1)^{s1}\ M1\ \ 2^{E1}$
  $(-1)^{s2}\ M2\ \ 2^{E2}$
- Exact Result
  $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s: $s1$ ^ $s2$
  - Significand M: $M1 * M2$
  - Exponent E: $E1 + E2$
- Fixing
  - If $M \geq 2$, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit `frac` precision
- Implementation
  - Biggest chore is multiplying significands

FP Addition

- Operands
  - $(-1)^{s1}\ M1\ \ 2^{E1}$
  - $(-1)^{s2}\ M2\ \ 2^{E2}$
  - – Assume $E1 > E2$
- Exact Result
  - $(-1)^{s}\ M\ \ 2^{E}$
  - – Sign s, significand M:
    - Result of signed align & add
  - – Exponent E: $E1$
- Fixing
  - – If $M \geq 2$, shift M right, increment E
  - – if $M < 1$, shift M left k positions, decrement E by k
  - – Overflow if E out of range
  - – Round M to fit `frac` precision



Mathematical Properties of FP Add

- Compare to those of Abelian Group
  - Closed under addition? YES
    - But may generate infinity or NaN
  - Commutative? YES
  - Associative? NO
    - Overflow and inexactness of rounding
  - 0 is additive identity? YES
  - Every element has additive inverse ALMOST
    - Except for infinities & NaNs
- Montonicity
  - $a \geq b \Rightarrow a+c \geq b+c$? ALMOST
    - Except for infinities & NaNs

Algebraic Properties of FP Mult

- Compare to Commutative Ring
  - Closed under multiplication? YES
    - But may generate infinity or NaN
  - Multiplication Commutative? YES
  - Multiplication is Associative? NO
    - Possibility of overflow, inexactness of rounding
  - 1 is multiplicative identity? YES
  - Multiplication distributes over addtion? NO
    - Possibility of overflow, inexactness of rounding
- Montonicity
  - $a \geq b \ \& \ c \geq \ 0 \Rightarrow a * c \geq b * c$? ALMOST
    - Except for infinities & NaNs

Floating Point in C

- C Supports Two Levels

`float`	single precision
`double`	double precision

- Conversions
  - Casting between `int`, `float`, and `double` changes numeric values
  - `Double` or `float` to `int`
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range
      - Generally saturates to TMin or TMax
  - `int` to `double`
    - Exact conversion, as long as int has ≤ 54 bit word size
  - `int` to `float`
    - Will round according to rounding mode

Answers to Floating Point Puzzles

```
int x = …;

float f = …;

double d = …;
```

Assume neither `d` nor `f` is NAN

`x == (int)(float) x`	**No: 24 bit significand**
`x == (int)(double) x`	**Yes: 53 bit significand**
`f == (float)(double) f`	**Yes: increases precision**
`d == (float) d`	**No: loses precision**
`f == -(-f);`	**Yes: Just change sign bit**
`2/3 == 2/3.0`	**No: 2/3 == 1**
`d < 0.0` ⇒ `((d*2) < 0.0)`	**Yes!**
`d > f` ⇒ `-f < -d`	**Yes!**
`d * d >= 0.0`	**Yes!**
`(d+f)-d == f`	**No: Not associative**

Quiz 6 #1

- With the following declarations:
  - Unsigned char a = ‘\xff’, b = ‘\x11’, c;

  After executing the statements:

  c = a ^ b;

  Variable c will hold the hexadecimal value

  - (a) GG
  - (b) EE
  - (c) 00
  - (d) 10
  - (e) 01

- Answer (b) EE
- Hint: 1111 1111
- 0001 0001 (^
- ---------------------
- 1110 1110
- Which is EE

Quiz 6 #2

- With the following declarations:
  - Unsigned char a = ‘\xff’, b = ‘\x11’, c;

  After executing the statements:

  c = a + b;

  Variable c will hold the hexadecimal value

  - (a) GG
  - (b) EE
  - (c) 00
  - (d) 10
  - (e) 01

- Answer (d) 10

- Method 1 – directly do binary add
  - After promoting both unsigned chars to int
    - 0000 0000 1111 1111
    - 0000 0000 0001 0001 (+
    - ------------------------

      0000 0001 0001 0000

    Written back to unsigned c, we have 0001 0000
- Method 2 – convert to decimal first, we have
    - 255 + 17 = 272 divided by 16 is 17 exact, which is hexadecimal "110", dropping the carry bit 1 we have "10" hexadecimal
- Method 1 is clearly more straightforward here.

Quiz 6 #3

- With the following declarations:
  - char a = ‘\xff’;
  - int x;

  After executing the statements:

  x = a << 1;

  Variable x will hold the hexadecimal value

  - (a) FF
  - (b) F0
  - (c) FF0
  - (d) 1FE
  - (e) FFFE

- Answer (e) FFFE for 16 bit int or FFFFFFFE for 32 bit int

- Hint: char is signed. When performing a << 1, a is first promoted to integer FFFF and the left shift result is FFFE, written back to int x.