Floating Point, Branching, and Assembler Directives

CS 217

Floating Point Instructions

- Performed by x87 floating point unit (FPU)
- Stack based and each item has 80-bits
  - Top of the stack: register ST0
  - Next: register ST1
  - …
  - Bottom: register ST7
- Load and store instructions
  - `fld, fst,fxch`, …
- Other instructions are FPU-specific
  - `Fadd/faddp, fsub/fsubp, fmul/fmulp, fimul/fimulp`,
  - …
- See Intel manual (volume 2) for the details

Floating Point

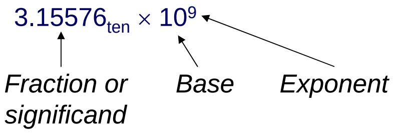
- Real numbers in mathematics

  $3.141592265\ldots_{ten}$ (π)

  $2.71828\ldots_{ten}$ (e)
- Scientific notation

  0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$ (seconds in a nanosecond)

  $3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a century)
- Floating point is like scientific notation



$3.15576_{ten} \times 10^{9}$

Fraction or significand

Base

Exponent

IEEE 754 Single Precision Floating Point

- General form for computer arithmetic

  $$(-1)^{S} \times F \times 2^{E}$$
  - S: sign of the floating point number
  - F: fraction or significand
  - E: exponent
- IEEE 754 single precision: 32-bit floating point

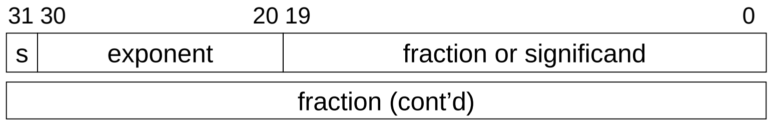
31	30 23	22 0
s	exponent	fraction or significant

- Questions:

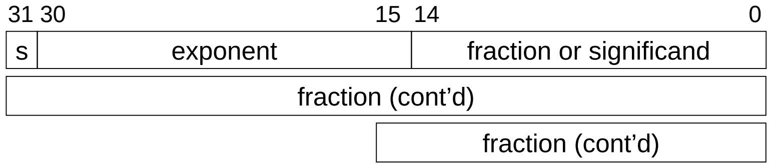
What is the smallest possible fraction?	$2.0_{ten} \times 10^{-38}$
What is the largest possible number?	$2.0_{ten} \times 10^{38}$
What is the largest 32-bit integer number?	4×10^{9}

IEEE 754 Double Precisions

- Double precision: 64-bit floating point
  - Sign bit + 52 bit fraction and 11-bit exponent
  - Approximate range: $2.0_{ten} \times 10^{-308}$ to $2.0_{ten} \times 10^{308}$



- Double extended precision: 80-bit floating point
  - Sign-bit + 63 bit fraction and 16-bit exponent
  - Approximate range: $2.0_{ten} \times 10^{-4932}$ to $2.0_{ten} \times 10^{4932}$



Increasing Precisions with Fewer Bits

- Normalization
  - Maximize the precision of fraction by adjusting exponent
    $0.000438 \times 10^4 = 0.438 \times 10^1$
  - In binary, normalization means

```
while (fraction's leading bit is 0) {
   fraction = fraction << 1;
   exponent--;
}
```

- 1 More bit in IEEE 754 standard
  - 0 has not leading 1, reserved exponent value 0 for it
  - For non-0 values, pack 1 more bit into the fraction, making the leading 1 bit of normalized binary numbers implicit
    $(-1)^S \times (1+ \text{fraction}) \times 2^E$
  - If we number the bits of the significant from left to right s1, s2, …
    $(-1)^S \times (1+ (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + \ldots) \times 2^E$

Floating Point Operations

- Use decimal floating point to demonstrate
  - 4-digit fraction
  - Exponent without bias
- Addition: $9.999_{ten} \times 10^1 + 1.610_{ten} \times 10^{-1}$
  - Align the numbers: $1.610_{ten} \times 10^{-1} = 0.01610_{ten} \times 10^1$
  - Add the fractions: $9.999_{ten} + 0.01610_{ten} = 10.015_{ten}$
  - Normalize the result: $10.015_{ten} \times 10^1 = 1.0015_{ten} \times 10^2$
  - Rounding to 4 digits: $1.002_{ten} \times 10^2$
- Multiply: $1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$
  - Add exponents: 10 + (-5) = 5
  - Multiply the fractions: $1.110_{ten} + 9.200_{ten} = 10.212_{ten}$
  - Normalize the result: $10.212_{ten} \times 10^5 = 1.021_{ten} \times 10^6$
  - Sign calculation: +1

IEEE 754 Standard

- Why is the sign bit away from the rest of the fraction?
- Should *exponent* be two's complement?
  - Examples:

$1.0_{two} \times 2^{-1}$	0	1 1 1 1 1 1 1 1	0 0 0 0 0	. . .	0
$1.0_{two} \times 2^{1}$	0	0 0 0 0 0 0 0 1	0 0 0 0 0	. . .	0

  - For simplified sorting, we cannot treat exponent as unsigned integer
- Bias in IEEE 754 standard
  - Use 127_{ten} for single (1023 for double, 16383 for double extended)
  - Single precision examples
    -1: $-1 + 127_{ten} = 126_{ten} = 0111\ 1110_{two}$
    +1: $1 + 127_{ten} = 128_{ten} = 1000\ 0000_{two}$
  - General representation
    $(-1)^S \times (1+ \text{fraction}) \times 2^{(\text{exponent-bias})}$
  - All operations will have to apply bias

Branching Instructions

- Unconditional branch

  ```
  jmp addr
  ```

- Conditional branch
  - Recall the six flags in EFLAGS registers (ZF, SF, CF, OF, AF, PF)
  - Every arithmetic instruction sets the flags according to its result

    ```
       cmpl %ebx, %eax
       je    L1
       ...                # ebx != eax
    L:
       ...                # ebx == eax
    ```

  - IA32 has conditional branch instructions for all these flags individually and some combinations

The Six Flags

- CF: Carry flag
  - Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; clear otherwise;
  - Indicates an overflow for unsigned integer arithmetic
  - Can be modified with `stc, clc, cmc, bt, bts, btr,` and `btc`
- ZF: Zero flag
  - Set if the result is zero; clear otherwise
- SF: Sign flag
  - Set equal to the most-significant bit of the result
- OF: Overflow flag
  - Set if the result is too large to fit or too small to fit (excluding the sign bit); clear otherwise. It is useful for signed (two's complement) operations
- PF: Parity flag
  - Set if the least-significant byte of the result contains an even number of 1 bits; clear otherwise
- AF: Adjust flat
  - The CF for BCD arithmetic

Conditional Branch Instructions

- For both signed and unsigned integers

`je`	(ZF = 1)	Equal or zero
`jne`	(ZF = 0)	Not equal or not zero

- For signed integers

`jl`	(SF ^ OF = 1)	Less than
`jle`	((SF ^ OF) || ZF) = 1)	Less or equal
`jg`	((SF ^ OF) || ZF) = 0)	Greater than
`jge`	(SF ^ OF = 0)	Greater or equal

- For unsigned integers

`jb`	(CF = 1)	Below
`jbe`	(CF = 1 || ZF = 1)	Below or equal
`ja`	(CF = 0 && ZF = 0)	Above
`jae`	(OF = 0)	Above or equal

- For AF and PF conditions, FPU, MMX, SSE and SSE2
  - See the Intel manual (volume 1 and 2)

Branching Example: if-then-else

C program **Assembly program**

```
   if (a > b)           movl a, %eax
                        cmpl b, %eax    # compare a and b
                        jle  .L2        # jump if a <= b

      c = a;            movl a, %eax    # if a > b
                        movl %eax, c
                        jmp     .L3
   else
      c = b;         .L2:               # a <= b
                        movl b, %eax
                        movl %eax, c

                     .L3:               # finish
```

Branching Example: for Loop

C program

```
for (i=0; i<100; i++){
  ...
}
```

Assembly program

```
    movl $0, %edx     # i = 0;
.L6:
    ...
    incl %edx         # i++;
    cmpl $99, %edx
    jle  .L6          # loop if i <= 99
```

Branching Example: while Loop

C program

```
while ( a == b )
  statement;
```

Assembly program

```
.L2:
    movl    a, %eax
    cmpl    b, %eax
    je      .L4
    jmp     .L3
.L4:
    statement
    jmp     .L2
.L3:
```

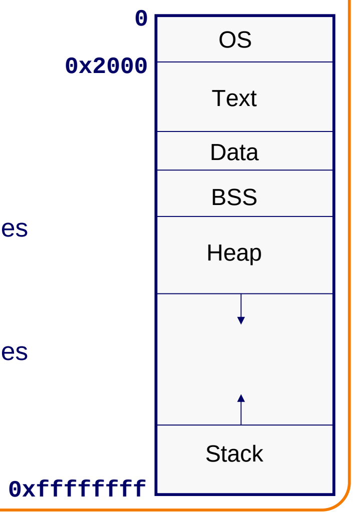
- Can you do better than this?

Assembler Directives

- Identify sections
- Allocate/initialize memory
- Make symbols externally visible or invisible

Identifying Sections

- Text (.section .text)
  - Contains code (instructions)
  - Default section
- Read-Only Data (.section .rodata)
  - Contains constants
- Read-Write Data (.section .data)
  - Contains user-initialized global variables
- BSS (.section .bss)
  - Block starting symbol
  - Contains zero-initialized global variables



Allocating Memory in BSS

- For global data
  `.comm symbol, nbytes, [desired-alignment]`
- For local data
  `.lcomm symbol, nbytes, [desired-alignment]`
- Example

```
.section .bss         # or just .bss
.equ   BUFSIZE 512    # define a constant
.lcomm BUF, BUFSIZE   # allocate 512 bytes
                      #     local memory for BUF
.comm x, 4, 4         # allocate 4 bytes for x
                      #     with 4-byte alignment
```

Allocating Memory in Data

- Specify
  - Alignment
    `.align nbytes`
  - Size and initial value

```
.byte   byteval1 [, byteval2 ...]
.word   16-bitval1 [, 16-bitval2 ...]
.long   32-bitval1 [, 32-bitval2 ...]
```

- Read-only data example: `const s[] = “Hello.”;`

```
    .section .rodata        # or just .rodata
s:  .string “Hello.”        # a string with \0
```

- Read-Write data example: `int x = 3;`

```
    .section .data          # or just .data
    .align 4                # alignment 4 bytes
x:  .long 3                 # set initial value
```

Initializing ASCII Data

- Several ways for ASCII data

```
.byte  150,145,154,154,157,0  # a sequence of bytes

.ascii “hello”                # ascii without null char
.byte 0                       # add \0 to the end

.ascii “hello\0”

.asciz “hello”                # ASCII with \0

.string “hello”               # same as .asciz
```

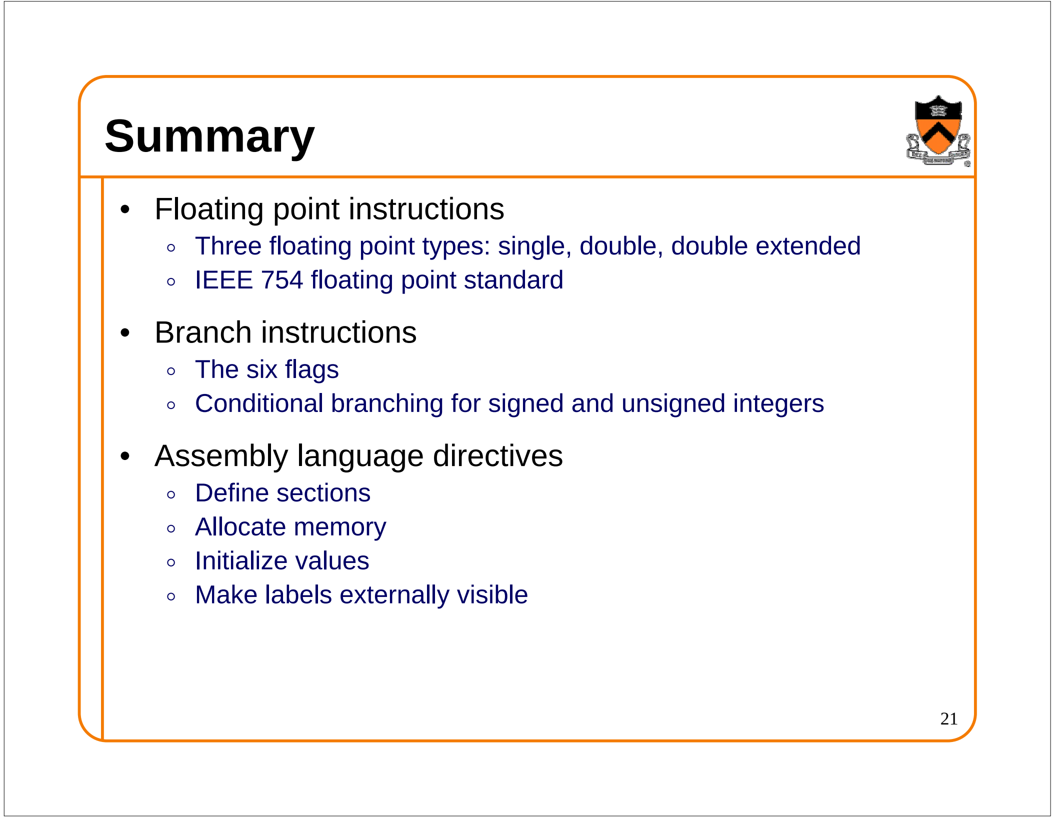
Making Symbols Externally Visible

- Default is local
- Specify globally visible
  `.globl symbol`
- Example: **`int x = 1;`**

```
    .data
    .globl x          # declare externally visible
    .align 4
x:  .long  2
```

- Example: **`foo(void){...}`**

```
    .text
    .globl foo
foo:
    ...
    leave
    return
```



Summary

- Floating point instructions
  - Three floating point types: single, double, double extended
  - IEEE 754 floating point standard
- Branch instructions
  - The six flags
  - Conditional branching for signed and unsigned integers
- Assembly language directives
  - Define sections
  - Allocate memory
  - Initialize values
  - Make labels externally visible