CS 33

Data Representation (Part 4)

Normalized Encoding Example

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**

  M **=** `1.`$\underline{\texttt{1101101101101}}_2$

  `frac` **=** $\underline{\texttt{1101101101101}}\texttt{0000000000}_2$

- **Exponent**

  E **= 13**

  bias **= 127**

  exp **= 140 =** $\texttt{10001100}_2$

- **Result:**

`s`	`exp`	`frac`
`0`	`10001100`	`11011011011010000000000`

Denormalized Values

- **Condition: exp = `000…0`**
- **Exponent value: E = 1 – Bias (instead of E = 0 – Bias)**
- **Significand coded with implied leading 0: M = $0.xxx\ldots x_2$**
  - **`xxx…x`: bits of `frac`, range [0,1)**
- **Cases**
  - **`exp` = `000…0`, `frac` = `000…0`**
    - » **represents zero value**
    - » **note distinct values: +0 and –0 (why?)**
  - **`exp` = `000…0`, `frac` ≠ `000…0`**
    - » **numbers closest to 0.0**
    - » **for S.P., range from .111...1 * 2^{-126} to .000...001 * 2^{-126}**
    - » **smallest normalized value is 1.0 * 2^{-126}**

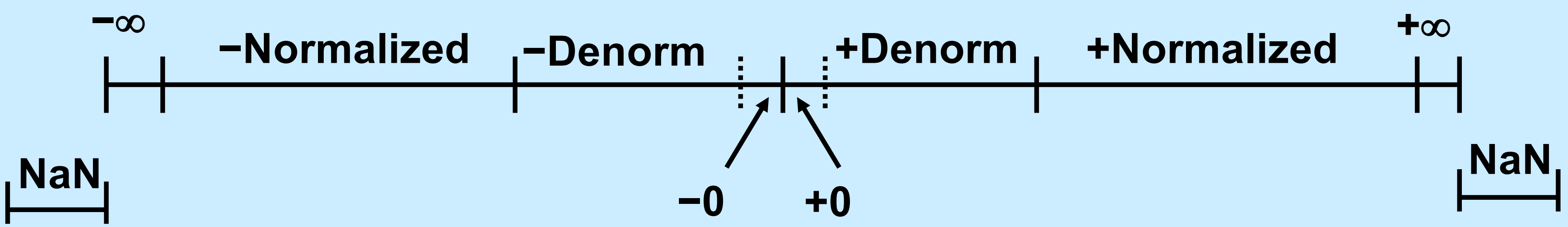
Special Values

- Condition: `exp` = 111…1

- Case: `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 = −∞

- Case: `exp` = 111…1, `frac` ≠ 000…0
  - not-a-number (NaN)
  - represents case when no numeric value can be determined
  - e.g., sqrt(–1), ∞ − ∞, ∞ × 0

Visualization: Floating-Point Encodings

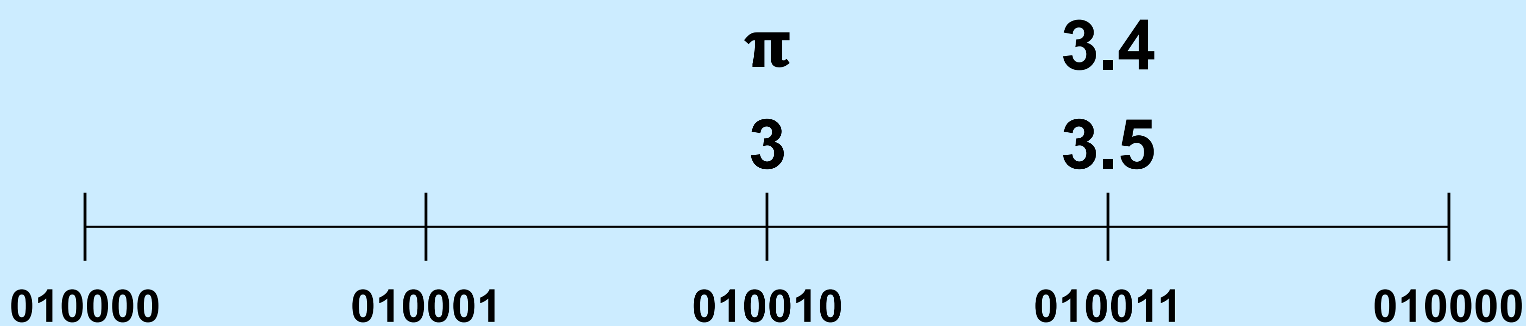


Mapping Real Numbers to Float

- The real number 3 is represented as
  0 100 10
- The real number 3.5 is represented as
  0 100 11
- How is the real number 3.4 represented?

  0 100 11
- How is the real number π represented?

  0 100 10

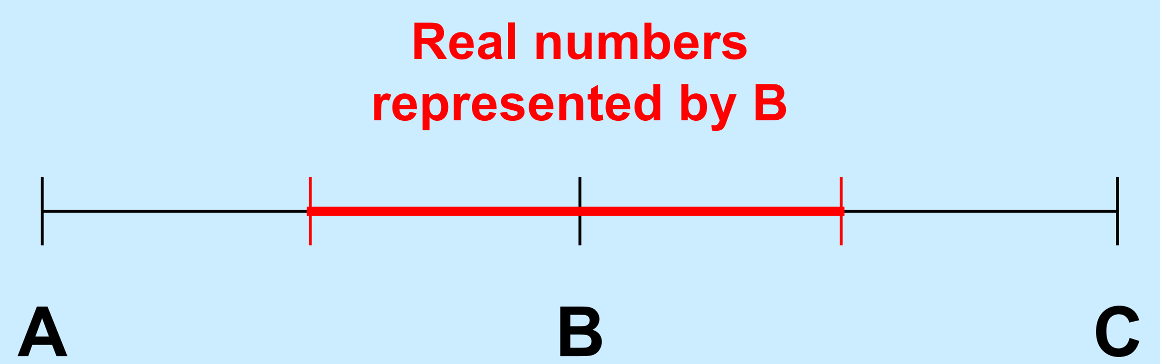


Mapping Real Numbers to Float

- If R is a real number, it's mapped to the floating-point number whose value is closest to R

Floats are Sets of Values

- If A, B, and C are successive floating-point values
  - e.g., 010001, 010010, and 010011
- B represents all real numbers from midway between A and B through midway between B and C



+/− Zero

- **Only one zero for ints**
  - an int is a single number, not a range of numbers, thus there can be only zero
- **Floating-point zero**
  - a range of numbers around the real 0
  - it really matters which side of 0 we're on!
    - » a very large negative number divided by a very small negative number should be positive

      $-\infty/-0 = +\infty$
    - » a very large positive number divided by a very small negative number should be negative

      $+\infty/-0 = -\infty$

Significance

- **Normalized numbers**
  - for a particular exponent value E and an S-bit significand, the range from 2^E up to 2^{E+1} is divided into 2^S equi-spaced floating-point values
    - » thus each floating-point value represents $1/2^S$ of the range of values with that exponent
    - » all bits of the signifcand are important
    - » we say that there are S significant bits – for reasonably large S, each floating-point value covers a rather small part of the range
      - high accuracy
      - for S=23 (32-bit float), accurate to one in 2^{23} (.0000119% accuracy)

Significance

- **Unnormalized numbers**
  - high-order zero bits of the significand aren't important
  - in 32-bit floating point, 0 00000000 00000000000000000000001 represents 2^{-149}
    - » it is the only value with that exponent: 1 significant bit (either 2^{-149} or 0)
  - 0 00000000 00000000000000000000010 represents 2^{-148}
    0 00000000 00000000000000000000011 represents $1.5*2^{-148}$
    - » only two values with exponent -148: 2 significant bits (encoding those two values, as well as 2^{-149} and 0)
  - fewer significant bits mean less accuracy
  - 0 00000000 00000000000000000000001 represents a range of values from $.5*2^{-9}$ to $1.5*2^{-9}$
  - 50% accuracy

Floating Point

- **Single precision (float)**

s	exp	frac
1	8-bits	23-bits

  – **range: ±1.8×10^{-38} – ±3.4×10^{38}, ~7 decimal digits**

- **Double Precision (double)**

s	exp	frac
1	11-bits	52-bits

  – **range: ±2.23×10^{-308} – ±1.8×10^{308}, ~16 decimal digits**

Quiz 1

Suppose *f*, declared to be a `float`, is assigned the largest possible floating-point positive value (other than +∞). What is the value of *g = f+1.0*?

a) 0

b) f

c) +∞

d) NaN

Float is not Rational …

- **Floating addition**
  - commutative: $a +_f b = b +_f a$
    - » yes!
  - associative: $a +_f (b +_f c) = (a +_f b) +_f c$
    - » no!
      - $2 +_f (1e38 +_f -1e38) = 2$
      - $(2 +_f 1e38) +_f -1e38 = 0$

Float is not Rational …

- **Multiplication**
  - commutative: $a *_f b = b *_f a$
    - » yes!
  - associative: $a *_f (b *_f c) = (a *_f b) *_f c$
    - » no!
      - $1e37 *_f (1e37 *_f 1e\text{-}37) = 1e37$
      - $(1e37 *_f 1e37) *_f 1e\text{-}37 = +\infty$

Float is not Rational …

- More …
  – multiplication distributes over addition:
  $a *_f (b +_f c) = (a *_f b) +_f (a *_f c)$
    » no!
    » $1e38 *_f (1e38 +_f -1e38) = 0$
    » $(1e38 *_f 1e38) +_f (1e38 *_f -1e38) = NaN$
  – insignificance:
    $x = y +_f 1$
    $z = 2 /_f (x -_f y)$
    $z == 2$?
    » not necessarily!
      • consider y = 1e38

CS 33

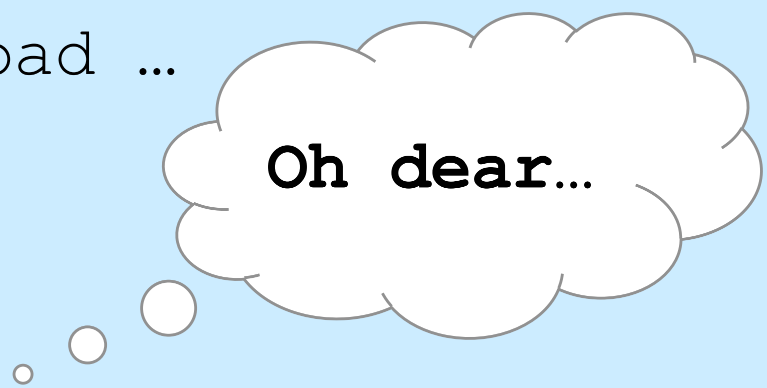
Signals Part 1

An Interlude Between Shells

- **Shell 1**
  - it can run programs
  - it can redirect I/O
- **Signals**
  - a mechanism for coping with exceptions and external events
  - the mechanism needed for shell 2
- **Shell 2**
  - it can control running programs

Whoops …

```
$ SometimesUsefulProgram xyz
Are you sure you want to proceed? Y
Are you really sure? Y
Reformatting of your disk will begin
in 3 seconds.
Everything you own will be deleted.
There's little you can do about it.
Too bad …
```



A Gentler Approach

- Signals
  - get a process's attention
    - » send it a signal
  - process must either deal with it or be terminated
    - » in some cases, the latter is the only option

Stepping Back …

- What are we trying to do?
  - interrupt the execution of a program
    - » cleanly terminate it

      or
    - » cleanly change its course
  - not for the faint of heart
    - » it's difficult
    - » it gets complicated
    - » (not done in Windows)

Signals

- Generated (by OS) in response to
  - exceptions (e.g., arithmetic errors, addressing problems)
    - » synchronous signals
  - external events (e.g., timer expiration, certain keystrokes, actions of other processes)
    - » asynchronous signals
- Effect on process:
  - termination (possibly producing a core dump)
  - invocation of a function that has been set up to be a signal handler
  - suspension of execution
  - resumption of execution

Signal Types

SIGABRT	*abort* called	term, core
SIGALRM	alarm clock	term
SIGCHLD	death of a child	ignore
SIGCONT	continue after stop	cont
SIGFPE	erroneous arithmetic operation	term, core
SIGHUP	hangup on controlling terminal	term
SIGILL	illegal instruction	term, core
SIGINT	interrupt from keyboard	term
SIGKILL	kill	forced term
SIGPIPE	write on pipe with no one to read	term
SIGQUIT	quit	term, core
SIGSEGV	invalid memory reference	term, core
SIGSTOP	stop process	forced stop
SIGTERM	software termination signal	term
SIGTSTP	stop signal from keyboard	stop
SIGTTIN	background read attempted	stop
SIGTTOU	background write attempted	stop
SIGUSR1	application-defined signal 1	stop
SIGUSR2	application-defined signal 2	stop

Sending a Signal

- `int kill(pid_t pid, int sig)`
  - send signal *sig* to process *pid*
- **Also**
  - *kill* shell command
  - type ctrl-c
    - » sends signal 2 (SIGINT) to current process
  - type ctrl-\
    - » sends signal 3 (SIGQUIT) to current process
  - type ctrl-z
    - » sends signal 20 (SIGTSTP) to current process
  - do something bad
    - » bad address, bad arithmetic, etc.

Handling Signals

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signo,
    sighandler_t handler);

sighandler_t OldHandler;

OldHandler = signal(SIGINT, NewHandler);
```

Special Handlers

- **SIG_IGN**
  - **ignore the signal**
  - `signal(SIGINT, SIG_IGN);`
- **SIG_DFL**
  - **use the default handler**
    - » **usually terminates the process**
  - `signal(SIGINT, SIG_DFL);`

Example

```
void sigloop() {
  while(1)
    ;
}

int main() {
  void handler(int);
  signal(SIGINT, handler);
  sigloop();
  return 1;
}
void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```

Digression: Core Dumps

- Core dumps
  - files (called "core") that hold the contents of a process's address space after termination by a signal
  - they're large and rarely used, so they're often disabled by default
  - use the ulimit command in bash to enable them

```
ulimit -c unlimited
```

  - use gdb to examine the process (post-mortem debugging)

```
gdb sig core
```

sigaction

```
int sigaction(int sig, const struct sigaction *new,
              struct sigaction *old);
struct sigaction {
   void (*sa_handler)(int);
   void (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t sa_mask;
   int sa_flags;
};

int main() {
   struct sigaction act; void myhandler(int);
   sigemptyset(&act.sa_mask); // zeroes the mask
   act.sa_flags = 0;
   act.sa_handler = myhandler;
   sigaction(SIGINT, &act, NULL);
   …
}
```


Example

```
int main() {
  void handler(int);
  struct sigaction act;
  act.sa_handler = handler;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, 0);

  while(1)
    ;
  return 1;
}

void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```

Quiz 2

```
int main() {
  void handler(int);
  struct sigaction act;
  act.sa_handler = hand
  sigemptyset(&act.sa_m
  act.sa_flags = 0;
  sigaction(SIGINT, &ac

  while(1)
    ;
  return 1;
}

void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```

You run the example program, then quickly type ctrl-C. What is the most likely explanation if the program then terminates?

a) **this "can't happen"; thus there's a problem with the system**
b) **you're really quick or the system is really slow (or both)**
c) **what we've told you so far isn't quite correct**

Waiting for a Signal …

```
signal(SIGALRM, RespondToSignal);

…

struct timeval waitperiod = {0, 1000};
      /* seconds, microseconds */
struct timeval interval = {0, 0};
struct itimerval timerval;
timerval.it_value = waitperiod;
timerval.it_interval = interval;

setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */
pause();  /* wait for it */
printf("success!\n");
```

Quiz 3

This program is guaranteed to print "success!".

a) no

b) yes

```
signal(SIGALRM, RespondToSignal);

…

struct timeval waitperiod = {0, 1000};
      /* seconds, microseconds */
struct timeval interval = {0, 0};
struct itimerval timerval;
timerval.it_value = waitperiod;
timerval.it_interval = interval;

setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */
pause();  /* wait for it */
printf("success!\n");
```

Masking Signals

```
setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */
```

No signals here, please!

```
pause();  /* wait for it */
```

Masking Signals

mask SIGALRM

```
setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */
```

No signals here

unmask and wait for SIGALRM

Doing It Safely

```
sigset_t set, oldset;
sigemptyset(&set);
sigaddset(&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, &oldset);
      /* SIGALRM now masked */
…
setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */

sigsuspend(&oldset);    /* unmask sig and wait */
/* SIGALRM masked again */

sigprocmask(SIG_SETMASK, &oldset, (sigset_t *)0);
      /* SIGALRM unmasked */
printf("success!\n");
```

Signal Sets

- **To clear a set:**

```
int sigemptyset(sigset_t *set);
```

- **To add or remove a signal from the set:**

```
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

- **Example: to refer to both SIGHUP and SIGINT:**

```
sigset_t set;

sigemptyset(&set);
sigaddset(&set, SIGHUP);
sigaddset(&set, SIGINT);
```

Masking (Blocking) Signals

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set,
      sigset_t *old);
```

- used to examine or change the signal mask of the calling process
  - » *how* is one of three commands:
    - SIG_BLOCK
      - the new signal mask is the union of the current signal mask and set
    - SIG_UNBLOCK
      - the new signal mask is the intersection of the current signal mask and the complement of set
    - SIG_SETMASK
      - the new signal mask is set

Signal Handlers and Masking

- **What if a signal occurs while a previous instance is being handled?**
  - **inconvenient …**
- **Signals are masked while being handled**
  - **may mask other signals as well:**

```
struct sigaction act; void myhandler(int);
sigemptyset(&act.sa_mask); // zeroes the mask
sigaddset(&act.sa_mask, SIGQUIT);
   // also mask SIGQUIT
act.sa_flags = 0;
act.sa_handler = myhandler;
sigaction(SIGINT, &act, NULL);
```

Timed Out!

```
int TimedInput( ) {
   signal(SIGALRM, timeout);
   …
   alarm(30);    /* send SIGALRM in 30 seconds */
   GetInput();   /* possible long wait for input */
   alarm(0);     /* cancel SIGALRM request */
   HandleInput();
   return(0);
nogood:
   return(1);
}

void timeout( ) {
   goto nogood;   /* not legal but straightforward */
}
```

Doing It Legally (but Weirdly)

```
sigjmp_buf context;

int TimedInput( ) {
   signal(SIGALRM, timeout);
   if (sigsetjmp(context, 1) == 0) {
      alarm(30);  // cause SIGALRM in 30 seconds
      GetInput(); // possible long wait for input
      alarm(0);   // cancel SIGALRM request
      HandleInput();
      return 0;
   } else
      return 1;
}

void timeout() {
   siglongjmp(context, 1);    /* legal but weird */
}
```

sigsetjmp/siglongjmp